Trabajo de Fin de Grado

Grado en Ingeniería Informática

Simulación de sistemas autónomos de transporte

Simulation of autonomous transportation systems

Abián Torres Torres

La Laguna, 1 de julio de 2018

D. **Rafael Arnay del Arco**, con N.I.F. 78.569.591-G profesor Ayudante Doctor de Universidad adscrito al Departamento de Nombre del Departamento de la Universidad de La Laguna, como tutor

D. **Iván Castilla Rodrígez**, con N.I.F. 78.565.451-G profesor Ayudante Doctor de Universidad adscrito al Departamento de Nombre del Departamento de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Simulación de sistemas autónomos de transporte"

ha sido realizada bajo su dirección por D. **Abián Torres Torres**,

con N.I.F. 54.111.308-J.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 1 de julio de 2018

Agradecimientos

Agradecer a mis tutores Rafael Arnay e Iván Castilla, así como al resto de sus compañeros, el interés mostrado por transmitir parte de su conocimiento y haber hecho posible la ejecución de este trabajo.

Agradecer también a mis compañeros, que han hecho posible que esta etapa haya sido más amena y entretenida.

Por último, quería destacar el apoyo de mis padres y familiares, ya que sin ellos nada de esto hubiera sido posible.

Licencia

Resumen

En la actualidad desarrollamos multitud de artefactos tecnológicos que requieren de una precisión elevada a la hora de llevar a cabo, de manera satisfactoria, los objetivos que previamente hemos definido. Por consiguiente, requerimos de algún tipo de mecanismo que nos permita representar o reproducir el comportamiento de nuestros modelos sistémicos a nivel computacional.

Procediendo de esta manera, tendremos la capacidad de obtener nuevas conclusiones o detectar ciertos detalles que en el análisis de un sistema real podríamos pasar por alto. El objetivo de este trabajo es la aplicación de esta metodología sobre sistemas de transporte autónomo.

Por tanto, el primer paso a realizar ha de ser un análisis previo del modelo de sistema de trasporte autónomo que queremos plasmar en nuestra simulación. En nuestro caso, examinando el tipo de herramientas que vamos a usar, centraremos el grueso del trabajo en el estudio y desarrollo de un software que nos de la posibilidad de simular sistemas de vehículos autónomos o robots de interiores (indoor). En concreto, el caso de uso que modelaremos será un sistema de automatización de sillas de ruedas para entornos hospitalarios.

El propósito de este software es la automatización de la ejecución de dos niveles de simulación. En primera instancia, deseamos obtener resultados acerca del comportamiento del robot y sus sensores con el entorno (simulación a bajo nivel). Sin embargo, esto es sólo un aspecto del problema, ya que de igual forma, nos interesa simular el impacto que tendría la implantación de un sistema en términos de coste/efectividad (simulación a alto nivel).

Palabras clave: Simulación, Vehículo Autónomo, Robótica, Transporte.

Abstract

Nowadays, we develop a great quantity of technological devices that require high precision in order to carry out satisfactorily the objectives that we have previously defined. Therefore, we require some type of mechanism that allows us to represent or reproduce the behavior of our systemic models at a computational level.

Proceeding in this way, we will have the capacity to obtain new conclusions or to detect certain details that in the analysis of a real system we could ignore. The objective of this work is based mainly on the application of this methodology on autonomous transport systems.

Therefore, the first step to be carried out must be a previous analysis of the autonomous transport system model that we want to define in our simulation. In our case, examining the type of tools we are going to use, we will focus the bulk of the work on the study and development of a software that gives us the possibility of simulating autonomous vehicle systems or interior robots (indoor). Specifically, the use case that we will model will be a wheelchair automation system for hospital environments.

The purpose of this software instrument is the automation of the execution of two levels of simulation. In the first instance, we want to obtain results about the behavior of the robot and its sensors with the environment (low level simulation). However, this is only one aspect of the problem, since similarly, we are interested in simulating the impact that the implementation of a system would have in terms of cost / effectiveness (high level simulation).

Keywords: Simulation, Autonomous Vehicle, Robotics, Transportation.

Índice general

Índice de ilustraciones

Índice de tablas

1. Introducción

1.1 Motivación

El diseño e implementación de herramientas en el campo de la **automatización industrial** está jugando un papel fundamental, tanto a un nivel de modelo social como económico.

Un estudio [1] realizado por la Universidad de Valencia y BBVA Research prevé que el 36% de los empleos en España están en riesgo de automatización. Hasta la fecha, y sobre todo a causa del gran avance que han sufrido campos de estudio como son la robótica y la inteligencia artificial, han sido publicados numerosos artículos en los que se recogen de manera negativa estos hechos. Sin embargo, más allá del impacto económico pronosticado, múltiples análisis elaborados por expertos en la materia confluyen hacia conclusiones totalmente contrarias. Un informe [2] elaborado por el Observatori de la Industria 4.0, indica que hasta 2030 el sector de la automatización generará, solo en Cataluña, más de 13.000 empleos de los que dejarán de existir.

Los trabajos en peligro suelen asociarse con perfiles de baja formación y cuya función se basa en tareas repetitivas, de tal modo que, las ventajas que experimentará el sistema radican en la posibilidad de establecer procesos en ejecución de forma ininterrumpida, niveles de calidad óptimos, ahorro en costes, reducción significativa en los tiempos de producción, mejora de la seguridad del personal, producción más flexible y mejora en cuanto al flujo de los datos en red, reduciendo así el tiempo de reacción.

Dentro de la automatización industrial, una de las tecnologías a destacar es la **conducción autónoma**. Esta es extrapolable a muchos ámbitos dentro del entorno laboral, así como a las múltiples situaciones o necesidades de cualquier ciudadano de a pie. No obstante, se trata de un área en pleno desarrollo, basta con echar un vistazo a las noticias de cada día. En la actualidad, se han documentado múltiples casos en los que este tipo de tecnología ha podido llegar a causar daños materiales o humanos más allá de factores externos. En consecuencia, necesitamos definir modelos que nos faciliten la labor de predecir el comportamiento e impacto que pueden llegar a tener

estos artefactos en un entorno real antes de su implantación. Por lo tanto, el objeto de estudio de este proyecto se basa fundamentalmente en la realización de una herramienta que ofrezca la capacidad de definir nuestros modelos y analizar su comportamiento bajo la ejecución de una **simulación computacional**.

1.2 Antecedentes y estado actual.

La simulación de un sistema o proceso real por medio de modelos a nivel computacional permite llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias. Una fase de simulación comprende, por convenio, distintas tareas a llevar a cabo, entre las cuales se encuentra una etapa de experimentación e interpretación por las que se realiza un análisis de la sensibilidad de los índices requeridos con el fin de tomar decisiones a posteriori que puedan llegar a afectar de manera positiva el comportamiento del sistema según el contexto en el que se trabaje. Esta metodología de desarrollo lleva aplicándose en el campo de la conducción autónoma desde el año 1950, con algunos ensayos elaborados por General Motors [3].

En concreto, este proyecto pretende abordar un análisis del comportamiento de sistemas de vehículos de transporte autónomos desde una perspectiva funcional, con el matiz de que también contará con un nivel de trabajo en el que se realizará un análisis de coste-efectividad, tal que estas herramientas quedarán integradas en un único módulo, permitiendo así tener más información en tiempo real que ayude a la obtención de conclusiones. En líneas generales, no es común encontrar trabajos con características similares al que describimos en este documento. ANSYS [4] es una de las potencias mundiales en el ámbito de las simulaciones ingenieriles. Entre sus productos ofrece algunas herramientas estructuradas en distintos niveles de simulación para vehículos autónomos. En concreto, ofrece la simulación del sistema del escenario de conducción, modelado y desarrollo de algoritmos, análisis de seguridad funcional, simulación del rendimiento del sensor, simulación del hardware de electrónica y simulación de semiconductores.

En 2016, desde el departamento de transporte de la reconocida consultora británica ATKINS, especializada en diseño, ingeniería y dirección de proyectos, se llevó a cabo una pequeña investigación [5] en la que se analiza el impacto de redes de vehículos autónomos conectados en tránsito urbano. En ella se establecen dos niveles de simulación, un nivel funcional y un segundo nivel que trabaja sobre el cálculo de ciertos índices asociados la seguridad vial, como puede ser los retardos en las llegadas al correspondiente destino, las distancias entre vehículos, las medias de tiempos por jornada, el volumen por secciones, tiempo de viaje, etc.

Sin embargo, es posible localizar multitud de desarrollos orientados a la implementación de estos dos niveles de simulación de manera independiente, especialmente a nivel funcional. Basta con observar el mercado de vehículos autónomos para comprobar que se trata de un campo de estudio que ha experimentado muchos avances. Grandes marcas como Tesla, Google, Hyundai o Toyota ya han mostrado en funcionamiento las primeras versiones de sus modelos iniciales, cuyas tecnologías pretenden extrapolar a otros ámbitos similares al de la conducción autónoma, como puede ser el proyecto BLAID [6] de Toyota, cuya finalidad es la elaboración de un collar que pueda ayudar a personas con visión notablemente reducida o nula a circular por entornos urbanos de manera segura. A pesar de ello, estos prototipos siguen sufriendo pequeños errores en el sistema que hacen dudar de si realmente, hoy en día, pueden llegar a ser capaces de alcanzar un nivel de autonomía total. Por este motivo, entornos usados para simulaciones computacionales de este tipo requieren un plus en términos de eficiencia, así como el uso de herramientas más completas y precisas, de manera que, provean de los útiles necesarios a investigadores en el campo.

Algunos de los avances se han presentado desde ramas de la inteligencia artificial, como el Machine Learning**,** o el desarrollo de herramientas software y hardware más sofisticadas para el diseño gráfico que permitan esbozar y procesar elementos de una manera más realista. La meta reside en contemplar la mayor cantidad de situaciones o inconvenientes posibles sobre el entorno simulado por el que transitan los vehículos.

1.3 Transporte Autónomo

Un robot o vehículo autónomo de transporte pretende imitar las capacidades humanas de manejo y control, o incluso, mejorarlas. Con este fin, el vehículo debe ser capaz de percibir el entorno que le rodea y navegar en consecuencia. El conductor debe elegir el destino final en un **mapa**, pero no requerir su influencia en ninguna operación mecánica del robot.

Para percibir el entorno, podemos recurrir a algunas de las técnicas desarrolladas hoy en día, como el sensor láser, radar, LIDAR o/y sistemas de posicionamiento global y visión por computador.

A nivel software, estos aparatos deben funcionar en base a múltiples sistemas de **tiempo real**, que deben operar de manera coordinada mediante un sistema de control. En general, es necesario un sistema de localización, de percepción del entorno y de planificación. Estos sistemas basan su funcionamiento en un conjunto de **algoritmos** que permiten transformar la información del entorno, mediante algunos componentes hardware, como los mencionados previamente, que permitan al vehículo tomar decisiones en la navegación.

Si bien, según la Sociedad de ingenieros Automotrices (**SAE** [7]) existen seis niveles de autonomía dependiendo de las capacidades del vehículo. Esta clasificación está más enfocada a robots tipo coche por su uso tan extendido, no obstante, es posible extrapolar su aplicación sobre robots enfocados a otros fines:

0. **Solo un conductor**: solo cuenta con sistemas que emitan algún tipo de emergencia.
1. **Asistente de conducción:** incluye sistemas para el control de la velocidad y mantener el vehículo en el carril.
2. **Semi-autonomía:** El conductor debe permanecer en alerta por si su intervención es requerida en alguna situación que el vehículo no pueda solucionar. Un ejemplo es el Mercedes-Benz Clase E, en venta desde 2016, el primer vehículo capaz de mantener su trayectoria dentro de un carril dónde no se disponen líneas o señales delimitadoras.
3. **Autonomía controlada:** Los vehículos son capaces de navegar de forma autónoma en entornos controlados como, por ejemplo, autopistas. En algunos modelos, como el Tesla S, este sistema también asegura la atención del conductor al comportamiento del vehículo mediante técnicas de visión por computador.
4. **Alto nivel de autonomía:** No es necesaria la supervisión de un conductor, eso sí, el coche debe tener la suficiente información del entorno. Por ejemplo, un entorno no propenso para la circulación de un vehículo de estas características puede ser aquel donde no se tenga conocimiento del mapa del medio y el terreno sea irregular. Volvo, Google o Uber están desarrollando coches a este nivel.
5. **Autonomía total:** Puede circular por cualquier carretera o ciudad siempre y cuando sea legal la conducción autónoma. El robot tiene la capacidad de reaccionar ante cualquier imprevisto. Empresas como Microsoft, Baidu o Apple, entre otras, han comenzado proyectos que tiene como objetivo la creación de coches de estas características. Según Volkswagen, se prevé que los coches de autonomía total lleguen al mercado en el año 2025, con su modelo I.D. VIZZION. Por otro lado, Tesla sostiene, con un cierto grado de incertidumbre, que en 2019 lanzará su primer modelo de navegación autónoma, catalogado de nivel cinco.

Por lo tanto, se trata de un campo de estudio en plena evolución, y por suerte, las predicciones apuntan a que en un par de años contaremos con vehículos completamente autónomos que nos faciliten aún más el desarrollo de nuestras vidas.

1.4 Concepto de simulación

Una **simulación** no es más que la representación de un modelo **en acción**. Es decir, describe el comportamiento de un **modelo** a lo largo del tiempo, en base a unos datos de entrada, que, partiendo de un estado inicial, proporcionará cierta información de relevancia en cuanto al comportamiento del modelo o la representación de la estructura a simular. Un modelo puede estar compuesto por estructuras, parámetros y algoritmos, como veremos en próximos capítulos.

Las primeras simulaciones efectuadas, a mediados de los cuarenta, tuvieron como finalidad medir el comportamiento de sistemas de balística. Desde entonces, ha tomado un grado de importancia mayor, sobre todo en el ámbito de la ciencia y la tecnología.

Este crecimiento tan extendido de la simulación como técnica se debe principalmente a la notable mejora de las herramientas informáticas en términos computacionales. En la actualidad, las máquinas son capaces de desarrollar cálculos matemáticos más complejos en un tiempo mucho más reducido. Además, las simulaciones son aplicadas en una amplia lista de campos de estudio como la química, la biología, la matemática, la física o desarrollos que persiguan obtener conclusiones en términos de comportamiento social o económico.

1.4.1 Simulación por eventos discretos

Una **simulación por eventos discretos** es una técnica usada en el ámbito de la informática para el modelado dinámico de sistemas. Se caracteriza por un control del tiempo que permite avanzar a este en **intervalos variables**, en función de la planificación de tales eventos en un futuro. Un requisito para aplicar esta técnica es que las variables que definen el sistema no cambien el comportamiento durante el intervalo simulado. A continuación, definimos los conceptos principales para tener en cuenta cuando realizamos una simulación de estas características.

- **Tiempo de simulación**: Valor del tiempo que el simulador puede avanzar a una velocidad superior al de un reloj común. Por lo tanto, mediante este parámetro podremos **acelerar** el tiempo de simulación. No obstante, las herramientas orientadas a simulaciones por eventos discretos ofrecen la capacidad de ejecutar la simulación en **tiempo real**, es decir, el avance del tiempo de simulación es, exactamente, de un segundo por cada segundo del tiempo real.

- **Evento**: Suceso que hace cambiar las variables de estado del sistema. Un evento puede pertenecer a una entidad o actor en el sistema y es ejecutado en tiempo constante.
- **Entidad**: Sienta las bases del sistema de simulación, el cual gira entorno a este concepto formando un sistema compuesto. Pueden existir múltiples instancias de un tipo de entidad.
- **Actividad**: Secuencia de eventos pertenecientes a una entidad que cierran un ciclo funcional. A diferencia de un evento, esta se produce en un intervalo de tiempo de simulación no puntual. Además, se debe establecer algún tipo de mecanismo que permita establecer la comunicación entre los distintos procesos.

Existen diferentes maneras de plantear un modelo de simulación de eventos discretos. Son lo que llamamos "perspectivas" o "worldviews" en inglés. Una de estas perspectivas, y una de las más populares, especialmente en software comerciales, es la orientada al proceso. Desde esta perspectiva, nos interesa identificar los procesos que existen en el sistema que queremos simular. No existe un estándar para la definición de estos procesos; sin embargo, en 1999 el profesor Wil van der Aalst, de la Universidad tecnológica de Eindhoven, y el profesor Arthur ter Hofstede de la Universidad tecnológica de Queensland establecieron el comienzo de una iniciativa que persigue establecer una base conceptual a la tecnología de procesos. En concreto, este trabajo denominado **Workflow Patterns** [8] (del inglés, Patrones de Flujo de trabajo), examina un conjunto de perspectivas (flujo de control, de datos, de recursos y de manejo de excepciones) que debe cumplir todo lenguaje de flujo de trabajo o lenguaje de modelado de procesos de negocio.

Para medir el impacto de la implantación del sistema de navegación autónoma que implementemos, tras obtener datos del comportamiento del vehículo desde una primera simulación, usaremos la herramienta PSIGHOS [9], desarrollada en **Java** por investigadores de la Universidad de La Laguna. Ésta proporciona las funcionalidades necesarias para llevar a cabo una simulación por eventos discretos siguiendo los patrones de flujo de trabajo mencionados.

1.5 Objetivo principal

El objetivo principal de este proyecto es el diseño e implementación de un software capaz de analizar, mediante simulación, tanto el comportamiento como el impacto de un sistema de transporte autónomo en un entorno real. Esta simulación, de carácter modular, quedará dividida en **dos niveles**, acorde a las necesidades mencionadas previamente, dado que, el modelo de interacción de un vehículo autónomo con el entorno sigue un conjunto de procedimientos dispar al que buscamos

en un estudio que explora conclusiones en términos de coste/efectividad. A pesar de todo, ambos módulos están directamente relacionados en base a ciertos parámetros como podrían ser el tiempo de trayectoria, número de fracasos durante la planificación, etc. En consecuencia, el caso de uso seleccionado determinará que modelos se pondrán en funcionamiento en cada una de las capas de simulación, modificando los parámetros que se crean oportunos.

1.6 Objetivos específicos

- Construir un sistema que simule el **comportamiento** del vehículo en una representación de un entorno real.
    - Proyectar un modelo del vehículo autónomo o robot en función del caso de uso deseado. Esta tarea conlleva el estudio de la arquitectura del robot, así como los distintos componentes hardware necesarios para su funcionamiento.
    - Esbozar el modelo de los distintos entornos con los que interactuará nuestro robot.
    - Integrar los modelos previamente definidos en un simulador realista que ofrezca la capacidad de establecer la apariencia visual y la interacción entre los distintos componentes.
    - Definir y configurar los algoritmos necesarios para llevar a cabo los distintos procesos propios del funcionamiento de un robot autónomo.
    - Adaptar los valores, según el caso de uso, de los distintos parámetros a los que se encuentra sujetos cada uno de los algoritmos.
    - Construir una interfaz de usuario que dote al usuario de la facilidad de establecer los distintos parámetros para esta capa de ejecución y ejecutar las tareas pertinentes.

- Construir un sistema que simule el **impacto** que tendría la implantación del sistema de transporte autónomo en un caso de uso real.
    - Definir nuestro modelo a partir de patrones de flujo de trabajo (**Workflow Patterns**), de tal manera que podamos efectuar una simulación de eventos discreto orientada al proceso. Este paso estará sujeto al caso de uso que se pretenda estudiar.
    - Implementar nuestro modelo acorde con el problema que pretendemos resolver.

    - Diseñar e implementar una interfaz de usuario que ofrezca la habilidad de seleccionar que información de comportamiento emplear para la simulación a alto nivel, acorde a diversos parámetros de filtrado o selección. Asimismo, ésta presentará de manera visual los resultados obtenidos.

- Establecer un mecanismo de **comunicación** entre los dos módulos de simulación.
    - Definir el formato de mensaje asociado a los resultados obtenidos en la simulación a alto nivel. Por ejemplo, el tiempo de trayectoria o el número de fracasos.
    - Implementar un servicio de envío de los resultados para el módulo de bajo nivel.
    - Implementar un servicio de recepción de los resultados para el módulo de alto nivel.

- Implementar un servicio para dar persistencia a los datos generados.
    - Implementar el servicio de acceso a la base de datos en el bajo nivel.
    - Implementar el servicio de acceso a la base de datos en el alto nivel.

1.7 Caso de uso

El objetivo principal de este proyecto consiste en la creación de un framework, que ofrezca los métodos básicos, para ejecutar una simulación con las características descritas. Si bien, debemos asegurar su correcto funcionamiento, así como el cumplimiento de los requisitos previamente estudiados. De manera que, se ha modelo un caso de uso, cuyo estudio aún sigue en desarrollo por investigadores de la Universidad de La Laguna.

En concreto, la estructura a modelar consiste en la implantación de un sistema de sillas de ruedas autónomas en un entorno hospitalario. La finalidad de su ejecución reside en la simplificación de las labores de los bedeles, de tal manera que, el tiempo destinado al desplazamiento de pacientes en sillas de ruedas se vea reducido a casos estrictamente necesarios donde la situación no sea abordable por una silla de ruedas autónoma. Una tarea común puede ser el desplazamiento de los pacientes que requieran de estos artefactos desde la entrada del hospital hasta la sala de consulta correspondiente. Sin embargo, el propósito general de este proyecto reside en que la

herramienta sea extrapolable a otros ámbitos de uso, como podría ser un aeropuerto, un hotel, o un muelle de carga y descarga.

Contextualizando el caso de uso sobre las funcionalidades del framework, el análisis del desplazamiento de las sillas de ruedas corresponde a la simulación a bajo nivel, donde se tomarán las medidas necesarias. En concreto, para definir nuestra simulación a alto nivel, en la cual modelamos el conjunto de actividades necesaria para el caso de uso, necesitaremos los tiempos de desplazamiento tomados desde el bajo nivel.

Con la finalidad de describir el problema en cuestión, a continuación, disponemos una ilustración en la cual es comprendido el conjunto de actividades, elementos y recursos necesarios para la resolución de la simulación de eventos discretos:

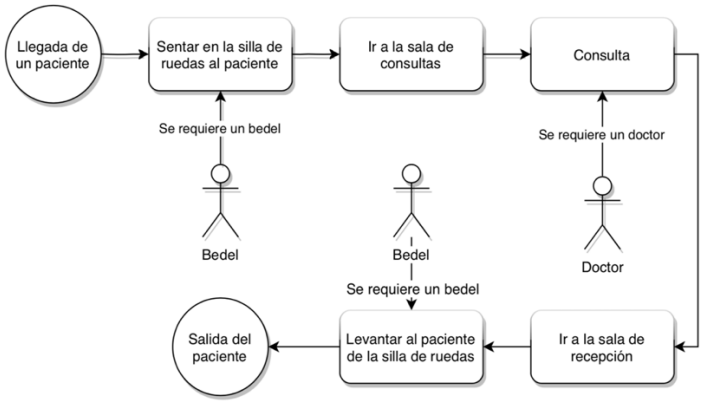

Ilustración 1: Esquema del modelo a implementar a alto nivel. Recuperado de fuente propia.

2. Metodología

En esta sección revisaremos las distintas herramientas empleadas para la ejecución del proyecto, así como las decisiones técnicas tomadas con el fin de solventar los retos e inconvenientes presentes durante el desarrollo del proyecto.

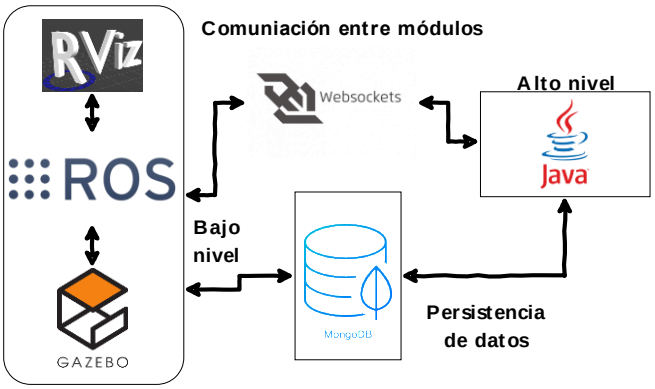

Ilustración 2: Estructura global del sistema de simulación. Recuperado de fuente propia.

El esqueleto general consta de dos módulos, los cuales representan los dos niveles de simulación. En la primera capa nos apoyaremos en las siguientes tres herramientas: ROS [10] (en inglés Robotic Operating System), Rviz [11] y Gazebo [12]. Mediante estos elementos conseguiremos llevar a cabo la reproducción del comportamiento del sistema robótico (Destacar el uso principal del cliente de ROS para **Python**, rospy). En la segunda capa, conectada con la primera via **WebSockets** [13], aprovecharemos la disponibilidad de la librería PSIGHOS.

Sin embargo, debemos idear algún mecanismo que ofrezca la capacidad de dar persistencia a los resultados obtenidos para posteriores iteraciones o análisis. Para ello, utilizaremos una base de datos no relacional MongoDB [14], en vista de las facilidades que ofrece de cara a la manipulación eficiente de los datos en términos temporales de consulta e inserción. Además, ésta podrá ejercer como mecanismo de intercambio de datos offline entre los dos módulos.

2.1 Simulación a bajo nivel

2.1.1 Tecnologías utilizadas

2.1.1.1 ROS (Robot Operating System)

ROS es una plataforma desarrollada principalmente con el objetivo de ofrecer una vía de elaboración de software comprendido en el campo de la robótica. Este provee la funcionalidad de un sistema operativo en un clúster heterogéneo por medio de un conjunto de servicios y librerías.

Fue desarrollado originalmente en 2007 por el laboratorio de inteligencia artificial de Stanford con el fin de dar soporte al proyecto STAIR. Desde 2008, el desarrollo y mantenimiento continua en manos de Willow Garage, una incubadora de empresas y un laboratorio de investigación robótica dedicada a la creación de **código abierto** para las aplicaciones para robots personales.

En la actualidad, ROS provee servicios estándar como abstracción del hardware, control de dispositivos de bajo nivel, paso de mensajes entre procesos, etc. Está basado en una arquitectura en **grafo**, donde el procesamiento de los datos toma lugar en los nodos, los cuales pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores.

Las principales **ventajas** que presenta ROS son las siguientes:

- Permite el uso de distintos lenguajes de programación como Python, C++ y Lisp.
- Puede ser ejecutado sobre máquinas tipo UNIX como Mac OS X, Ubuntu, Debian, Fedora o Gentoo. En nuestro caso, hemos optado por la distribución de Ubuntu.
- Cuenta con una enorme comunidad de desarrollo compuesta por expertos en la materia, aficionados, fabricantes de hardware o incluso, empresas que dan soporte, como Google.
- Proporciona una gran cantidad de ejemplos y librerías dedicadas a algoritmos de navegación, visión artificial, etc.

Estructura de trabajo en ROS

ROS permite organizar y descargar nuestros ficheros de programa de una forma más cómoda por medio de **repositorios**. Estos pueden estar compuestos por uno o

más paquetes y pilas. Un **paquete** contiene todos los ficheros de configuración y ejecución necesarios para lanzar nuestros **nodos** o ejecutables, en concreto, los ficheros encargados de esta función siguen la extensión **launch**. Además, estamos en la disposición de distribuir paquetes que compartan una misma funcionalidad de manera única por medio de las **pilas**. Una pila sería el equivalente a una librería en otros sistemas de programación.

En nuestro entorno, por motivos de agilidad, los ficheros launch serán ejecutados desde **scripts de bash**. La estructura del repositorio diseñado es descrita en la Tabla 1:

Paquete	**Descripción**	**Contenido**
control	Controles necesarios para el movimiento del robot.	• **control.yaml**: configuración del motor del robot. • **control.launch**: inicio del nodo **robot_state_publisher** y **controller_spawner** , necesarios para generar los comando de movimiento del robot y examinar su estado.
costum_msgs	Mensajes personalizados de ROS cuyo contenido define los resultados de una simulación a bajo nivel.	• **GlobalSegmentResultsMsg.msg**: resultados globales para un segmento individual de la trayectoria. • **GlobalSimulationResultsMsg.msg**: resultados globales de la simulación. • **IndividualIterationResults.msg**: resultados de la trayectoria para una iteración de experimento. • **IndividualSegmentResults.msg**: resultados de un segmento para una iteración de experimento. • **SegmentMetadata.msg**: información de interés asociada a un segmento. • **IndividualSegmentResults.msg**: información de interés asociada a cada uno de los segmentos. • **SimulationMetadata.msg**: información de interés asociada a la simulación. • **SimulationMsg.msg:** información completa de una simulación a bajo nivel.
description	Ficheros descriptivos del robot y de la configuración de rviz. Si deseamos usar una nueva configuración robótica, estos ficheros deben ser modificados.	• **rviz_amcl.launch**: inicio del nodo rviz con la configuración requerida para el proceso de navegación. • **rviz_gmapping.launch**: inicio del nodo rviz con la configuración requerida para el proceso de mapeo. • **vehicle.gazebo**: configuración del robot asociada a gazebo. • **vehicle.xacro**: configuración del robot. Incluye aspecto y físicas.
gazebo_simulation	Ficheros de configuración para la simulación en gazebo.	• **world.launch**: ejecución del fichero **empty_world.launch** (disponible en el paquete gazebo_ros). Éste iniciará una instancia del simulador **gazebo** con las características del entorno proporcionado, en el cual será situado el robot mediante el nodo **spawn**.
navigation	Ficheros de configuración necesarios para el funcionamiento de la pila de navegación de ROS. Los parámetros deben ser ajustados de manera manual desde un editor de texto.	• **Ficheros yaml** cuyo contenido comprende los diferentes parámetros de la pila de navegación. La descripción individual es recogida en la tabla…. • **amcl.launch**: inicio de los nodos **amcl** (localización), **tf** (transformaciones), y las pilas **rosservice** (servicios de ROS) y **move_base** (planificadores). Estos elementos son fundamentales para establecer un proceso de navegación autónoma. • **gmapping.launch**: ejecución del nodo **gmapping**, necesario para la tarea de mapeo. • **vehicle_teleop.launch**: inicia el nodo **turtlebot_teleop**, requerido para desplazar el robot en el proceso de mapeo.
path_utilities	Contiene los scripts de python mencionados en la *Tabla 3*.	• **save_path.launch**: inicia el nodo **save_path**, implementado con el fin de construir nuestras trayectorias. • **load_path.launch**: inicia el nodo **load_path**, implementado con el fin de ejecutar nuestras simulaciones.

Tabla 1: Descripción del contenido del repositorio construido. Recuperado de fuente propia.

Funcionamiento de ROS

El núcleo de funcionamiento de ROS reside en una instancia **roscore**, la cual gestiona y establece la comunicación entre los distintos nodos en ejecución, actuando, así como bróker en el sistema. Un proceso roscore dispone tres submódulos cuando se lanza por primera:

- **Máster** de ROS: provee un servicio para el registro e identificación de los nodos del sistema.
- **Servidor de parámetros** de ROS: se trata de un diccionario compartido y multivariado al que se puede acceder a través de una API de red. Todos los nodos están en la disposición de utilizar este servidor para almacenar y consultar parámetros en tiempo de ejecución.
- Nodo **rosout**: La salida por defecto equivalente de ROS a stdout/stderr.

La comunicación entre nodos es establecida por medio buses denominados **topics**, que intercambian **mensajes**. Si un nodo está interesado en un dato en concreto, este debe subscribirse (**subscriber**) al topic que lo proporcione con el fin de obtener su valor. Si, por el contrario, nuestra intensión es publicar información a través de un nodo en un topic, este deberá ser definido como **publisher** para ese topic. En definitiva, ROS nos dota de la capacidad de crear una comunicación muchos a muchos, comúnmente usada en sistemas ORB (Object Request Broker) entre múltiples nodos.

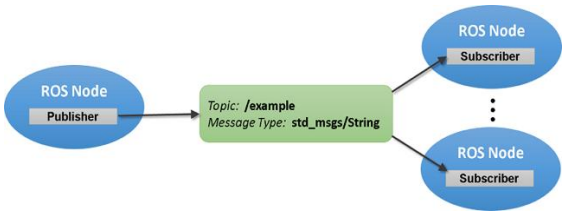

Ilustración 3: Ejemplo de comunicación básica en ROS. Recuperado de *https://www.mathworks.com/help/robotics/examples/exchange-data-with-ros-publishers-and-subscribers.html*

A un nivel inferior de abstracción del funcionamiento, el mecanismo elemental de comunicación entre nodos es el **servicio**. Este tipo de estructura encomienda la conexión entre nodos, utilizando dos modelos de mensajes, uno para la petición de la información, y otro, la respuesta dada por el otro nodo a esa petición. En el caso que se precise la necesidad de almacenar los datos publicados por algún nodo, podemos recurrir al tipo de formato **bag**. Los clientes de los lenguajes Python (**rospy**) y C++ (**roscpp**) ya ofrecen las funcionalidades básicas para gestionar este

tipo de ficheros, facilitando el almacenamiento de datos, como, por ejemplo, los datos obtenidos desde un sensor, necesarios para desarrollar y probar algoritmos.

Hay que destacar que, la distribución de ROS utilizada es concretamente **ROS Kinetic Kame**. Por lo tanto, una vez conocemos el funcionamiento del framework de robótica, debemos recurrir a algún tipo de mecanismo que nos permita modelar y representar el comportamiento físico y aspecto visual de los distintos componentes de nuestro entorno, permitiendo generar los datos aproximados de la simulación de nuestro sistema real. Para ello nos serviremos del extendido simulador **Gazebo**.

2.1.1.2 Gazebo

Gazebo [12] es un programa **open source** distribuido bajo la licencia Apache 2.0 que lleva un largo tiempo utilizándose en el ámbito de la investigación comprendida en las ramas de la robótica y la Inteligencia Artificial, convirtiéndose en uno de los mejores **simuladores** de robots de la actualidad.

La motivación principal del uso de gazebo reside en que el campo de la robótica es caro, requiere de una elevada inversión en todos los aspectos del desarrollo. Sin embargo, gracias a simuladores de estas características, tenemos la capacidad de diseñar nuestros sistemas sin previas subvenciones o sobrecostes a la hora de poner en práctica nuestros proyectos.

Por medio de este software, podremos diseñar o utilizar sensores, robots y entornos creados por la comunidad, dado que ROS ofrece la integración en **tiempo real** con Gazebo. Algunas de las **ventajas** que ofrece Gazebo son las siguiente:

- Múltiples **motores de físicas**. (Open Dynamics Engine, Bullet, etc)
- **Motor de renderizado** avanzado.
- Soporte para **plugins**.
- Programación y simulación en la nube mediante **CloudSim** y su ejecución en Amazon AWS y GzWeb.
- Soporte para herramientas de **línea de comando**.
- Librería con una amplia gama de **robots comerciales**, **sensores** y **cámaras**.

Para la puesta en funcionamiento de Gazebo, el procedimiento comprende la ejecución del nodo **gazebo_ros** desde el fichero **world.launch**, incluyendo el modelo del entorno y el robot como parámetros de entrada.

Modelos en Gazebo

En el marco de este proyecto, trabajaremos con el modelaje 3D de los siguientes componentes en Gazebo, concretamente en su versión 7.0:

- Elementos propios del robot: ruedas, motor, cuerpo robótico base, capas protectoras, sensores, etc. Para una configuración apropiada, es conveniente establecer el modo en que interactúan y se enlazan (**links**) las distintas piezas mediante las siguientes propiedades:

    - **Inercia** [16]: Propiedad física, independiente a cada componente, que describe la resistencia ofrecida a modificar su estado de movimiento. Es representada por la matriz de inercia, distinta para cada forma geométrica.
    - Aspecto o configuración **visual**. (Geometría)
    - Elemento de **colisión**: define la forma del componente, permitiendo así al motor de físicas gestionar colisiones con el resto de los objetos. Es definido de la misma manera que el aspecto visual.
    - **Materiales**: propiedades como el color o la textura.
    - **Pose** de cada elemento con respecto al elemento padre. Puede ser determinado mediante coordenadas 3D, (x, y, z), o un cuaternión que proporcione una dimensión extra que indique información acerca de la orientación de la pieza en el plano.

- Mundo o **world**: Representa el entorno con el que interactuará nuestro robot. En particular, usaremos un modelo básico en él dispondremos los obstáculos que creamos pertinentes.

Cada uno de estos componentes serán proyectados de manera individual a partir de figuras básicas. Gazebo, internamente, procesa los modelos desde ficheros con formato SDF, URDF o, para una mayor comodidad, **XACRO**, un lenguaje de macros XML. En este trabajo se hará uso de este último, no obstante, si se requiere un diseño más preciso podemos utilizar herramientas como Blender [17] o SketchUp [18].

Modelo Robótico

En robótica, cada componente de la estructura, por muy pequeño que sea, es fundamental. Un buen modelaje de nuestro robot evitará grandes quebraderos de cabeza a la hora de configurar nuestros algoritmos.

- Normalmente, con una configuración diferencial de dos ruedas, no conseguiremos un **equilibrio** que propicie una buena navegación del robot. Con este fin, podemos ubicar alguna rueda guía que proporcione la estabilidad suficiente para que los comandos de velocidades destinados al motor diferencial favorezcan el movimiento deseado.
- No siempre, las ruedas expuestas al motor diferencial están sujetas a la misma **tracción**, esto puede ocasionar que el robot no pueda ejecutar trayectorias en línea recta. Este problema debe ser solucionador con un sistema de control dinámico que varía los comandos de velocidad enviados a cada rueda, facilitando la persistencia del robot en la trayectoria.

Una distinción que debemos tener en consideración reside en las restricciones holonómicas. Un robot es **holonómico** si es capaz de cambiar su dirección instantáneamente, es decir, sobre su misma posición sin necesidad de rotar previamente. Por lo tanto, decimos que un robot diferencial que se desplace en el plano 2D cuenta con 3 **grados de libertad** (número de magnitudes que pueden variar de manera independiente), 2 establecen su posición en los dos ejes de coordenadas del plano bidimensional y otro su orientación.

Los modelos de robots diferenciales son determinantes para evitar obstáculos mediante los algoritmos de planificación local y mapeo. Habitualmente, se suele fijar una representación interna o **modelo de huella** que añade un margen de error, en función de su forma. Los más comunes son descritos a continuación:

- **Modelo de punto**: Se trata de la representación más eficiente y es determinada por una forma circular (sobre una estructura circular o cuadrada), cuyo centro es delimitado de manera arbitraría, convenientemente en el del eje del motor diferencial. El radio es definido mediante una distancia mínima configurable.
- **Modelo de línea**: Se trata de la representación ideal para robots dónde su anchura y longitud difieren. Es determinada por una línea, cuyo inicio y fin es fijado de manera arbitraría, convenientemente en el centro de cada par de ejes, y cuyo radio es definido mediante una distancia mínima configurable que parte desde el punto final.

- **Modelo de dos círculos**: Se trata de la representación ideal para robots. Su forma es similar a la de un cono o triangular. Ésta es determinada por dos círculos, cuyo inicio y fin es definido de manera arbitraría, convenientemente en el centro de cada par de ejes, y cuyos radios son establecidos mediante una distancia mínima configurable.
- **Modelo de polígono**: Es el modelo más general, definido por un polígono, dónde sus vértices son descritos de manera arbitraria, así como la distancia mínima a los límites de ese vértice.

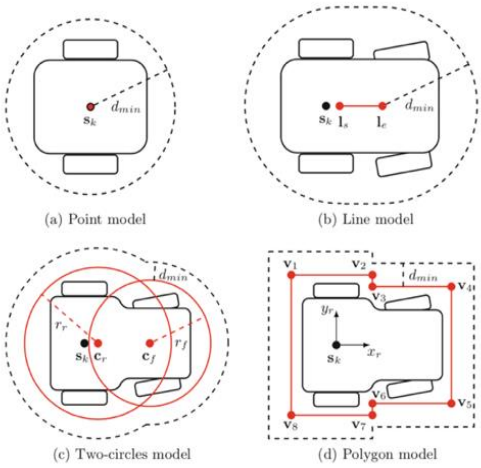

Ilustración 4: Modelos de huella comunes en un robot de eje diferencial. Recuperado de: Robot Operating System. The Complete Reference (Volume 2), año: 2017, autor: Anis Koubaa. [19]

Con el fin de simular nuestro caso de uso, hemos diseñado un modelo de robot de punto. A pesar de no proporcionar una semejanza total a las características de una silla de ruedas convencional, permitirá tomar datos del comportamiento del vehículo, que serán suministrados al alto nivel (No obstante, se han respetado las medidas estándar de una silla de ruedas en las dimensiones de largo y anchura). El motivo principal que nos ha llevado a escoger esta opción es su sencillez. Como hemos introducido, el uso de modelos sencillos contribuye a simplificar el resto de las tareas. Las características físicas del prototipo, descritas sobre el fichero **vehicle.xacro** (*Tabla 1*) son las siguiente:

- Dos ruedas cilíndricas, separadas por 86 cm y de 10 cm de radio cada una, unidas a un motor diferencial, definido mediante el plugin **differential_drive_controller** *[20]* de gazebo en el fichero **vehicle.gazebo**. (*Tabla 1*)

- Dos ruedas guías esféricas, una frontal y otra trasera, de 5 cm de radio cada una. Estos dos elementos no irán unidas a ningún motor, sin embargo, aportarán equilibrio a la estructura.

- Una base rectangular con unas dimensiones de 120 cm x 76 cm x 10 cm (largo x ancho x alto).

- Un sensor láser hokuyo, definido mediante el plugin **gazebo_ros_head_hokuyo_controller** *[20]*, en el fichero **vehicle.gazebo**. (*Tabla 1*)

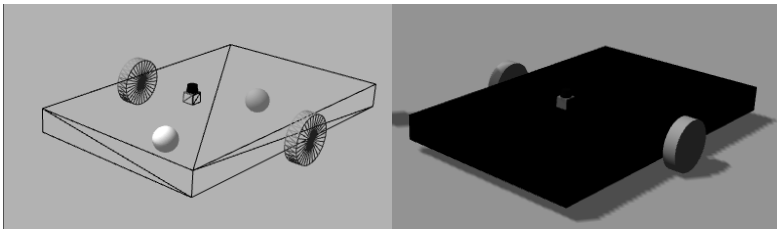

Ilustración 5: Modelo de robot diferencial utilizado. Recuperado de fuente propia.

Sensores

Los sensores son el medio principal de percepción del entorno de un robot. Estos permitirán establecer la localización del robot en un mapa y detectar obstáculos en la fase de navegación. Existen distintos tipos de sensores basados en diferentes principios físicos. No obstante, lo más común es usarlos en conjunto para añadir redundancia, de tal manera que, podamos reducir el ruido generado, por los sensores o factores externos, al máximo. En la práctica, los sensores más utilizados suelen ser los láseres o LIDAR [21], cámaras, GPS [22] y sensores de ultrasonido [23].

Como hemos introducido en el capítulo *Modelo Robótico,* nuestro robot irá equipado únicamente por un sensor láser **Hokuyo**. Perfecto para nuestro caso de uso, ya que se trata de un modelo destinado a entornos de interiores. En el mercado disponemos de varios modelos Hokuyo. Si bien, se trata de una simulación, y por suerte, Gazebo ofrece algunos plugins para los sensores más relevantes del mercado. Estos plugins parten de una configuración base para nuestros sensores, la cual es adaptable. Por lo tanto, podemos definir las características que más nos convengan. Además, debemos definir, en los ficheros de configuración del sensor (**vehicle.gazebo**) y en los correspondientes a la construcción de mapas de costes (**costmap_common.yaml**), sobre que **topic** se publicará los datos obtenidos desde el sensor.

- Rango de visión de 360°.

- Precisión de ±30 milímetros para distancias inferiores a 10 metros. La precisión indica la diferencia entre la distancia medida y la esperada. En una situación real, puede verse afectada por la temperatura, la reflectancia del objetivo o la luz ambiental.
- Mide distancias en el plano 2D, suficiente para nuestras necesidades. No obstante, en comparación con otros sensores láser, Hokuyo ofrece un límite de profundidad bastante reducido.

Creación de obstáculos

Una vez hemos concretado los modelos de nuestro robot y el mundo, debemos definir algún mecanismo que recree de manera automatizada situaciones reales a las que se podría enfrentar el robot. La solución implementada consiste en la disposición de modelos de obstáculos básicos de manera **estática**. El hecho de que estos obstáculos sean estáticos reside en que, por lo general, los algoritmos de planificación proporcionados por la pila de navegación estándar de ROS, no suelen dar buenos resultados a la hora de evitar obstáculos con comportamiento dinámico.

Además, para establecer una relación con el alto nivel, fraccionaremos la trayectoria en segmentos o tramos, pues es muy común que el elemento principal tenga que realizar varios desplazamientos para efectuar una o varias actividades. Cada uno de estos segmentos será definido por un punto de partida u objetivo anterior y un punto final o nuevo objetivo. Así mismo, con el fin de establecer el procedimiento de colocación de los obstáculos, uno de los parámetros de entrada a la ejecución de la simulación será la **distancia entre obstáculos**. Por lo tanto, la cantidad de obstáculos que serán dispuestos por segmento será calculada en función de la longitud de cada uno. El algoritmo diseñado con este pretexto, y disponible bajo el script **obstacles_util.py** (Tabla 1), es el descrito a continuación:

- Parámetros:

  n: Número de obstáculos para un segmento.

  l: Tamaño del lado para un modelo de obstáculo cuadrado.

  L: Longitud del segmento.

  d: Distancia entre obstáculos.

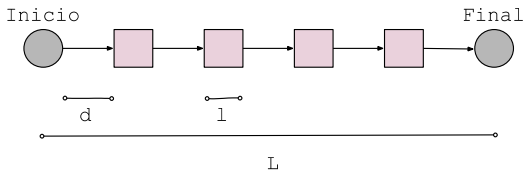

Ilustración 6: Representación visual de los distintos parámetros a tener en cuenta para la distribución de obstáculos de un segmento. Recuperado de fuente propia.

- Cálculos:
  - En primer lugar, procedemos a medir el número de obstáculos correspondientes por segmento, teniendo en cuenta que la distancia entre cada uno de ellos debe ser idéntica. De igual forma, esa distancia debe mantenerse entre los puntos de inicio y final con respecto al obstáculo más próximo. En cuanto a la longitud del lado de los obstáculos, será un valor constante para todas las simulaciones, y en nuestro caso, hemos optado por obstáculos cuadrados con lado igual a 0.5 metros de longitud.

$$L = d * (n + 1) + l * n \rightarrow n = \frac{(L - d)}{(l + d)}$$

  - Conociendo el número de obstáculos posibles, el punto inicial y final, estamos en la disposición de calcular una distribución uniforme a lo largo de la trayectoria del segmento. Para ello, hemos utilizado la funcionalidad npts, de la clase Geod, perteneciente a la librería de Python **pyproj** [24] (Pyproj está orientada a cálculos para el campo de la geodesia). Este método proporciona una lista de coordenadas a partir de una longitud/latitud final e inicial y el número de puntos deseados. Finalmente, este vector de puntos será utilizado para disponer nuestros obstáculos en las coordenadas correspondientes.
  - Sin embargo, debemos examinar que ningún obstáculo sea situado sobre el robot y el punto final. De tal forma que, el procedimiento consistirá en eliminar cualquier punto que sea ubicado a una distancia arbitraria (Convenientemente, el radio del robot) desde el punto inicial o final.
  - Finalmente se desplazará cada obstáculo de manera aleatoria, sumando o restando un valor a las coordenadas x e y establecido por el intervalo, en metros, comprendido entre 0 y un valor arbitrario constante. Para las simulaciones llevadas a cabo, el intervalo establecido oscila entre los valores [0, 0.1].

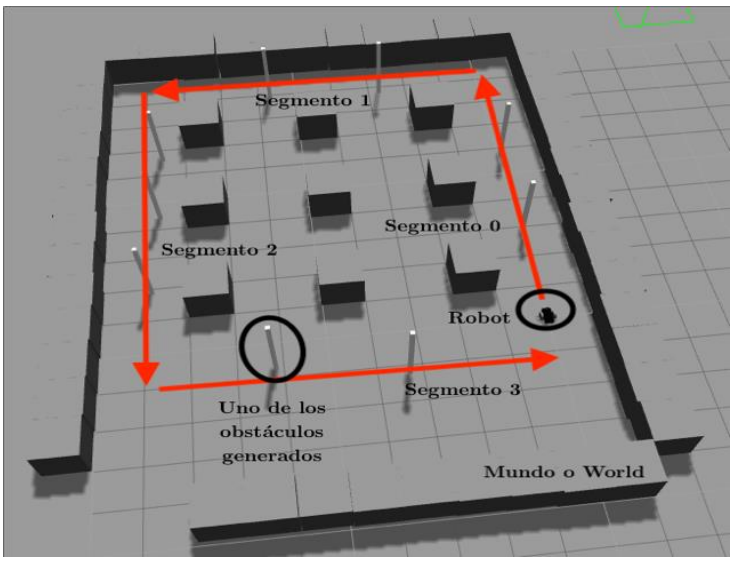

Ilustración 7: Captura de una simulación en Gazebo para una disposición de obstáculos con 3 metros de separación. Recuperado de fuente propia.

2.1.1.3 Rviz

Rviz es un **visualizador** 3D muy útil a la hora de obtener retroalimentación acerca del comportamiento de nuestros robots. Este software no es más que una pila proveída como paquete por defecto en las distintas distribuciones de ROS. Algunas de las funcionalidades de interés son las siguientes:

- Permite detectar los posibles errores que se hayan ocasionado en sistema de visión. Para ello, disponemos de un menú en cuál podemos añadir distintos elementos de manera simultánea, como, por ejemplo, la trayectoria que debería seguir el robot definida por el planificador o la nube de puntos descrita por el algoritmo de localización, entre otros.
- Proporciona un modo interactivo con el fin de ajustar ciertos parámetros directamente desde el apartado gráfico del entorno, como, por ejemplo, la posición del robo o la posición de la cámara.
- Todos los datos son capturados en tiempo real mediante la comunicación con los nodos en ejecución establecidos en los procesos de navegación.

En cuanto a la ejecución de Rviz, dispondremos de dos ficheros launch, **rviz_amcl.launch** y **rviz_gmapping.launch** (Tabla 1). Cada uno de ellos activará la pila de **rviz** con la finalidad de disponer la configuración de visualización para el proceso de navegación y mapeo respectivamente, a partir de ficheros de configuración rviz.

Creación de planes de navegación

Para definir los **planes** que debe ejecutar el robot de manera autónoma hemos utilizado Rviz. Una de las características interesante del visualizador es la capacidad de interactuar, de manera limitada, con los distintos elementos visuales de la simulación, como, por ejemplo, establecer objetivos de navegación, medir distancias, etc. Por ello, nos hemos beneficiado de estas funcionalidades con el fin de generar nuestros ficheros de planificación.

Los ficheros de navegación contendrán la posición inicial del robot, establecida previamente desplazando el robot desde Gazebo, y los objetivos marcados. El mecanismo ha sido implementado mediante el cliente de ROS para Python, **rospy**, y trabaja sobre el nodo **save_path**, cuyo funcionamiento es el siguiente:

- Desplazamos el robot en Gazebo para establecer la posición inicial del robot.
- Una vez definida la pose inicial del robot, volvemos al visualizador Rviz y presionamos el botón 2D Pose Estimate. De este modo, Rviz comenzará a capturar los objetivos que deseemos seleccionándolos en el mapa con el puntero. Debemos puntualizar que, para que el plan pueda ser ejecutado de manera satisfactoria, los objetivos deben ser fijados en zonas accesibles.
- Tras asentar los objetivos en el mapa, debemos enviar un mensaje vacío al topic **path_ready**, al cual se encuentra subscrito el nodo **save_path.** Finalmente, el mensaje es procesado y la información recogida es almacenada en un fichero **bag**.

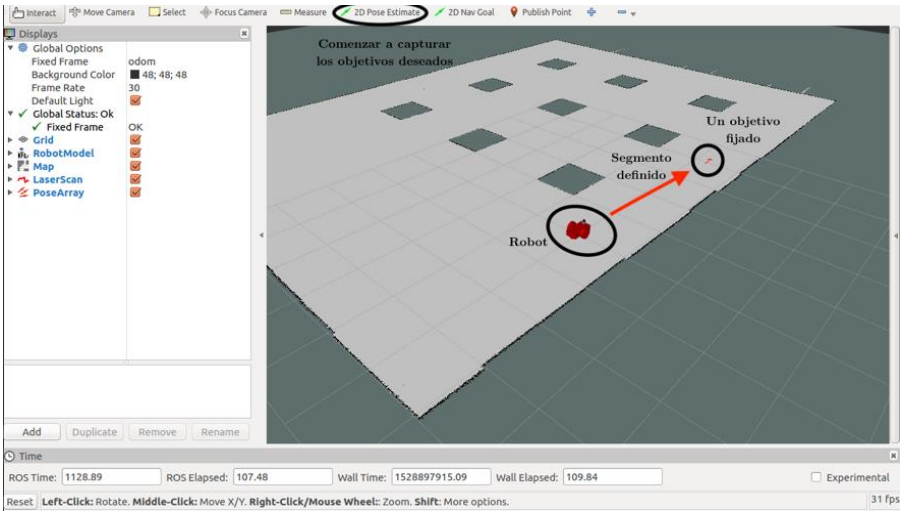

Ilustración 8: Captura de una definición de plan sobre Rviz. Recuperado de fuente propia.

Una vez ejecutemos el nodo de simulación **load_path**, el fichero bag debe ser incluido como parámetro de entrada. El flujo de ejecución de un plan es el siguiente,

(Teniendo en cuenta que los objetivos son enviados a la pila de navegación mediante la pila **actionlib** [25]):

Tantas veces como número de simulaciones seleccionadas:

Disponer obstáculos en el mundo.

Disponer el robot en la posición inicial indicada por el fichero del plan.

Para cada segmento seleccionado en el plan y mientras no tenga lugar un plan fallido (colisión, objetivo inalcanzable o tiempo de simulación límite superado):

Notificar de un nuevo objetivo a la pila de navegación.

Comenzar a capturar resultados para el segmento actual.

Esperar a una notificación de objetivo alcanzado.

Finalizar de capturar datos para el segmento actual.

Eliminar obstáculos y reiniciar el estado del mundo de gazebo a la inicial.

Calcular los resultados globales de la simulación.

2.1.1.4 Pila de navegación de ROS

Toda navegación de un robot tiene como finalidad ejecutar una ruta desde un punto A hacia un punto B de manera satisfactoria. Si bien, el problema es mucho más complejo cuando queremos que esta tarea se lleve a cabo de manera autónoma sin necesidad de estar constantemente supervisando el robot. En la actualidad, existen múltiples aproximaciones que han conseguido buenos resultados, sin embargo, aún cabe la posibilidad de mejorar el rendimiento y el comportamiento de los vehículos que cumplen con estas características, sobre todo en casos dónde el margen de error exigido es mínimo.

Para resolver esta cuestión, necesitamos conocer cierta información:

- **Localización**: El robot debe saber en todo momento donde se encuentra.
- **Mapa**: El robot debe ser capaz de construir una representación de su entorno con el fin de establecer un valor preciso de su posición y continuar avanzando.
- **Planificación global de rutas**: El robot debe ser capaz de calcular el mejor camino para llegar a su destino.
- **Planificación local de rutas**: El robot debe ser capaz de seguir la ruta calculada evitando obstáculos.

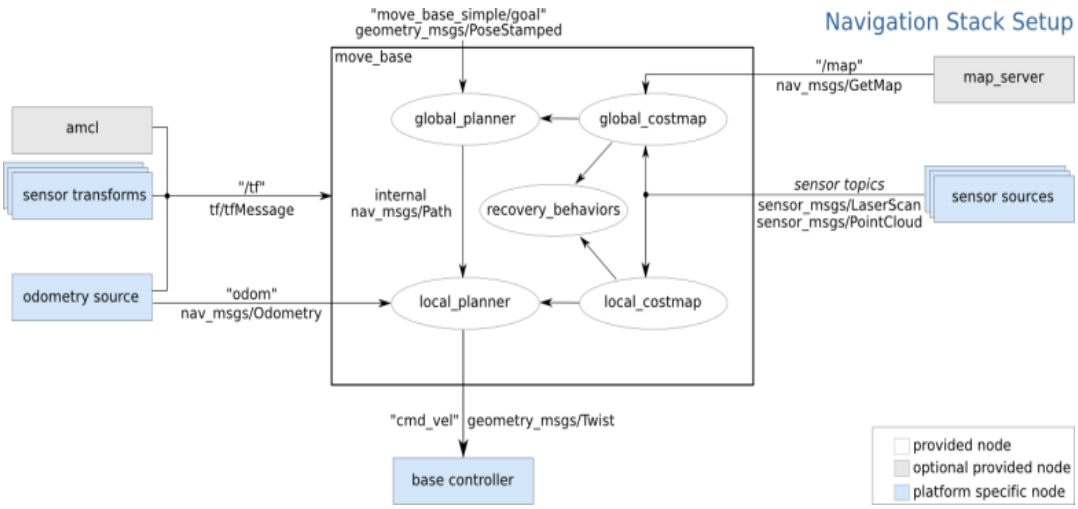

Ilustración 9: Organización de los distintos nodos de la pila de navegación de ROS. Recuperado de http://wiki.ros.org/navigation/Tutorials/RobotSetup.

En este capítulo revisaremos estos conceptos y cual ha sido el procedimiento utilizado para solventar cada uno de estos subproblemas mediante **la pila de navegación de ROS** [29], la cual contiene la implementación de algunos de los algoritmos más extendidos en el ámbito de la navegació autónoma. Estos algoritmos son configurados por medio ficheros de **yaml**. La ubicación de los ficheros yaml será proporcionada a la pila **move_base**, así como a los nodos **amcl** y **gmapping**, a través de los ficheros launch (Tabla 1). El nodo move_base provee la implementación de gran parte de las funcionalidades de la pila de navegación, como, por ejemplo, los algoritmos de planificación de rutas. Los ficheros yaml empleados son los siguientes:

Fichero	**Descripción**
amcl.yaml	Parámetros de configuración del algoritmo de localización **AMCL**.
teb_local_planner.yaml	Parámetros de configuración del algoritmo de planificación local **TEB**.
dwa_local_planner.yaml	Parámetros de configuración del algoritmo de planificación local **DWA**.
global_planner.yaml	Parámetros de configuración del método de planificación global **GlobalPlanner**. Éste permite escoger entre el algoritmo de rutas **Dijkstra** o **A***.
costmap_common.yaml	Parámetros de configuración comunes para el **mapa local** y **global de costes**, como, por ejemplo, el radio de inflación.
global_costmap.yaml	Parámetros de configuración del mapa **global de costes**.
local_costmap.yaml	Parámetros de configuración del mapa **local de costes**.
move_base.yaml	Parámetros de configuración globales a la pila **move_base**, como, por ejemplo, los planificadores utilizados, la frecuencia de computo de éstos, etc.

Tabla 2: Ficheros yaml empleados para la configuración de la pila de navegación de ROS. Recuperado de fuente propia.

Árbol de transformaciones

El árbol de transformaciones es uno de los aspectos más importantes en el modelaje y el cálculo de las cinemáticas de un robot. En un árbol de transformaciones se recogen todas las relaciones entre los distintos componentes la

simulación, tanto del entorno, como los elementos individuales del robot (ruedas, sensores, base, etc). Por ejemplo, transformar el movimiento del vehículo mediante el marco de referencia local con respecto a la referencia global (nuestro mundo) permite obtener la pose de nuestro robot respecto al sistema de coordenadas de nuestro mapa, necesario para el funcionamiento correcto de nuestros planificadores de rutas. Este proceso es utilizado también por en el módulo de **odometría** e internamente ejecuta un proceso de cómputo matricial basado en la ejecución de transformaciones en cascada entre los sistemas de coordenadas de los distintos componentes. El resultado final es un árbol de transformaciones obtenido desde el nodo **tf** de la pila de navegación. Es muy importante establecer una configuración correcta del árbol de transformaciones, de lo contrario, el comportamiento de nuestro robot no corresponderá al deseado. En el ámbito de nuestro proyecto, el inicio del nodo tf tiene lugar a partir del fichero launch **amcl.launch**.

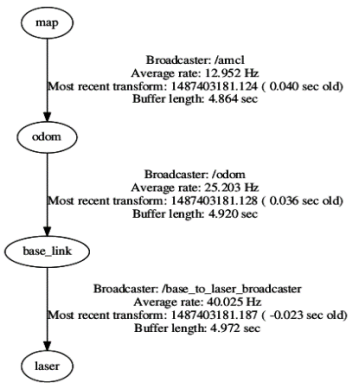

Ilustración 10: Ejemplo de árbol de transformación que tiene como marco de referencia el mapa. El orden de las transformaciones sigue un sentido descendente. Recuperado de https://answers.ros.org/question/255102/tf2-warning-when-setting-a-goal/ .

Localización

Necesitamos dar con algún procedimiento que obtenga como salida la posición o **pose** del robot en cada momento. Pero, primero tenemos que establecer una representación de esa posición en el espacio 3D o 2D. En esta iteración del proyecto, desarrollaremos la simulación bajo una localización 2D y por convenio, utilizaremos un vector de la forma (x, y, θ), dónde x e y representan la posición en el plano cartesiano y θ la orientación del robot.

Estos valores son dados con respecto a un marco de referencia al que llamaremos **sistemas de coordenadas del mundo**, el cual podremos fijar en cualquier localización del mapa. Por lo tanto, nuestro objetivo es calcular los valores del vector

(x_o, y_o, θ_o) en un instante cualquiera, para posteriormente calcular nuevos valores (x_t, y_t, θ_t) en un instante t siguiente, siempre respecto al marco global que hemos determinado. Estos cálculos vienen definidos por lo que se denomina, el **modelo de movimiento**.

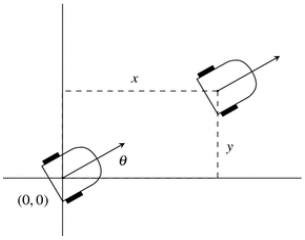

Ilustración 11: Variación de la posición en un plano 2D. Recuperado de https://link.springer.com/chapter/10.1007/978-3-319-62533-1_5

Normalmente, con el fin de establecer su posición, un robot suele tener acceso a los siguientes componentes:

- **Sensores**: Los sensores son la fuente de información principal de un robot. Estos proporcionan la información necesaria para percibir el entorno y definir la posición del robot en el mapa. No obstante, siempre que trabajemos con sensores estamos sujetos a un grado de incertidumbre, ya que, por lo general, estos están sujetos a factores aleatorios (propios del sensor o el medio) que hacen que los valores obtenidos no sean del todo precisos. Por lo tanto, no podemos asegurar que todo movimiento llevado a cabo por el robot ha sido el deseado. En consecuencia, debemos establecer un **modelo de medición** que permita dar a conocer al robot el grado de aleatoriedad al que está sujeto el proceso de estimación de su localización en el mapa. Este modelo es propio de cada tipo de sensor.
- **Mapa del entorno**: El robot debe conocer un mapa del entorno. Lo ideal es que este sea obtenido por alguna técnica de **mapeo**, en nuestro caso, profundizaremos en el funcionamiento de la técnica **SLAM**, implementada en el paquete de navegación de ROS. Sin embargo, un entorno también puede ser modelado mediante alguna herramienta gráfica si se cuenta con los datos necesarios. Este mapa, finalmente debe ser proporcionado a los algoritmos correspondientes en formato **bitmap**, en el cual se especifica en una matriz el color de cada pixel dependiendo de su resolución.
- **Odometría**: La odometría proporciona una estimación de la pose del robot respecto a un sistema de coordenadas interno, cuya relación con el sistema global desconocemos. La información de odometría será suministrada, en ROS, a los algoritmos de localización sobre el topic **odom**. Además, nos permitirá calcular la **distancia recorrida** aplicando la distancia euclidea desde el punto estimado actual con respecto al anterior, acumulando así la distancia recorrida

previamente. Este procedimiento se llevará a cabo dependiendo de la frecuencia de publicación que hayamos establecido en el topic odom, pero, por lo general, el valor por defecto será suficiente para obtener una buena precisión. La distancia euclidea es descrita por:

punto inicial: (x_1, y_1)
punto final: (x_2, y_2,)
d: Distancia total recorrida con respecto a la posición previa.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

La distancia calculada queda sujeta a la incertidumbre generada por el ruido aditivo del sensor y factores externos asociados con los actuadores. Finalmente, tomando el tiempo de navegación entre los dos puntos, seremos capaces de estimar la velocidad empleada para recorrer la trayectoria, permitiéndonos construir el mensaje con los resultados correspondientes al bajo nivel. El cálculo de estas magnitudes queda definido en el script **results_util.py**.

v: Velocidad empleada.

$\mathbf{t_1}$: Tiempo del reloj de simulación tomado en el instante del comienzo.

$\mathbf{t_2}$: Tiempo del reloj de simulación tomado en el instante final.

$$v = \frac{d}{(t_2 - t_1)}$$

Este modelo de movimiento permite calcular la posición (x, y, θ) en un instante t, respecto a un instante t-1. El desplazamiento entre dos instantes de tiempo está sujeto a un factor aleatorio cuyo origen reside principalmente en los sensores del robot. Por lo tanto, el desplazamiento medido, por ejemplo, para el sistema de coordenadas x, será descrito por la distribución de probabilidad $p(x_t \mid u_t ; x_{t-1})$, la cual suele ser muy parecida a la forma descrita por una función gaussiana, donde la estimación del nuevo valor de x vendrá dado por el candidato x_i cuya probabilidad sea superior a las demás.

La situación a de la ilustración 4 describe, en el plano (x, y), una distribución muy común para cualquier robot correctamente modelado. Las situaciones b y c describen una distribución con error elevado de traslación y rotación respectivamente.

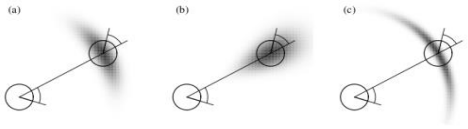

Ilustración 12: Distribuciones de probabilidad comunes en la estimación de la posición en el instante siguiente. Recuperado de: https://cuentos-cuanticos.com/2012/05/08/modelando-la-incertidumbre/#more-3057

ROS proporciona el nodo **amcl** [30], un sistema probabilístico de localización en el plano 2D que implementa una aproximación al algoritmo MCL o **Monte Carlo Localization** [31] descrito por Dieter Fox, cuyo funcionamiento se basa en un filtro de partículas que permite hacer un seguimiento del robot usando los tres componentes que hemos descrito previamente.

Mapeo

La técnica de mapeo consiste en la elaboración de una representación del entorno por el cual transitará el robot. Un **mapa** suele utilizarse con el fin de dar soporte a los procesos de localización y planificación de rutas, comprendidos en la navegación autónoma.

Los mapas generados en ROS siguen una representación denominada **Descomposición por celdas**. En ella, los objetos son representados por celdas, cuyo tamaño dependerá de la resolución de las herramientas utilizadas para la generación. Cada celda tendrá asociada un valor booleano o comprendido en el intervalo [0.0, 1.0], construyendo lo que se denomina un mapa de ocupación. El valor límite que determina si una celda está ocupada o no es establecido por un umbral arbitrario. Es la solución más extendida debido a su sencillez y la no necesidad de establecer suposiciones en cuanto a la naturaleza geométrica de los objetos.

ROS ofrece una implementación del algoritmo SLAM [32] (del inglés, Simultaneous Localization and Mapping) bajo el paquete **gmapping** [33]. Este proceso requiere de la disposición de un sensor láser en nuestro robot para definir, de manera simultánea, nuestros mapas e identificar la localización del robot en cada momento. El procedimiento requiere de algún mecanismo remoto que permita desplazar el robot a lo largo del mundo. Para ello, tras activar el nodo gmapping, utilizaremos la implementación del nodo **turtlebot_teleop**, extraído del paquete de navegación del robot **turtlebot** [34], el cual nos permitirá mover el robot mediante comandos de teclado al mismo tiempo que construimos el mapa. Una vez estemos conformes con el resultado, realizaremos una llamada al servicio **map_saver,** asociado al nodo **map_server** [35], a fin de almacenar el resultado en un fichero pgm con la descomposición por celdas del mapa y un segundo fichero yaml con algunos metadatos relacionados al mapa, como, por ejemplo, la resolución, el origen o los umbrales de ocupación. El comando en cuestión es el siguiente: rosrun map_server map_saver -f <nombre_del_mapa>

En la práctica, los sensores son imperfectos ya que sus mediciones están sujetas a un cierto grado de incertidumbre, al igual que el comportamiento de los actuadores. Por este motivo, las distintas aproximaciones del algoritmo SLAM toman la posición del

robot y los objetos como variables aleatorias, aplicando, por ejemplo, el tan extendido algoritmo Filtro de Kalman [36], útil para sistemas expuestos a ruido blanco aditivo.

En el caso de gmapping, el filtro de partículas utilizado es el filtro de **Rao-Blackwellized** [37]. Éste se fundamente en un proceso más eficiente, desarrollado mediante el muestreo en un subespacio de la distribución de probabilidad de estados y representando el resto con una estadística diferente.

Una vez construido el mapa, para hacer uso de éste en nuestra simulación, debemos lanzar el nodo map_server de ROS, que proveerá la información necesaria para que los algoritmos de planificación y localización hagan uso de él.

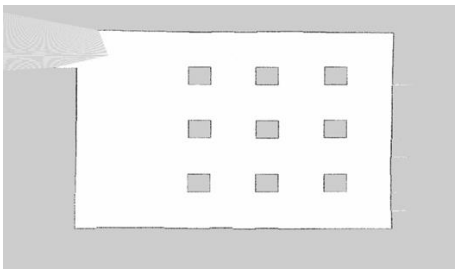

Ilustración 13: Ejemplo de mapa generado a partir del algoritmo gmapping. Recuperado de fuente propia.

Planificación global de rutas

Un planificador global debe ser capaz de generar una trayectoria libre de obstáculos entre dos puntos del mapa, de tal manera que, el robot pueda seguirla con el fin de llegar al objetivo. Los componentes necesarios para realizar esta tarea son:

- El mapa del entorno. Necesario para evitar la definición de la trayectoria sobre obstáculos.
- Las coordenadas iniciales del robot.
- Las coordenadas del objetivo final.

Con esta información el algoritmo podrá calcular la ruta óptima, en función de distintos parámetros, como puede ser aquella que minimice la distancia a recorrer entre las dos posiciones. ROS incluye implementaciones de varios algoritmos de búsqueda en grafo, pero los más extendidos en la comunidad investigadora son **A*** [38] y **Dijkstra** [39].

El empleo de Dijkstra suele ser la opción más común, en parte, gracias a su fácil implementación. Además, cuando abordamos la implementación del algoritmo A*, es necesario establecer una buena heurística si queremos obtener un buen rendimiento, lo cual hemos corroborado durante el análisis de ambos planificadores. Hemos detectado un comportamiento más eficiente a partir la implementación de ROS para

el algoritmo de Dijkstra, ya que, por lo general, ésta ofrece una trayectoria curvilínea más suave. Consecuentemente, el planificador A* es propenso a generar trayectorias muy próximas a los obstáculos, lo que puede generar circunstancias no deseadas, sobre todo en el ámbito de aplicación seleccionado.

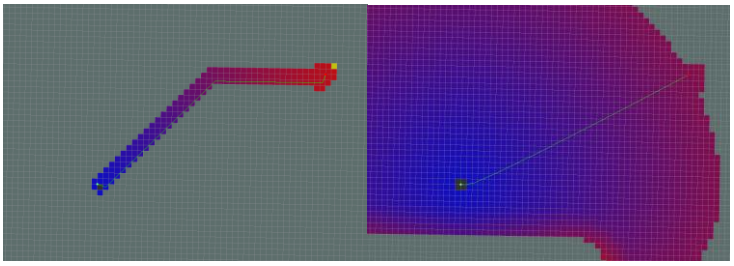

Ilustración 14: Comportamiento del planificador global A (izquierda) frente al planificador global Dijkstra (derecha). Recuperado de http://wiki.ros.org/global_planner.*

Planificación local de rutas

Una vez conocemos la trayectoria global óptima para llegar a nuestro objetivo, debemos enviar comandos de velocidad sobre el motor del robot para que este pueda moverse sobre la ruta. Ese comando de velocidad es descompuesto, mediante un conjunto de controladores, en tres componentes conocidos como velocidad de traslación (eje x), lateral (eje y) y de rotación (θ).

Sin embargo, el mundo no es estático, en el se dispondrán obstáculos de manera dinámica. Por lo que necesitamos alimentar el sistema con la información del entorno en cada momento y corregir la trayectoria si fuese necesario, evitando desviarse demasiado de la ruta preestablecida. En el caso de que esta última situación se produjese, el tiempo de simulación podría resultar excesivo. En consecuencia, debemos idear algún mecanismo que aborte la respectiva iteración de la simulación, etiquetándola como fracaso. Lo perfecto sería que el procedimiento empleado tome como entrada un factor que dependa directamente de las características del robot. Conociendo la velocidad máxima del robot, la distancia en línea recta de cada tramo y el número de obstáculos a evitar en el segmento, podemos medir el tiempo requerido para finalizar el plan del tramo en el mejor caso. Sin embargo, este hecho es hipotético, ya que, como sabemos, la navegación está sujeta a múltiples inconvenientes ocasionados por causas externas o internar al funcionamiento del robot. Con este fin, utilizaremos un **factor multiplicador del tiempo**, el cual deberemos establecer de manera experimental para cada configuración de robot empleado. El resultado será la definición de un tiempo límite de simulación para cada segmento de la planificación. A continuación, exponemos los cálculos, implementados en el fichero **obstacles_util.py**, para su resolución del problema:

C: Distancia a recorrer para evitar un obstáculo. Es fundamental tener en cuanta el **radio de inflación**, un parámetro arbitrario que permite inflar el valor de costes de los obstáculos en el **mapa de costes**, aportando un margen error con el fin de impedir colisiones en la navegación. Un mapa de costes es definido a partir de la combinación de la información tomada desde el sensor láser y el mapa de ocupación 2D o 3D. Existen dos tipos de mapas de costes, un **mapa de costes local**, donde la información se procesa de manera dinámica, dependiente rango de visión establecido para el robot y el **mapa de costes global**, el cual debe ser estático y dispone todo el mapa del entorno.

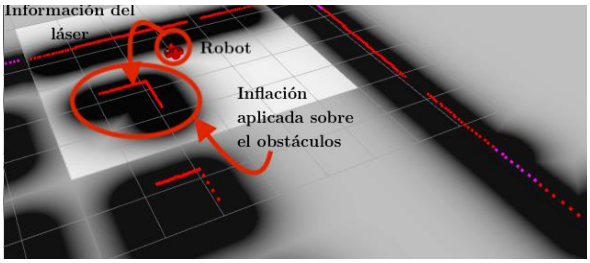

Ilustración 15: Captura de una simulación en Rviz para visualizar la información del láser y la inflación del obstáculo con un valor de 3.0 de radio. Recuperado de fuente propia.

a: Distancia total a recorrer en línea recta. Para calcular este valor partimos de la longitud total del segmento, por ello, es muy importante restar los valores de un parámetro. En concreto, es necesario sustraer el valor del radio de inflación, concepto en el que profundizaremos más adelante.

r: Radio de la circunferencia a bordear para evitar un obstáculo.

i: Radio de inflación.

b: Distancia total a recorrer para evitar todos los obstáculos.

n: Número de obstáculos para un segmento.

l: Tamaño del lado para el modelo cuadrado de obstáculo.

L: Longitud del segmento.

d: Distancia entre obstáculos.

D: Distancia total a recorrer.

T: Tiempo empleado en el mejor caso para cumplir el objetivo. Este será el valor para incrementar en función del factor temporal.

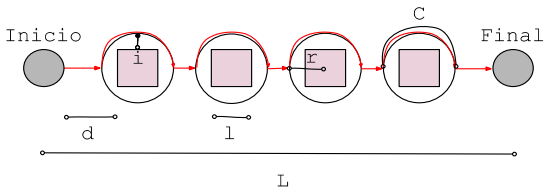

Ilustración 16: Parámetros para el cálculo de la distancia a recorrer en el mejor caso. Recuperado de fuente propia.

$$r = \frac{l}{2} + i$$

$$C = \frac{2\pi r}{2} \rightarrow \pi r$$

$$a = \big(d * (n+1)\big) - ((i*2)*n)$$

$$b = C * n$$

$$D = a + b$$

$$T = \frac{D}{v}$$

Una vez abordado este inconveniente, procedemos a la configuración de nuestros planificadores locales. Tenemos a nuestra disposición varios algoritmos que pretenden resolver este problema y, tal como hemos procedido con el planificador global, nuestra elección se ha decantado por los dos más usados y estudiados por la comunidad de ROS hoy en día, el planificador local DWA [40] y TEB [41]. El uso de uno u otro dependerá de nuestras necesidades y los recursos disponibles

2.1.2 Desarrollo

Con el fin de solventar la implementación de la simulación a bajo nivel, hemos implementado los scripts de Python descritos en la *Tabla 3*:

Script	**Descripción**
save_path.py	Captura los objetivos dispuestos por el usuario en el visualizador Rviz. Ofrece la capacidad de generar y almacenar nuestros planes para futuras simulaciones.
db_client.py	Servicio objeto de la comunicación con la base de datos a fin de insertar los resultados obtenidos en los experimentos del bajo nivel.
obstacles_util.py	Define el algoritmo de generación de obstáculos, así como la llamada a los servicios correspondientes para la ubicación de éstos en el entorno dispuesto en el simulador gazeboy el cálculo del factor temportal de navegación.
results_util.py	Captura información relativa a la simulación y la navegación del robot en el entorno con el fin de obtener información de relevancia, como, por ejemplo, la distancia recorrida, el tiempo requerido para llegar al objetivo o la velocidad media registrada.
simulation_util.py	Gestiona la ejecución de nuestros experimentos. Los scripts correspondientes a la generación de obstáculos, la captura de resultados y la comunicación con la base de datos están vinculados a este fichero. Desde esta utilidad será procesado el fichero de planificación con el propósito de enviar los objetivos al módulo de navegación de nuestro robot.
load_path.py	Encargado de la definición y ejecución del nodo load_path. La función principal del nodo load_path reside en el uso de los diferentes métodos, implementados en simulation_util.py, para la ejecución de nuestras simulaciones. Además, tras finalizar el proceso de simulación, los resultados obtenidos serán suministrados a la simulación a alto nivel y almacenados en la base de datos.
gui.py	Implementa una capa visual de interacción con el usuario. Desde ella serán accesibles todas las funcionalidades necesarias para llevar a cabo nuestras simulaciones.

Tabla 3: Scripts de python desarrollados con la finalidad de realizar nuestras simulaciones a bajo nivel. Recuperado de fuente propia.

Las relaciones entre los scripts mencionados previamente y las herramientas provistas figuran en la *Ilustración 17*. La puesta en funcionamiento de los nodos desarrollados debe realizarce por medio de los ficheros launch, dispobibles en el capítulo de *7.1 Ficheros launch*, y descritos en la Tabla 1.

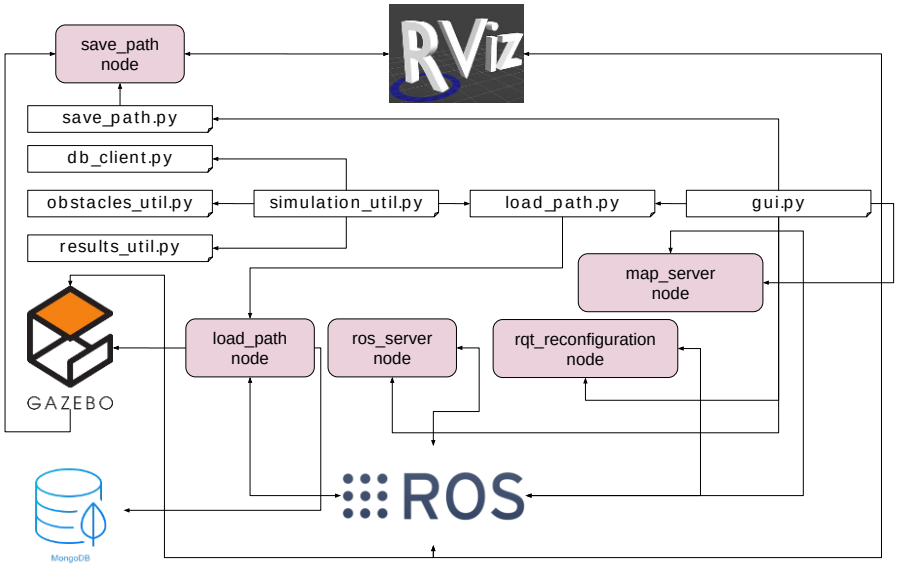

Ilustración 17: Relación entre las herramientas provistas y utilidades implementadas para el bajo nivel. Recuperado de fuente propia.

2.2 Simulación a alto nivel

2.2.1 Tecnologías utilizadas

2.2.1.1 PSIGHOS

PSIGHOS es un **simulador** de eventos discretos orientado al proceso desarrollado en **Java** por profesores y alumnos de la Universidad de La Laguna. Esta librería proporciona versatilidad y múltiples funcionalidades a la hora de llevar a cabo tareas de redistribución de recursos y reingeniería de procesos.

El método que ofrece el simulador, con la finalidad de proceder en el modelaje de nuestros sistemas, se basa en una lista de clases de Java, resumida a continuación a las necesidades de nuestro proyecto, permitiendo así configurar nuestro caso de uso descrito en la Ilustración 1:

- **Experiment**: Permite definir cuantas simulaciones queremos llevar a cabo y de que tipo. Además, podemos indicar la forma en la que visualizaremos los resultados.
- **Simulation**: En ella deberemos definir los distintos componentes de nuestra simulación.

- **ElementType**: Los elementos son las entidades principales de nuestra simulación que interactuarán con el sistema. Estos establecerán el inicio de las actividades y los recursos requeridos. Mediante esta clase, podemos agrupar todos los elementos de un mismo tipo.
- **Resource**: Los recursos no son más que algún tipo de medio material o humano que facilita la realización de las actividades planificadas. Un recurso presenta diferentes características, como su disponibilidad horaria, o la posibilidad de estar sujeto a interrupciones en algún momento en concreto del ciclo de vida de la simulación. Cada recurso se puede emplear para múltiples objetivos o roles.
- **ResourceType**: Mediante esta clase definiremos los distintos roles que puede asumir cada recurso, lo que permite limitar a que clase de actividades se puede asignar su uso.
- **Workgroup**: Un grupo de trabajo demarca la cantidad y el tipo de recursos necesarios para realizar una tarea.
- **ActivityFlow**: Las actividades son los cometidos o tareas que ejecutan los elementos. Cada actividad puede ser asignada a varios grupos de trabajo, asignando el tiempo requerido para su cumplimiento.
- **ElementGenerator**: Esta clase define la forma en la que los recursos serán generados, lo que conlleva regular el instante de creación, el tipo de elemento, así como el flujo de trabajo asociado.
- **SimulationPeriodCycle**: Facilita la delimitación de los horarios o ciclos de trabajo, fijando la activación de forma estática o variable en función de nuestras necesidades. Un ejemplo podría ser un ciclo de trabajo que se active cada día.
- **TimeFunctionFactory**: Ofrece distintos métodos para generar distribuciones temporales sujetas a una cierta probabilidad. Por ejemplo, una distribución normal o uniforme.
- **ExclusiveChoiceFlow**: Define el patrón de selección de flujo exclusivo. Permite habilitar un mecanismo de elección, mediante una condición, para bifurcar el flujo hacia una rama en concreto de entre varias. Por ejemplo, podríamos la salida hacia una actividad u otra dependiendo del tipo de recurso escogido para la realización de la simulación. Para ello, podemos definir algún método de escucha (**listener**) que compruebe dicha condición consultando los parámetros de la simulación.

La definición del modelo, para el caso de uso establecido, estará descrita bajo un fichero java, cuyo contenido comprende la clase **WheelChairsSimulation** (extiende las diferentes propiedades de la clase **Simulation)** y en la cual se recogen todas las

características de nuestro modelo. La ejecución de los experimentos queda ligada a la clase **WheelChairsExperiment** (extiende las diferentes propiedades de la clase **Experiment**) y tiene lugar desde la interfaz de usuario, implementada a alto nivel.

Más allá de las características descritas en este apartado, la librería ofrece la implementación de otros patrones de workflow, como, por ejemplo, MultipleChoice, Join o Merge. Éstos resultan de gran utilidad en el momento que nuestros modelos adquieren un cierto grado de complejidad.

2.2.2 Desarrollo

Con el fin de solventar la implementación de la simulación a alto nivel, hemos precisado de los ficheros de Java descritos en la *Tabla 4*:

Fichero	**Descripción**
ROSListener.java	Comunicación con ROS a través de websockets [13]. El método de comunicación es efectuado a través de la suscripción al topic [15] o bus de comunicación establecido en el módulo de bajo nivel.
DatabaseService.java	Acceso a la base de datos con el fin de insertar datos y consultar la información almacenada desde el módulo de bajo nivel.
HistoricalModeGUI.java	Interfaz de usuario correspondiente al modo de simulación "historial". El modo historial ejecuta los experimentos asociados a nuestro modelo a partir los datos provistos desde el bajo nivel.
MainGUI.java	Interfaz de usuario principal. En ella, el usuario proporcionará los parámetros de entrada necesarios para la ejecución de la simulación.
CompleteResults.java	Describe la estructura correspondiente a los datos obtenidos desde la simulación a bajo y alto nivel.
PSIGHOSResults.java	Describe la estructura correspondiente a los datos obtenidos desde la simulación a alto nivel.
ROSResults.java	Describe la estructura correspondiente a los datos obtenidos desde la simulación a bajo nivel.
ROSSegment.java	Describe la estructura correspondiente a los datos obtenidos, desde la simulación a bajo nivel, para un segmento individual de la trayectoria realizada por el robot.
ROSSimulationMedatada.java	Describe información de utilidad correspondiente a una simulación de bajo nivel.
WheelchairInfoListener.java	Oyente necesario para capturar la información de nuestros experimentos a alto nivel.
WheelChairsEndListener.java	Oyente necesario para capturar la finalización de nuestros experimentos a alto nivel.
WheelChairsExperiment.java	Establece el método de ejecución de nuestros experimentos.
WheelChairsSimulation.java	Describe el modelo de nuestro caso de uso. (*Ilustración 1*)

Tabla 4: Ficheros de Java implementados con el fin de solventar la ejecución de nuestras simulaciones a alto nivel. Recuperado de fuente propia.

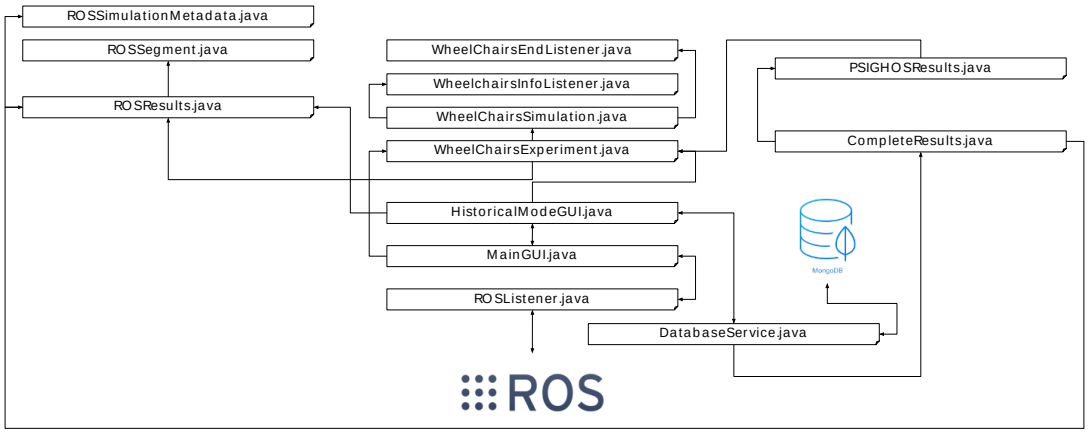

Ilustración 18: Relación entre las herramientas provistas y utilidades implementadas para el alto nivel. Recuperado de fuente propia.

2.3 Persistencia de los datos

Tras realizar nuestras simulaciones, interesa otorgar persistencia a los datos resultantes, de tal manera que, esa información pueda ser utilizada en futuros análisis del caso de uso sin la necesidad de lanzar nuevas simulaciones. Existen muchas alternativas a fin de resolver este requisito. En nuestro caso, hemos optado por el sistema de base de datos, NoSQL **y** orientado a documentos, MongoDB. Es un proyecto desarrollado bajo el concepto de código abierto y a diferencia de los sistemas de bases de datos relacionales donde la información es almacenada en tablas, MongoDB utiliza una estructura de datos similar a JSON, en concreto ésta es conocida como BSON, fundamentada en la representación binaria de las estructuras de datos, lo que aporta mayor velocidad en términos de almacenamiento y consulta a nuestras aplicaciones.

Nuestros datos serán almacenados, por medio de los clientes de MongoDB para Python y Java, en las siguientes **colecciones**. (Una colección es el equivalente a una tabla SQL convencional):

- **low_level_collection**: Resultados obtenidos en las simulaciones a bajo nivel.
- **high_level_collection**: Resultados individuales obtenidos en cada tanda de simulación a alto nivel.
- **high_level_total_results_collection**: Resultados globales obtenidos en cada tanda de simulación a alto nivel.
- **merged_collection**: Datos obtenidos a partir de operaciones de agregación entre los distintos resultados de alto nivel.

- **simulations_collection**: Metadatos correspondientes a cada configuración de simulación ejecutada a bajo nivel. De esta manera podremos establecer operaciones de agregación, estableciendo así consultadas más avanzadas.

2.4 Comunicación entre niveles de simulación

Nuestra herramienta esta compuesta principalmente por dos módulos de simulación que en primera instancia no tienen la capacidad de comunicarse entre sí. No obstante, ROS ha conglomerado una comunidad extensa de desarrolladores que se encargan de mantener y desarrollar mejoras o nuevas funcionalidades sobre el framework. Por suerte, contamos con algunos artilugios que nos facilitarán la labor:

- **rosbridge_suite** [26]: Se trata de un paquete de ROS que provee una **API JSON** para programas externos al framework, dando acceso a la subscripción o publicación en los distintos topics definidos a el fin de obtener o suministrar información entre los servicios involucrados. Hay varios front ends o capas superiores que sirven de interfaz sobre rosbridge_suite, por ejemplo, un servidor websocket para navegadores. La puesta en funcionamiento esta definida bajo el nodo **rosbridge_websockets** y, para ello, el comando establecido es el siguiente: roslaunch rosbridge_server rosbridge_websocket.launch
- **WebSockets** [13]: Es una tecnología que ofrece un canal de comunicación full-duplex y bidireccional sobre un único **socket TCP**. Está diseñada para su uso sobre navegadores y servidores web, sin embargo, puede utilizarse sobre cualquier aplicación cliente/servidor.
- **java_rosbridge** [27]: Está librería de java ofrece una interfaz sobre WebSockets que presenta funcionalidades como publicar o subscribirse a los distintos topics abiertos desde el servidor de rosbridge_server. Está implementada sobre la librería **Jetty 9** [28], orientada al desarrollo de servidores web cuyo mecanismo de comunicación ofrece soporte para múltiples protocolos, entre ellos WebSocket. Con el fin de establecer la comunicación con ROS, implementaremos un cliente en Java, haciendo uso de java_rosbridge, que permite realizar la recepción de los datos de la simulación a bajo nivel.

La **comunicación** entre los dos módulos será establecida de la siguiente manera:

1. Desplegamos el servidor ROS.
2. Desplegamos el "oyente" de Java para ROS, especificando los parámetros correspondientes de la simulación de eventos discretos.
3. Ejecutamos la simulación a bajo nivel.

4. Una vez finalizada la simulación robótica, el mensaje es enviado al oyente de Java en formato JSON.
5. Tras recibir y procesar el mensaje, la simulación a alto nivel es ejecutada.

El **mensaje** definido contiene la siguiente información (Consultar mensajes personalizados de ROS en *Tabla 1*):

- Metadatos de la simulación: ficheros de simulación, planificadores usados, etc.
- Resultados medios de la simulación: velocidades promedio, tiempos promedio, número de fracasos, etc.
- Resultados medios por segmento: velocidades promedio, tiempos promedio, número de fracasos, etc.
- Resultados individuales por iteración de simulación: velocidades, tiempos, etc.

Hay que destacar que los datos promedio son extraídos de iteraciones **exitosas** de la simulación, es decir, el vehículo ha conseguido completar el plan. De esta manera, favorecemos a la obtención de una distribución gaussiana sobre los tiempos, lo cual proporciona un estudio más preciso en el computo global de los resultados. La definición de este mensaje, así como la captura de información de la simulación, tiene lugar bajo el script de Python **results_util.py** (Tabla 3).

2.5 Interfaz de Usuario

Con el fin de conceder una buena experiencia de usuario, se han diseñado dos interfaces interactivas, una independiente para cada módulo. De esta manera, se podrá gestionar cada nivel de simulación de una manera más intuitiva y eficiente, ya que, tanto ROS como PSIGHOS actualmente no ofrecen está facilidad.

La interfaz para la simulación a bajo nivel, implementada sobre el paquete de Python **tkinter**, hará uso de **scripts** previamente definidos con el fin de consumir los ficheros launch descritos en capítulos anteriores, ofreciendo las siguientes funcionalidades:

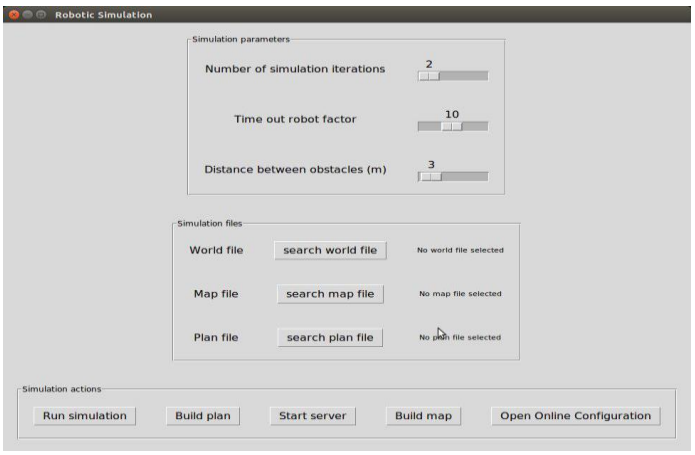

Ilustración 19: Interfaz de usuario implementada para la simulación a bajo nivel. Recuperado de fuente propia.

- **Parámetros de la simulación**: Los parámetros de simulación se encuentran dispuestos bajo la etiqueta "Simulation parameters". En esta sección podremos establecer, mediante controles deslizantes, el número de iteraciones de nuestra simulación, el factor temporal de ejecución de nuestra simulación y la distancia entre obstáculos.
- **Ficheros de la simulación**: Los ficheros de configuración de simulación se encuentran dispuestos bajo la etiqueta "Simulation files". En esta sección podremos seleccionar, mediante un buscador de archivos, el fichero con la configuración de nuestro mundo de Gazebo, el mapa del mundo y el plan a ejecutar. Se deberá seleccionar uno u otro dependiendo de las acciones que deseemos tomar.
- **Acciones de la simulación**: Las acciones disponibles para la simulación se encuentran dispuestos bajo la etiqueta "Simulation actions". En esta sección podremos seleccionar, mediante un conjunto de botones, la ejecución de la simulación (partiendo de la configuración fijada), construir un plan de rutas, ejecutar el servidor para establecer la comunicación con el módulo de alto nivel, construir el mapa del mundo seleccionado y abrir la interfaz **rqt_reconfigure**, disponible en ROS para la reconfiguración de los parámetros de la simulación, útil para realizar cambios en tiempo de ejecución. La base lógica de cada una de estas acciones ha sido esclarecida en capítulos previos, y su uso en la interfaz se realizará por medio de llamada a los scripts de bash correspondientes a cada rutina definida en los ficheros launch.

Una vez hemos generado los resultados a bajo nivel, necesitamos manipular esa información con el objetivo de ejecutar experimentos sobre nuestro caso de uso. El

modo de interacción, accesible al usuario, corresponde a una capa visual de configuración integrada por los siguientes componentes:

- **Marco principal**: En él seleccionaremos los parámetros de entrada para la ejecución de nuestros experimentos. En concreto, las variables necesarias corresponden al número de experimentos deseados a partir de la configuración proporcionada, el número de bedeles, de doctores, de sillas de ruedas autónomas y/o manuales, la cantidad de minutos entre cada llegada de pacientes, al igual que la cifra de estos, un factor diferenciador que permita variar los tiempos de rutas de las sillas de ruedas manuales respecto a las autónomas (Los tiempos utilizados corresponden a sillas de ruedas autónomas) y finalmente el periodo temporal de simulación medido en días.

  Además, nuestros experimentos pueden efectuarse a partir de datos generados directamente desde una simulación de ROS (**Modo escucha**) o sobre resultados previamente almacenados en la base de datos (**Modo historial**). Usar uno u otro dependerá de nuestras necesidades. No obstante, el modo historial resulta bastante útil a la hora de realizar operaciones de agregación.

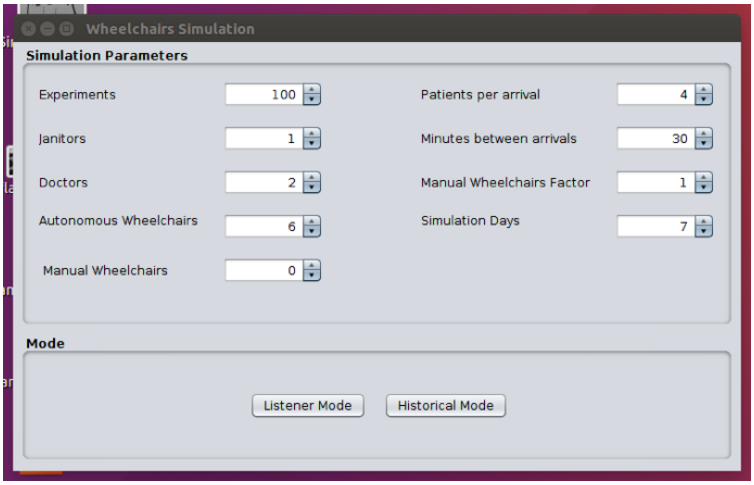

Ilustración 20: Marco principal. Recuperado de fuente propia.

En el caso que precisemos del modo historial, se desplegará en pantalla un marco secundario en el cual podremos filtrar nuestros datos de entrada de la siguiente manera:

1. Seleccionamos, en la tabla superior, una configuración de simulación previamente almacenada.

2. A continuación, en la segunda tabla, dispondremos de todas las simulaciones realizadas a bajo nivel para dicha configuración, podremos seleccionar una o varias (Operación de agregación). Es imprescindible comprobar que las instancias de simulación escogidas sean útiles. Una simulación será útil si tras su ejecución han resultado dos o más iteraciones sin fracaso previo a la llegada a un objetivo. Este indicador es imprescindible debido a que requerimos de agrupaciones de datos equivalentes a ese tamaño con el fin de obtener algunos estadísticos como la desviación típica.
3. Seleccionamos el rango de valores para el parámetro "distancia entre obstáculos" mediante el control deslizante dispuesto en la sección central de la ventana. De este modo, podremos llevar a cabo análisis de nuestras simulaciones más detallado. Además, contamos con una tabla, en la sección inferior, la cual expone de manera dinámica ciertos indicativos correspondientes a cada segmento del computo global de las instancias de simulación seleccionadas.
4. Tras llevar a cabo los pasos descritos previamente, estamos en la disposición de ejecutar nuestros experimentos o almacenar los datos agregados para un uso posterior.

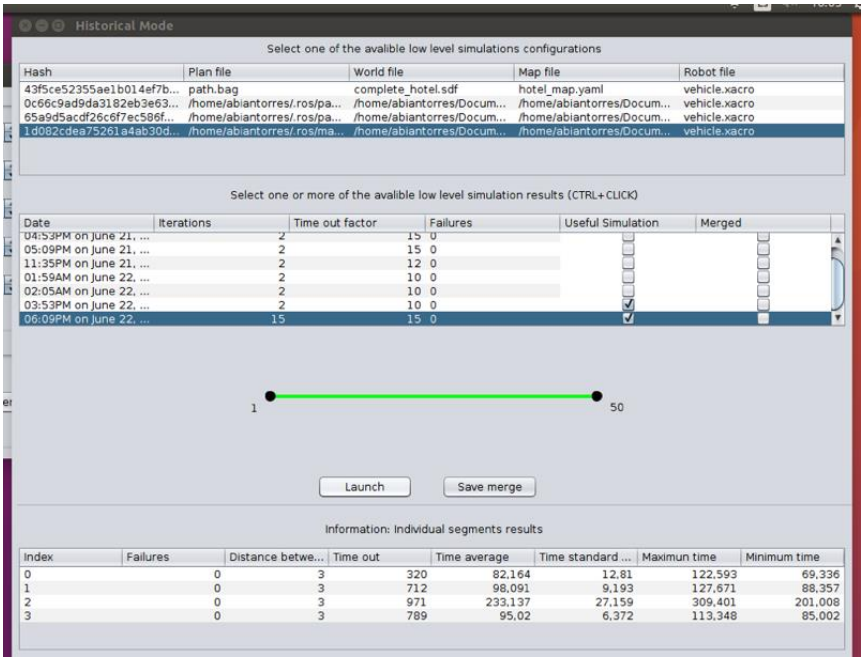

Ilustración 21: Marco secundario correspondiente al modo historial. Recuperado de fuente propia.

Tras finalizar una tanda de experimentos completa, se dispondrá, en una ventana emergente y en formato **json**, los resultados individuales de cada experimento, así como los valores en el cómputo global.

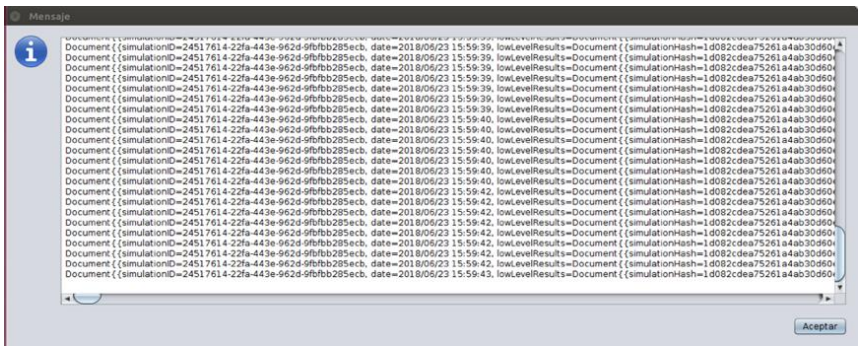

Ilustración 22: Ventana emergente en la que se disponen algunos resultados de una simulación a alto nivel. Recuperado de fuente propia.

3. Resultados

Con el fin de inferir qué planificador local de rutas es el más apropiado para poner en funcionamiento nuestro sistema, hemos elaborado un pequeño análisis comparativo entre las dos configuraciones definidas para los planificadores estudiados en este documento. El mundo empleado para este pequeño experimento ha sido diseñado mediante el editor de gazebo y no se corresponde con ninguna localización real.

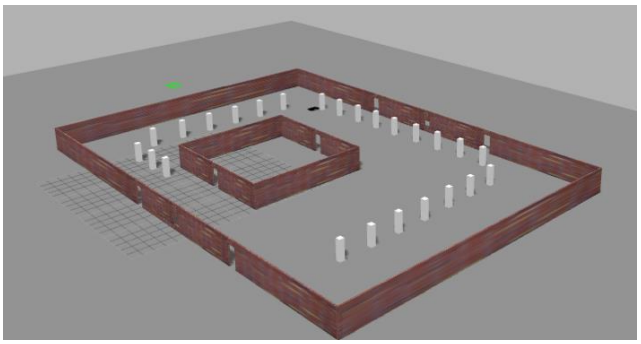

Ilustración 23: Mundo diseñado en gazebo para el proceso de experimentación. En la ilustración se disponen obstáculos con una separación de 3 metros. Recuperado de fuente propia.

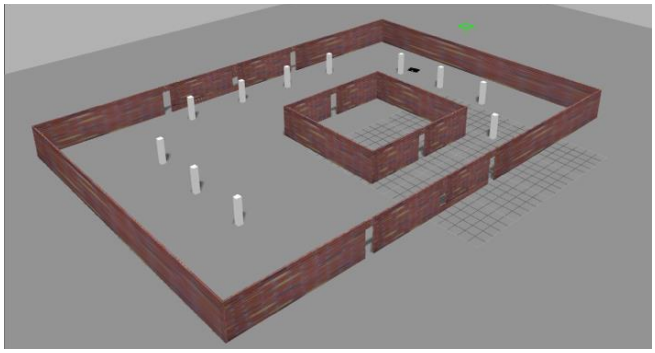

Ilustración 24: Mundo diseñado en gazebo para el proceso de experimentación. En la ilustración se disponen obstáculos con una separación de 7 metros. Recuperado de fuente propia.

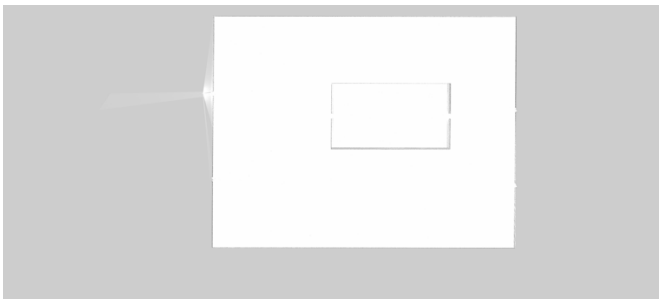

Ilustración 25: Mapa del mundo generado mediante el algoritmo gmapping. Recuperado de fuente propia.

Empleando la herramienta de automatización para simulación diseñada, hemos ejecutada cuatro tandas de experimentos, dos para cada planificador local, divididas en 21 iteraciones cada una, y sobre un plan de 4 segmentos (Ver disposición en la Ilustración 23). Los parámetros de configuración seleccionados corresponden a un valor de 8.0 para el factor temporal de simulación y dos valores de distancias entre obstáculos con el fin de observar posibles patrones diferenciadores de comportamiento entre diversas situaciones. Los resultados obtenidos quedan recogidos en la siguiente tabla:

Planificador local	DWA		TEB	
Número de fracasos	0	0	1	0
Número de simulaciones	21	21	21	21
Factor temporal de simulación	8.00	8.00	8.00	8.00
Distancia entre obstáculos (metros)	3.00	7.00	3.00	7.00
Tiempo promedio (desviación típica) (segundos)	142.09 (4.93)	140.32 (2.73)	190.85 (12.13)	201.03 (47.75)
Distancia recorrida promedio (desviación típica) (metros)	117.87 (4.52)	115.78 (4.18)	124.40 (6.30)	127.07 (5.93)
Velocidad promedio (desviación típica) (metros/segundos)	0.83 (0.05)	0.83 (0.04)	0.65 (0.05)	0.66 (0.1)
Tiempo (mínimo, máximo) (segundos)	(137.63, 156.31)	(137.38, 146.95)	(170.71, 209.26)	(157.43, 331.28)
Distancia recorrida (mínima, máxima) (metros)	(99.07, 124.79)	(97.63, 117.31)	(104.85, 130.07)	(121.28, 142.97)
Velocidad (mínima, máxima) (metros/segundos)	(0.65, 0.86)	(0.66, 0.85)	(0.53, 0.72)	(0.39, 0.77)

Tabla 5: Resultados obtenidos para los planificadores dwa y teb. Recuperado de fuente propia.

Podemos observar un comportamiento más uniforme del planificador local DWA respecto al planificador local TEB, tal y como muestra la desviación típica de las distintas magnitudes. Si bien, a pesar de la gran cantidad de posibilidades que ofrece TEB, en cuanto a flexibilidad de movimientos se refiere, lograr una configuración óptima resulta algo complejo debido, principalmente, a la gran cantidad de parámetros a ajustar en comparación a DWA (La optimización de estos algoritmos podría ser objeto de estudio en iteraciones futuras). Además, con DWA, a medida que aumentamos la distancia entre los obstáculos, los índices del tiempo, distancia y velocidad de simulación presentan una pequeña mejora, así como una menor desviación entre los resultados obtenidos. Por lo tanto, el siguiente paso será la experimentación con nuestro modelo de alto nivel a partir de los datos temporales por tramos correspondientes a la tanda con DWA y 7.0 metros de distancia entre obstáculos.

La configuración inicial de nuestra simulación a alto nivel es la siguiente (Los parámetros no se corresponden a ningún caso real, no son más que una aproximación):

Parámetro	Valor
Sillas de ruedas autónomas	6
Bedeles	1
Doctores	2
Pacientes por llegada	4
Minutos entre llegadas	30
Días de simulación	7
Experimentos	100

Tabla 6: Parámetros de entrada para el experimento del caso de uso a alto nivel. Recuperado de fuente propia.

A continuación, presentamos los resultados promedios obtenidos a partir de los 100 experimentos ejecutados. El objetivo principal que nos lleva a modelar este caso de uso podría ser la reducción de costes, a través de la minimización del tiempo invertido, sobre la tarea de asistencia por parte de los bedeles a pacientes en sillas de ruedas y la reducción de tiempos de trayecto o espera de los pacientes.

Recurso	Instancias	Tiempo empleado promedio (horas)
Silla de ruedas autónoma	1336	557.0
Doctor	1336	278.0
Bedel	2672	89.0

Tabla 7: Uso de los recursos dispuestos en la simulación a alto nivel. Recuperado de fuente propia.

Estados del paciente	Instancias
Pacientes atendidos	1336
Pacientes que han esperando por un doctor	696
Pacientes que han esperando por un bedel	1067

Tabla 8: Contabilización de los pacientes atendidos y que han tenido que esperar por un doctor o bedel. Recuperado de fuente propia.

Actividad	Tiempo empleado Promedio (minutos)	Desviación típica (minutos)
Consulta	12.49	1.46
Ocupar/abandonar la silla	3.96	0.82
Espera del bedel	5.42	4.15
Espera del doctor	3.25	3.58
Ruta	4.67	0.006

Tabla 9: Tiempos promedios para cada actividad de nuestro modelo del sistema de sillas de ruedas autónomas. Recuperado de fuente propia.

4. Conclusiones y líneas futuras

El desarrollo de este proyecto ha derivado en un conjunto de herramientas provechosas de cara a futuras iteraciones. Tras obtener los primeros resultados de nuestras simulaciones, hemos sido capaces de distinguir el comportamiento de nuestro vehículo autónomo según la lógica computacional asociada a nuestro módulo de navegación, dando pie a establecer la configuración más apropiada para nuestro caso de uso.

Asimismo, ha sido posible definir una base de intercambio entre dos niveles de simulación, lo cual abre puertas de cara a futuras líneas de trabajo sobre este prototipo de reproducción de modelos autónomos. Por ejemplo, una siguiente línea de trabajo podría establecer un mecanismo visual e interactivo de cara a la definición de nuestros modelos de eventos discretos, disponiendo así una comunicación, con un grado de complejidad mayor entre los dos niveles de simulación.

Llegados a este punto, otros componentes o elementos, como, por ejemplo, pacientes, doctores y debeles podrían figurar en nuestro modelo de simulación a bajo nivel. Este hecho aportaría un escalón más en términos de similitud con los sistemas reales, culminando con un análisis, en términos de coste-eficiencia, más preciso. No obstante, los resultados obtenidos a partir del caso de uso empleado pueden ser de gran utilidad como paso previo al estudio e implantación de estos sistemas de vehículos autónomos en un entorno hospitalario. Incluso, a partir de modificaciones puntuales, podríamos extrapolar nuestra configuración a un ámbito de aplicación diferente.

La gran variedad de posibilidades que ofrece esta herramienta extiende el rango de mejoras, por ejemplo, dentro del campo de la visión por computador y la inteligencia artificial. En pleno comienzo del siglo XXI, el avance en términos de rendimiento computacional ha generado una expansión de las distintas técnicas de inteligencia artificial, y el campo de la navegación autónoma es uno de esos ámbitos de uso en la cual ha tenido una influencia positiva, sobre todo por lo que corresponde a la percepción del entorno y obstáculos. Hoy en día, ROS ofrece la posibilidad de trabajar sobre estas líneas de desarrollo, y más allá de la lógica implementada en este proyecto, las aproximaciones más avanzadas hoy en día reman en esta dirección.

Por lo tanto, las herramientas desarrolladas y el pequeño estudio desempeñado podrían desembocar en un proyecto que permita analizar la implantación, de manera precisa, en entornos propensos para la automatización, como podrían ser aeropuertos, hospitales o muelles de carga y descarga.

5. Summary and Conclusions

The development of this project has resulted in a set of useful tools for future iterations. After obtaining the first results of our simulations, we were able to distinguish the behavior of our autonomous vehicle according to the computational logic associated with our navigation module, giving rise to establish the most appropriate configuration for our use case.

Likewise, it has been possible to define a basis of exchange between two levels of simulation, which opens doors for future lines of work on this prototype of reproduction of autonomous models. For example, a following line of work could establish a visual and interactive mechanism for the definition of our models of discrete events, thus providing communication, with a greater degree of complexity, between the two levels of simulation.

At this point, other components or elements, such as, patients, doctors and janitors, could be included in our simulation model at a low level. This fact would contribute a step more in terms of similarity with real systems, culminating with an analysis, in terms of cost-efficiency, more precise. However, the results obtained from the case of use can be very useful as a preliminary step to the study and implementation of these autonomous vehicle systems in a hospital environment. Even, from specific modifications, we could extrapolate our configuration to a different scope of application.

The wide variety of possibilities offered by this tool extends possible improvements within the field of computer vision and artificial intelligence. At the beginning of the 21st century, progress in terms of computational performance has generated an expansion of the different techniques of artificial intelligence, and the field of autonomous navigation is one of those areas of use in which it has had considerable influence, especially, in what corresponds to the perception of the environment and avoid obstacles. Today, ROS offers the possibility of work on these lines of development, and beyond the logic implemented in this project, the most advanced approaches today converge towards this direction.

Therefore, the tools developed and the small study carried out could lead to a project that allows analyzing the implementation, in a precise way, in environments prone to automation, such as airports, hospitals or loading and unloading docks.

6. Presupuesto

El presupuesto requerido para la realización de este proyecto queda registrado en la *Tabla 10*. El coste es evaluado en función del valor medio por hora asociado al contrato laboral de un ingeniero informático, equivalente a 20€.

Tarea	**Horas empleadas**	**Coste (euros)**
Gestión del conocimiento	90	1800
Desarrollo o Implementación del simulador	105	2100
Pruebas de funcionamiento	20	400
Análisis de resultados	20	400
Documentación	20	400
Reuniones de seguimiento y otras actividades	45	900
Total	300	6000

Tabla 10: Presupuesto contabilizado por tareas tras la realización del proyecto. Recuperado de fuente propia.

7. Apéndice

Todos los ficheros de implementación y documentación asociados al simulador de dos niveles están disponibles en el enlace al siguiente repositorio de github:

https://github.com/abiantorres/autonomous-vehicles-system-simulation

7.1 Ficheros launch

7.1.1 control.launch

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <rosparam file="$(find control)/config/control.yaml" command="load"/>
  <node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false" output="screen" ns="/vehicle" args="joint_state_controller
mobile_base_controller"/>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" respawn="false" output="screen">
    <param name="robot_description" command="$(find xacro)/xacro.py '$(find
description)/urdf/vehicle.xacro'"/>
    <remap from="/joint_states" to="/vehicle/joint_states" />
  </node>
</launch>
```

7.1.2 rviz_amcl.launch

```
<?xml version="1.0"?>
<launch>
  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
description)/urdf/vehicle.xacro'"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher"/>
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
description)/rviz/amcl.rviz"/>

</launch>
```

7.1.3 rviz_gmapping.launch

```
<?xml version="1.0"?>
<launch>
  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
description)/urdf/vehicle.xacro'"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher"/>
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
description)/rviz/mapping.rviz"/>
</launch>
```

7.1.4 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"/>
  <arg name="x" default="0"/>
  <arg name="y" default="0"/>
  <arg name="z" default="0"/>
  <arg name="roll" default="0"/>
  <arg name="pitch" default="0"/>
  <arg name="yaw" default="0"/>
  <arg name="world_file" default="$(find
gazebo_simulation)/world/complete_hotel.sdf"/>
  <arg name="robot_file" default="vehicle.xacro"/>
  <param name="world_file" value="$(arg world_file)"/>
  <param name="robot_file" value="$(arg robot_file)"/>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(arg world_file)"/>
    <arg name="paused" value="$(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>
  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
description)/urdf/$(arg robot_file)'"/>
  <node name="spawn" pkg="gazebo_ros" type="spawn_model" output="screen" args="-
urdf -param robot_description -x $(arg x) -y $(arg y) -z $(arg z) -R $(arg roll)
-P $(arg pitch) -Y $(arg yaw) -model vehicle"/>
</launch>
```

7.1.5 amcl.launch

```
<?xml version="1.0"?>
<launch>
  <master auto="start"/>
  <arg name="map_file" default="hotel_map.yaml"/>
  <param name="map_file" value="$(arg map_file)"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />
  <node pkg="tf" type="static_transform_publisher" name="map_odom_broadcaster"
args="0 0 0 0 0 0 map odom 100"/>
  <node pkg="amcl" type="amcl" name="amcl" output="screen">
    <!--remap from="scan" to="vehicle/laser/scan"/>
    <param name="odom_frame_id" value="odom"/>
    <param name="odom_model_type" value="diff-corrected"/>
    <param name="base_frame_id" value="base_link"/>
    <param name="update_min_d" value="0.5"/>
    <param name="update_min_a" value="1.0"/-->
    <rosparam file="$(find navigation)/config/amcl.yaml" command="load" />
  </node>
  <node pkg="rosservice" type="rosservice" name="global_loc" args="call --wait
/global_localization"/>
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find navigation)/config/costmap_common.yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find navigation)/config/costmap_common.yaml"
command="load" ns="local_costmap" />
    <rosparam file="$(find navigation)/config/local_costmap.yaml" command="load"
/>
    <rosparam file="$(find navigation)/config/global_costmap.yaml"
command="load" />
    <rosparam file="$(find navigation)/config/dwa_local_planner.yaml"
command="load" />
    <rosparam file="$(find navigation)/config/teb_local_planner.yaml"
command="load" />
    <rosparam file="$(find navigation)/config/trayectory_local_planner.yaml"
command="load" />
    <rosparam file="$(find navigation)/config/global_planner.yaml"
command="load" />
    <rosparam file="$(find navigation)/config/nav_fn_global_planner.yaml"
command="load" />
    <rosparam file="$(find navigation)/config/move_base.yaml" command="load" />
    <remap from="cmd_vel" to="cmd_vel"/>
    <remap from="odom" to="odom"/>
    <remap from="scan" to="vehicle/laser/scan"/>
  </node>
</launch>
```

7.1.6 gmapping.launch

```
<?xml version="1.0"?>
<launch>
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>
  <node pkg="gmapping" name="slam_gmapping" type="slam_gmapping"
output="screen">
    <param name="delta" value="0.01"/>
    <param name="xmin" value="-20"/>
    <param name="xmax" value="20"/>
    <param name="ymin" value="-20"/>
    <param name="ymax" value="20"/>
    <remap from="scan" to="vehicle/laser/scan"/>
    <param name="base_frame" value="base_link" />
    <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"/>
  </node>
</launch>
```

7.1.7 vehicle_teleop.launch

```
<?xml version="1.0"?>
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key"
name="turtlebot_teleop_keyboard"  output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="cmd_vel"/>
  </node>
</launch>
```

7.1.8 load_path.launch

```
<?xml version="1.0"?>
<launch>
  <env name="ROSCONSOLE_FORMAT"
value="[${severity}][${thread}][${node}/${function}:${line}]: ${message}"/>
  <arg name="plan_file" default="path.bag"/>
  <param name="plan_file" value="$(arg plan_file)" />
        <arg name="n_iterations" default="2"/>
  <param name="n_iterations" value="$(arg n_iterations)" />
        <arg name="distance_between_obstacles" default="3.0"/>
  <param name="distance_between_obstacles" value="$(arg
distance_between_obstacles)" />
  <arg name="robot_radius" default="0.8"/>
  <param name="robot_radius" value="$(arg robot_radius)" />
  <arg name="max_robot_speed" default="0.8"/>
  <param name="max_robot_speed" value="$(arg max_robot_speed)" />
  <arg name="obstacle_length" default="0.5"/>
  <param name="obstacle_length" value="$(arg obstacle_length)" />
  <arg name="max_obstacle_shiftment" default="0.1"/>
  <param name="max_obstacle_shiftment" value="$(arg max_obstacle_shiftment)" />
```

```
  <arg name="timeout_factor" default="10"/>
  <param name="timeout_factor" value="$(arg timeout_factor)" />
  <node pkg="path_utilities" type="load_path.py" name="load_path"
output="screen">
        <param name="goal_frame_id" value="map"/>
    <param name="plan_file" value="$(arg plan_file)" />
    <param name="n_iterations" value="$(arg n_iterations)" />
    <param name="obstacle_length" value="$(arg obstacle_length)" />
    <param name="robot_radius" value="$(arg robot_radius)" />
    <param name="max_robot_speed" value="$(arg max_robot_speed)" />
    <param name="timeout_factor" value="$(arg timeout_factor)" />
    <param name="max_obstacle_shiftment" value="$(arg max_obstacle_shiftment)"
/>
    <param name="distance_between_obstacles" value="$(arg
distance_between_obstacles)" />
  </node>
</launch>
```

7.1.9 save_path.launch

```
<?xml version="1.0"?>
<launch>
        <env name="ROSCONSOLE_FORMAT"
value="[${severity}][${thread}][${node}/${function}:${line}]: ${message}"/>
        <arg name="output_file" default="path.bag"/>
  <node pkg="path_utilities" type="save_path.py" name="save_path"
output="screen">
        <param name="goal_frame_id" value="map"/>
    <param name="output_file" value="$(arg output_file)" />
  </node>
</launch>
```

7.2 Algunos ficheros de la simulación a alto nivel

7.2.1 WheelChairsSimulation.java

```
package
es.ull.autonomous_vehicles_system_simulation.high_level_simulation.simulation;

import java.util.ArrayList;

import
es.ull.autonomous_vehicles_system_simulation.high_level_simulation.results_struc
tures.ROSResults;
import es.ull.iis.function.TimeFunction;
import es.ull.iis.function.TimeFunctionFactory;
import es.ull.iis.function.TimeFunctionParams;
import es.ull.iis.simulation.condition.ResourceTypeAcquiredCondition;
import es.ull.iis.simulation.model.ElementType;
import es.ull.iis.simulation.model.ResourceType;
import es.ull.iis.simulation.model.Simulation;
```

```
import es.ull.iis.simulation.model.SimulationPeriodicCycle;
import es.ull.iis.simulation.model.SimulationTimeFunction;
import es.ull.iis.simulation.model.TimeDrivenElementGenerator;
import es.ull.iis.simulation.model.TimeUnit;
import es.ull.iis.simulation.model.WorkGroup;
import es.ull.iis.simulation.model.flow.ActivityFlow;
import es.ull.iis.simulation.model.flow.DelayFlow;
import es.ull.iis.simulation.model.flow.ExclusiveChoiceFlow;
import es.ull.iis.simulation.model.flow.ReleaseResourcesFlow;
import es.ull.iis.simulation.model.flow.RequestResourcesFlow;

public class WheelChairsSimulation extends Simulation {

	/*************
	 * CONSTANTS *
	 *************/
	// Model Events
	final static String STR_REQ_CHAIR = "Request chair";
	final static String STR_REQ_JANITOR = "Request janitor";
	final static String STR_REL_CHAIR = "Drop chair";
	final static String STR_REL_JANITOR = "Drop janitor";
	final static String STR_M_APPOINTMENT = "Appointment with manual chair";
	final static String STR_A_APPOINTMENT = "Appointment with autonomus
chair";
	final static String STR_M_STAND = "Stand up from a manual chair";
	final static String STR_A_STAND = "Stand up from an autonomus chair";

	// Predefined events times (Random uniform distribution)
	final private static ElementReplicableTimeFunction T_APPOINTMENT
		= new
ElementReplicableTimeFunction(TimeFunctionFactory.getInstance("UniformVariate",
600, 900));
	final private static ElementReplicableTimeFunction T_M_SEAT
		= new
ElementReplicableTimeFunction(TimeFunctionFactory.getInstance("UniformVariate",
60, 180));
	final private static ElementReplicableTimeFunction T_A_SEAT
		= new
ElementReplicableTimeFunction(TimeFunctionFactory.getInstance("UniformVariate",
60, 180));
	final private static ElementReplicableTimeFunction T_M_STAND
		= new
ElementReplicableTimeFunction(TimeFunctionFactory.getInstance("UniformVariate",
60, 180));
	final private static ElementReplicableTimeFunction T_A_STAND
		= new
ElementReplicableTimeFunction(TimeFunctionFactory.getInstance("UniformVariate",
60, 180));

	// Model elements and sources
	final static String STR_SECTION = "Segment";
	public final static String STR_JANITOR = "Janitor";
	public final static String STR_DOCTOR = "Doctor";
	final static String STR_PATIENT = "Patient";
	public final static String STR_AUTO_CHAIR = "Autonomous Chair";
	public final static String STR_MANUAL_CHAIR = "Manual Chair";

	// Simulation times (in seconds):
	/** Simulation start sencond */
	//final private static long START_TIME = 0;
	/** Simulation length: 8 hours (in seconds) */
```

```
        //final private static long END_TIME = 28800;
        /** Last arrival of patients: one hour before simulation end (7h) */
        //final private static long LAST_ARRIVAL = 25200;

        public final static TimeUnit unit = TimeUnit.SECOND;

        /** End condition for simulation */
        final private WheelChairsEndListener END_CONDITION;

        /**********************
         * SEGMENTS TIME DATA *
         **********************/
        // Instantiate the segments times container that will build from low
simulation data
        private static ArrayList<ElementReplicableTimeFunction> segments;

        /**********************
         * DEFAULT CONSTRUCTOR *
         **********************/
        public WheelChairsSimulation(Integer id, Integer nJanitors, Integer
nDoctors, Integer nAutoChairs,
                        Integer nManualChairs, Integer patientsPerArrival,
Integer minutesBetweenArrivals,
                        Double manualFactor, ROSResults rosResults,
                        Long startTime, Long endTime) {
                // Set simulation information
                super(id, "", TimeUnit.SECOND, startTime, endTime);
                setSegments(rosResults.getElementReplicableTimeFunctions());
                // Set the simulation end condition
                END_CONDITION = new WheelChairsEndListener();
                addInfoReceiver(END_CONDITION);

                restartTimeFunctions();
                // El paciente es el elemento del modelo

                final ElementType etPatient = new ElementType(this, STR_PATIENT);

                // Los recursos son: Doctores y Silla

                // Bedeles
                final ResourceType rtJanitor = new ResourceType(this,
STR_JANITOR);
                rtJanitor.addGenericResources(nJanitors);
                // Sillas autónomas
                final ResourceType rtAChair = new ResourceType(this,
STR_AUTO_CHAIR);
                rtAChair.addGenericResources(nAutoChairs);
                // Sillas manuales
                final ResourceType rtMChair = new ResourceType(this,
STR_MANUAL_CHAIR);
                rtMChair.addGenericResources(nManualChairs);
                // Doctores
                final ResourceType rtDoctor = new ResourceType(this, STR_DOCTOR);
                rtDoctor.addGenericResources(nDoctors);

                //Definición de los flujos de trabajo

                final WorkGroup wgJanitorMChair = new WorkGroup(this, new
ResourceType[]
                                {rtJanitor, rtMChair}, new int[] {1,1});
                final WorkGroup wgJanitor = new WorkGroup(this, rtJanitor, 1);
```

```
                final WorkGroup wgJanitorAChair = new WorkGroup(this, new
ResourceType[]
                                {rtJanitor, rtAChair}, new int[] {1,1});
                final WorkGroup wgAppointment = new WorkGroup(this, new
ResourceType[] {rtDoctor} , new int [] {1});

                // Creamos todos los pasos del proceso
                final DelayFlow[] actASections = new DelayFlow
[getSegments().size()];
                final DelayFlow[] actASectionsBack = new DelayFlow
[getSegments().size()];
                final DelayFlow[] actMSections = new DelayFlow
[getSegments().size()];
                final DelayFlow[] actMSectionsBack = new DelayFlow
[getSegments().size()];

                // Solicitud de recursos: sillas de ruedas y bedeles
                final RequestResourcesFlow reqChair = new
RequestResourcesFlow(this, STR_REQ_CHAIR, 1, 2);
                reqChair.addWorkGroup(0, wgJanitorAChair, T_A_SEAT);
                reqChair.addWorkGroup(1, wgJanitorMChair, T_M_SEAT);
                final RequestResourcesFlow reqJanitor = new
RequestResourcesFlow(this, STR_REQ_JANITOR, 1, 1);
                reqJanitor.addWorkGroup(wgJanitor);

                // Dejar de usar recursos: sillas de ruedas y bedeles.
                final ReleaseResourcesFlow relJanitorBeforeAppointment = new
ReleaseResourcesFlow(
                                this, STR_REL_JANITOR, 1, wgJanitor);
                final ReleaseResourcesFlow relJanitorAfterSeat = new
ReleaseResourcesFlow(this, STR_REL_JANITOR, 1, wgJanitor);
                final ReleaseResourcesFlow relChair = new
ReleaseResourcesFlow(this, STR_REL_CHAIR, 1);

                // Creamos una actividad de consulta por cada tipo de silla
                final ActivityFlow actMAppointment = new ActivityFlow(this,
STR_M_APPOINTMENT);
                actMAppointment.addWorkGroup(0, wgAppointment, T_APPOINTMENT);
                final ActivityFlow actAAppointment = new ActivityFlow(this,
STR_A_APPOINTMENT);
                actAAppointment.addWorkGroup(0, wgAppointment, T_APPOINTMENT);

                // Creamos los tramos de la ruta que siguen las sillas
                for (int i = 0; i < getSegments().size(); i++) {
                        // Sillas autónomas
                        // Ruta de ida
                        actASections[i] = new DelayFlow(this, STR_SECTION + i,
getSegments().get(i));
                        // Ruta de vuelta
                        actASectionsBack[getSegments().size() - i - 1] = new
DelayFlow(this, STR_SECTION +
                                        " (back)" + (getSegments().size() - i -
1), getSegments().get(i));
                        // Sillas manuales
                        // Ruta de ida
                        actMSections[i] = new DelayFlow(this, STR_SECTION + i,
                                        new ModifiedFunction(getSegments().get(i),
manualFactor, 0.0));
                        // Ruta de vuelta
                        actMSectionsBack[getSegments().size() - i - 1] = new
DelayFlow(this, STR_SECTION + " (back)" +
```

```
                                        (getSegments().size() - i - 1), new
ModifiedFunction(getSegments().get(i), manualFactor, 0.0));
                }

                // Conectamos los tramos de la ruta que siguen las sillas
                for (int i = 1; i < getSegments().size(); i++) {
                        actASections[i-1].link(actASections[i]);
                        actASectionsBack[i - 1].link(actASectionsBack[i]);
                        actMSections[i-1].link(actMSections[i]);
                        actMSectionsBack[i - 1].link(actMSectionsBack[i]);
                }

                final ExclusiveChoiceFlow condFlow0 = new
ExclusiveChoiceFlow(this);

                // If the chair is automated, release the janitor after being
seated
                reqChair.link(condFlow0);
                condFlow0.link(relJanitorAfterSeat, new
ResourceTypeAcquiredCondition(rtAChair)).link(actASections[0]);
                condFlow0.link(actMSections[0]);


                actMSections[getSegments().size() -
1].link(relJanitorBeforeAppointment).link(actMAppointment)
                        .link(reqJanitor).link(actMSectionsBack[0]);
                actASections[getSegments().size() -
1].link(actAAppointment).link(actASectionsBack[0]);

                // Creamos una actividad para levantarse de cada tipo de silla
                final DelayFlow delMStand = new DelayFlow(this, STR_M_STAND,
T_M_STAND);
                final ActivityFlow actAStand = new ActivityFlow(this,
STR_A_STAND);

                // En el caso de las sillas automáticas, requiere un bedel
                actAStand.addWorkGroup(0, wgJanitor, T_A_STAND);
                actMSectionsBack[getSegments().size() -
1].link(delMStand).link(relChair);
                actASectionsBack[getSegments().size() -
1].link(actAStand).link(relChair);

                //Horario de llegada de pacientes
                final SimulationPeriodicCycle arrivalCycle = new
SimulationPeriodicCycle(unit, 0,
                                new SimulationTimeFunction(unit,
"ConstantVariate", minutesBetweenArrivals * 60),
                                (int)(getLastArrival(startTime, endTime) /
(minutesBetweenArrivals * 60)));
                new TimeDrivenElementGenerator(this, patientsPerArrival,
etPatient, reqChair, arrivalCycle);
                addInfoReceiver(END_CONDITION);
        }

        /***********
         * METHODS *
         **********/

        /** Set last arrival one hour before the end time.
         * @param startTime
         * @param endTime
```

```
	 * @return value in seconds of the last arrival */
	private static Long getLastArrival(Long startTime, Long endTime) {
		return ((endTime - startTime) > 3600) ? (endTime - 3600) : 0;
	}

	private static void restartTimeFunctions() {
		T_APPOINTMENT.restart();
		T_A_SEAT.restart();
		T_M_SEAT.restart();
		T_A_STAND.restart();
		T_M_STAND.restart();
		for (ElementReplicableTimeFunction function : getSegments()) {
			function.restart();
		}
	}

	public static void resetTimeFunctions() {
		T_APPOINTMENT.reset();
		T_A_SEAT.reset();
		T_M_SEAT.reset();
		T_A_STAND.reset();
		T_M_STAND.reset();
		for (ElementReplicableTimeFunction function : getSegments()) {
			function.reset();
		}
	}

	/*******************************
	 * TIME FUNCTION MODIFIER CLASS *
	 *******************************/

	private static class ModifiedFunction extends TimeFunction {
		private double scale;
		private double offset;
		private TimeFunction function;

		/**
		 * @param scale
		 * @param offset
		 */
		public ModifiedFunction(TimeFunction function, double scale,
double offset) {
			this.scale = scale;
			this.offset = offset;
			this.function = function;
		}

		@Override
		public double getValue(TimeFunctionParams params) {
			return scale * function.getValue(params) + offset;
		}

		@Override
		public void setParameters(Object... params) {
			// TODO Auto-generated method stub

		}

	}

	/***************
```

```
         * GET AND SET *
         **************/

        /** @return the segments */
        public static ArrayList<ElementReplicableTimeFunction> getSegments() {
                return segments;
        }


        /** @param segments the segments to set */
        public static void setSegments(ArrayList<ElementReplicableTimeFunction>
segments) {
                WheelChairsSimulation.segments = segments;
        }


}
```

Bibliografía

[1] R. Doménech, J. R. García, M. Montañez y A. Neut, «bbvaresearch,» 19 marzo 2018. [En línea]. Available: https://www.bbvaresearch.com/wp-content/uploads/2018/03/Cuan-vulnerable-es-el-empleo-en-Espana-a-la-revolucion-digital.pdf.

[2] Observatori de la Industria 4.0, «premsa.gencat.cat,» [En línea]. Available: http://premsa.gencat.cat/pres_fsvp/docs/2018/04/25/13/40/08044276-5fe4-4f7f-86d8-b44b88c3374c.pdf.

[3] G. Motors, «www.gm.com,» [En línea]. Available: https://www.gm.com/.

[4] ANSYS, «www.ansys.com,» [En línea]. Available: https://www.ansys.com/.

[5] ATKINS, «assets.publishing.service.gov.uk,» [En línea]. Available: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/530091/impacts-of-connected-and-autonomous-vehicles-on-traffic-flow-summary-report.pdf.

[6] Toyota, «www.toyota.co.uk,» [En línea]. Available: https://www.toyota.co.uk/world-of-toyota/stories-news-events/toyota-project-blaid.

[7] SAE, «www.sae.org,» [En línea]. Available: https://www.sae.org/.

[8] W. v. d. Aalst y A. t. Hofstede, «Workflow Patterns,» [En línea]. Available: http://www.workflowpatterns.com/.

[9] PSIGHOS, «github.com/ull-isaatc/sighos,» [En línea]. Available: https://github.com/ull-isaatc/sighos.

[10] ROS, «www.ros.org,» [En línea]. Available: http://www.ros.org/.

[11] Rviz, «wiki.ros.org/rviz,» [En línea]. Available: http://wiki.ros.org/rviz.

[12] Gazebo, «gazebosim.org,» [En línea]. Available: http://gazebosim.org/.

[13] Wikipedia, «WebSocket,» [En línea]. Available: https://es.wikipedia.org/wiki/WebSocket.

[14] MongoDB, «www.mongodb.com,» [En línea]. Available: https://www.mongodb.com/.

[15] ROS, «Topics,» [En línea]. Available: http://wiki.ros.org/Topics.

[16] Wikipedia, «List of moments of inertia,» [En línea]. Available: https://en.wikipedia.org/wiki/List_of_moments_of_inertia.

[17] Blender, «Blender,» [En línea]. Available: https://www.blender.org/.

[18] SketchUp, «SketchUp,» [En línea]. Available: https://www.sketchup.com/es.

[19] A. Koubaa, Robot Operating System (ROS). The Complete Reference (Volume 2), 2017.

[20] ROS, «Gazebo plugins in ROS,» [En línea]. Available: http://gazebosim.org/tutorials?tut=ros_gzplugins.

[21] Wikipedia, «LIDAR,» [En línea]. Available: https://es.wikipedia.org/wiki/LIDAR.

[22] Wikipedia, «GPS,» [En línea]. Available: https://es.wikipedia.org/wiki/GPS.

[23] Wikipedia, «Sensores de ultrasonido,» [En línea]. Available: https://es.wikipedia.org/wiki/Sensor_ultras%C3%B3nico.

[24] PyPi, «pyproj,» [En línea]. Available: https://pypi.org/project/pyproj/.

[25] ROS, «actionlib,» [En línea]. Available: http://wiki.ros.org/actionlib.

[26] ROS, «rosbridge_suite,» [En línea]. Available: http://wiki.ros.org/rosbridge_suite.

[27] stefie10, «java_rosbridge,» [En línea]. Available: https://github.com/h2r/java_rosbridge.

[28] Eclipse, «Jetty,» [En línea]. Available: http://www.eclipse.org/jetty/.

[29] ROS, «navigation,» [En línea]. Available: http://wiki.ros.org/navigation.

[30] ROS, «amcl,» [En línea]. Available: http://wiki.ros.org/amcl.

[31] Wikipedia, «Monte Carlo localization,» [En línea]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_localization.

[32] Wikipedia, «Localización y modelado simultáneos,» [En línea]. Available: https://es.wikipedia.org/wiki/Localizaci%C3%B3n_y_modelado_simult%C3%A1neos.

[33] ROS, «gmapping,» [En línea]. Available: http://wiki.ros.org/gmapping.

[34] ROS, «TurtleBor,» [En línea]. Available: http://wiki.ros.org/Robots/TurtleBot.

[35] ROS, «map_server,» [En línea]. Available: http://wiki.ros.org/map_server.

[36] Wikipedia, «Filtro de Kalman,» [En línea]. Available: https://es.wikipedia.org/wiki/Filtro_de_Kalman.

[37] G. Grisetti, C. Stachniss y W. Burgard, «Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters,» [En línea]. Available: http://www2.informatik.uni-freiburg.de/~stachnis/pdf/grisetti07tro.pdf.

[38] Wikipedia, «A*,» [En línea]. Available: https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*.

[39] Wikipedia, «Dijkstra,» [En línea]. Available: https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra.

[40] ROS, «dwa_local_planner,» [En línea]. Available: http://wiki.ros.org/dwa_local_planner.

[41] ROS, «teb_local_planner,» [En línea]. Available: http://wiki.ros.org/teb_local_planner.

[42] A. Mahtani, L. Sánchez, E. Fernández y A. Martinez, Effective Robotics Programming with ROS (Third Edition), Birmingham: Packt Publishing Ltd., 2016.

[43] L. Joseph, ROS Robotics Projects, Birmingham: Packt Publishing Ltd., 2017.

[44] A. Martinez y E. Fernández, Learning ROS for Robotics Programming, Birmingham: Packt Publishing Ltd., 2013.

[45] L. Joseph, Mastering ROS for Robotics Programming, Birmingham: Packt Publishing Ltd., 2015.

[46] C. Fairchild y D. T. L. Harman, ROS Robotics By Example, Birmingham: Packt Publishing Ltd., 2016.

[47] M. Quigley, B. Gerkey y W. D. Smart, Programming Robots with ROS, Gravenstein Highway North: O'Reilly Media, 2016.

[48] Wikipedia, «URI,» [En línea]. Available: https://es.wikipedia.org/wiki/Identificador_de_recursos_uniforme.