Capítulo 4

Práctica: Navegación global de un Taxi con GPP

Una vez explicado el contexto, los objetivos y las herramientas empleadas en este proyecto, en este capítulo se detallarán las mejoras de infraestructura y aplicación en la práctica de JdeRobot-Academy denominada “*global\_navigation*”, así como el árbitro creado y la solución realizada. Esta práctica ya formaba parte del entorno JdeRobot-Academy, pero se han realizado diferentes mejoras que se describirán a lo largo del capítulo.

* 1. Enunciado de la práctica

El propósito de esta práctica es que un taxi autónomo sea capaz de navegar desde un punto de la ciudad a cualquier otro punto. El taxi dispondrá de un mapa de la ciudad y un sensor GPS que le proporciona una estimación de su posición en la ciudad. Este vehículo posee un actuador de movimiento basado en velocidad lineal y velocidad de giro. Todos estos elementos permitirán realizar un algoritmo de planificación de rutas, además de un algoritmo de pilotaje que se fundamenta en la posición.

En esta práctica el alumno deberá programar el algoritmo de navegación global Gradient Path Planning (GPP) para poder obtener una planificación de la ruta que ha de seguir desde la posición actual del robot hasta la posición definida por el usuario en la interfaz gráfica (GUI). Una vez calculada la ruta más corta entre los puntos mencionados, en el GUI se podrá visualizar la ruta en el mapa 2D, así como una imagen con el campo de gradiente calculado por el código del alumno (nos dará información acerca de la ruta en función de la distancia). Una vez calculada la ruta, el alumno deberá proporcionar un algoritmo de pilotaje para poder alcanzar el objetivo. Este algoritmo deberá tener en cuenta el campo calculado mediante el GPP y la posición del taxi.

El algoritmo de cálculo del campo se realizará en una iteración sin tiempo límite, y el algoritmo de pilotaje responde a un control reactivo.

* 1. Diseño general de la práctica

En este apartado se dará una descripción de cómo se ha diseñado la práctica, es decir, se explicará la jerarquía que distribuye el comportamiento de la práctica en tareas más sencillas.

4.2.1 Diseño general

Las prácticas de JdeRobot se realizan sobre el entorno de prácticas JdeRobot/Academy. Este entorno se encarga de esconder la complejidad de diseñar la infraestructura, la aplicación o cualquier herramienta necesaria para las prácticas, evitando que el alumno tenga que lidiar con estas dificultades. Con este entorno se pretende que el alumno pueda afianzar los conocimientos teóricos sobre robótica y que pueda programar el comportamiento de los robots de forma práctica.

Vamos a proceder a explicar los puntos necesarios para el desarrollo de una práctica. Lo primero que debemos crear es la infraestructura de la práctica. La infraestructura engloba la creación de los modelos necesarios, así como los plugins que van a necesitar estos modelos; además se deberá crear un entorno (mundo de Gazebo) por donde podrá navegar nuestro robot.

Es importante mencionar que también se debe crear un archivo de configuración que termina con extensión .cfg para poder indicar los puertos que emplea cada plugin que tiene el robot. Con este archivo de configuración indicamos los puertos que tiene que usar nuestra aplicación para comunicarse con gazeboserver (este componente de JdeRobot nos permite acceder a las interfaces de sensores y actuadores de los robots simulados en Gazebo).

El siguiente paso es la creación de la interfaz gráfica, que nos facilitará la resolución de las prácticas. En esta interfaz gráfica se mostrará una ventana donde podemos ver un mapa del mundo creado en Gazebo así como la posición actual del robot. Es importante mencionar que se ha creado el componente Grid, el cual será invocado a través de la interfaz gráfica. Esta clase nos proporcionará una rejilla, la cual es necesaria para la resolución de esta práctica.

El último paso es crear el árbitro que nos permitirá calificar la práctica. Este árbitro también tiene un visor gráfico. El fichero (referee.py) en el que se programa el comportamiento del árbitro está escrito en PyQt5 al igual que la interfaz gráfica.

Además, se proporcionará un archivo MyAlgorithm.py, que servirá como plantilla para que el alumno pueda programar la solución de navegación del robot.

Todos estos ficheros que son necesarios para crear el entorno de la práctica se explicarán en más profundidad en los puntos 4.3 (infraestructura), 4.4 (aplicación gráfica), 4.5 (árbitro) y 4.6 (solución).

4.2.2 Hilos de ejecución de la práctica

Un hilo de ejecución (thread) es una característica que permite a una aplicación realizar varias tareas a la vez. En la práctica se ha dividido el trabajo en diferentes partes. Por lo que emplearemos hilos de ejecución para poder llevar a cabo diferentes tareas simultáneamente. En esta práctica existen tres procesos diferenciados:

* Hilo de control: Este hilo es el encargado de actualizar los datos de los sensores y los actuadores a través de las interfaces ICE. El tiempo de refresco de este hilo es muy importante, y debe ser un periodo de tiempo muy corto, ya que este componente se encarga de establecer la velocidad y la dirección del robot en todo momento. Si este tiempo fuera muy grande, las decisiones que modifican la trayectoria del robot podrían ser incorrectas. En este caso el hilo(ThreadMotors) que se emplea es para enviar órdenes a los motores y se actualiza cada 80 milisegundos.
* Hilo de la interfaz gráfica de usuario (GUI): Este hilo se encarga de ir actualizando la GUI. Además, cuenta con los manejadores de eventos del GUI, donde se ejecuta el código de MyAlgorithm. El intervalo de actualización de este hilo es muy importante. Es necesario que este intervalo sea pequeño, ya que tenemos que mostrar la posición del robot en el mapa en tiempo real. El hilo de ejecución de la GUI (ThreadGUI) se actualizará cada 50 ms.
* Hilo del árbitro: Este hilo será el encargado de ir actualizando los parámetros que se muestran en el árbitro. El intervalo de actualización de este hilo debe ser pequeño, ya que como en el caso de la GUI, debemos garantizar al usuario que los parámetros que le mostramos son actuales. El hilo de ejecución del referee se actualiza cada 50 ms.

4.2.3 Cómo ejecutar la práctica

La práctica “global\_navigation” se puede ejecutar de diferentes formas, con un script o sin script. El script se ha escrito para que sea más cómodo para los alumnos lanzar la práctica con un único terminal.

Si no queremos hacer uso del script se puede lanzar la práctica abriendo tres terminales y ejecutando en cada uno de ellos:

* Lanzar Gazebo: gazebo cityLarge.world
* Ejecutar la práctica y lanzar la GUI: python2 globalNavigation.py --mapConfig=taxiMap.conf --Ice.Config=teleTaxi.cfg
* Lazar el árbitro: python2 referee.py --mapConfig=taxiMap.conf --Ice.Config=teleTaxi.cfg

Si queremos ejecutar la práctica con el script (run\_it.sh) debemos tener en cuenta que, si la máquina donde se va a ejecutar es muy poco potente, será mejor no lanzar la interfaz de Gazebo por defecto. La práctica se puede ejecutar de la siguiente manera:

* Ejecución sin ver el mundo de Gazebo: ./run\_it.sh
* Ejecución viendo el mundo de Gazebo: ./run\_it.sh GUI

* 1. Infraestructura

En este punto se comentará el entorno creado para poder llevar a cabo la práctica de navegación global. Comenzamos con una descripción del robot que se ha empleado para el desarrollo de la práctica, así como una descripción de los actuadores y sensores que posee este robot, y una explicación del entorno por el que se moverá nuestro robot.

* + 1. Taxi\_Holo

El robot que se ha empleado en esta práctica es un nuevo modelo de robot creado en un programa de modelado 3D (como pueden ser Blender, SketchUp, etc). Este robot, en la versión anterior de “global\_navigation”, se llamaba yellowTaxi. Es un robot creado para poder moverse de forma autónoma o teledirigida por un escenario. Este robot posee sensores de odometría (estudio de la estimación de la posición de vehículos) que le permiten saber cuál es su posición en todo momento; así como motores que le permiten moverse por el escenario de manera adecuada. No posee otros sensores como pueden ser cámaras o lásers.

El modelo yellowTaxi tiene en cuenta las características propias de un automóvil, pero con el aspecto de un taxi. Este robot posee unas dimensiones grandes, ya que mide aproximadamente 5.75 metros de largo, 3 metros de ancho y posee una altura de 3 metros.

Una característica importante de este modelo es que es no holonómico. Holonómico quiere decir que la cantidad de grados de libertad (cada una de las variables necesarias para obtener los movimientos de un cuerpo en el espacio) totales y los que podemos controlar es la misma. Esto es una propiedad importante dado que los coches reales son no holonómicos. Un automóvil tiene tres grados de libertad (x, y, orientación) de los cuales solamente dos son controlables: conducción (aceleración adelante y atrás) y dirección (volante); pero hay ciertos movimientos que son imposibles de realizar, como un movimiento de lado.

El modelo yellowTaxi de la versión antigua de la práctica “global\_navigation” se puede ver en la siguiente imagen.



Figure 4.1: Modelo yellowTaxi

Este modelo tenía fallos en su estructura que hacían que el taxi apenas se pudiera desplazar en el mundo o no fuera capaz de rotar, aunque le enviáramos órdenes de velocidad de tracción y velocidad de rotación muy elevadas. Este inconveniente suponía que el taxi tardara mucho tiempo en llegar a un objetivo cercano o que fuera incapaz de girar suficiente en las curvas, lo que hacía que nuestro taxi se chocara contra las paredes de los obstáculos. Por este motivo, se creó un nuevo modelo de taxi denominado taxi\_holo.

El nuevo modelo de taxi cuenta con sensores de odometría y motores que permiten el movimiento al igual que el modelo antiguo. Este taxi también es capaz de moverse de forma autónoma y teledirigida por el escenario.

El nuevo modelo posee unas dimensiones de menor tamaño, aunque comparado con otros robots tiene grandes dimensiones. El tamaño de este taxi es de 4 metros de largo, 2 metros de ancho y una altura de 1.5 metros. Este taxi pesa 750 kg. Este modelo es también no holonómico como el anterior, pero resuelve de forma eficiente los problemas de movimiento que tenía el antiguo modelo. El nuevo modelo de taxi lo podemos ver en la siguiente imagen.

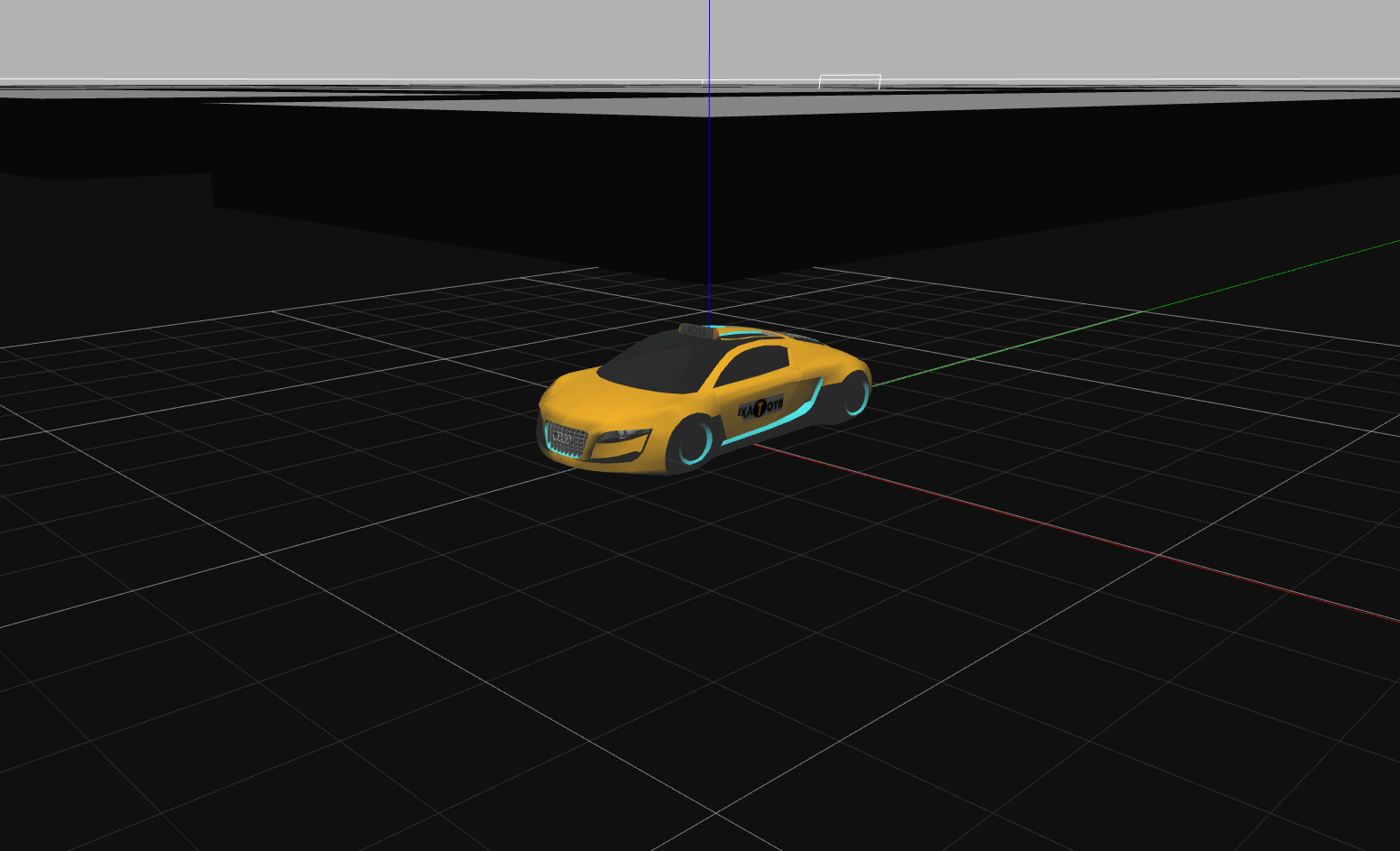


Figure 4.2: Modelo taxi\_holo

* + 1. Sensores de odometría

Los sensores de odometría son ampliamente utilizados en este proyecto (se emplearán en las tres prácticas), ya que son una gran fuente de información para los algoritmos en los que se apoya el pilotaje de nuestro robot. La odometría se emplea para estimar la posición (x, y, orientación) de un robot móvil en todo momento. La posición estimada por parte de los sensores de odometría es relativa a la posición inicial del robot. Por tanto, emplearemos los sensores de odometría para estimar la posición del taxi en el mundo, y a partir de este dato estimar la velocidad que posee el taxi (lo cual se empleará en el árbitro). Los sensores de odometría estiman la posición de las ruedas izquierda y derecha en un intervalo de tiempo determinado. Al conocer las coordenadas de posición anteriores resulta más sencillo poder obtener la nueva posición de nuestro robot. La plataforma JdeRobot nos aísla de la complejidad que conllevan los sensores de odometría, facilitándonos una variable que contiene la posición (x, y, orientación), que podremos usar para estimar la posición del robot en el mundo.

* + 1. Plugins

JdeRobot consta de un amplio repertorio de drivers que facilitan el envío y la recepción de datos de los sensores y actuadores. El lenguaje de programación en el que se desarrollarán estos plugins es C++. En esta práctica se han empleado dos drivers:

* holoCarPose3d: Es el plugin que emplearán los componentes para poder obtener su posición en tiempo real. Este plugin también se utiliza para cambiar la posición de los componentes.
* holoCarMotors: Es el plugin con el que interactúa el componente, es el que permite dotar al componente de velocidad, tanto velocidad de tracción como velocidad de rotación. También modifica los datos de la interfaz de usuario.
  + 1. Modelo cityLarge

El objetivo de esta práctica es que nuestro taxi sea capaz de navegar por una ciudad hasta un punto destino, por lo que tendremos que crear el entorno (ciudad) donde se moverá el taxi. Con este propósito se ha creado un modelo de ciudad llamado “cityLarge”. Este modelo al igual que en el caso del taxi fue creado con una herramienta de modelado 3D. El modelo cityLarge se corresponde con una ciudad de grandes dimensiones. En esta ciudad no veremos casas, semáforos, parques u otros elementos habituales en las ciudades reales, sino que se ha simplificado su creación. Lo que podremos ver en esta ciudad son bloques que se corresponden con obstáculos o carreteras. También podemos ver elementos propios de las carreteras como son rectas, curvas más simples o curvas pronunciadas, así como una zona amplia que se corresponde con una especie de plaza.

Este modelo se ha modificado respecto a la versión antigua, debido a que en dicha versión debajo de la ciudad había un palo de sujeción. Este elemento hacía que el coche adquiriera un poco de movimiento al ejecutar la práctica. Por lo tanto, se ha eliminado este palo de sujeción.

En la imagen 4.3 podemos ver la parte de debajo del modelo cityLarge para observar que había un palo de sujeción de la ciudad. En la imagen 4.4 se observa el nuevo modelo cityLarge sin dicho elemento de sujeción. Por último, en la foto 4.5 tenemos un plano desde arriba de toda la ciudad.

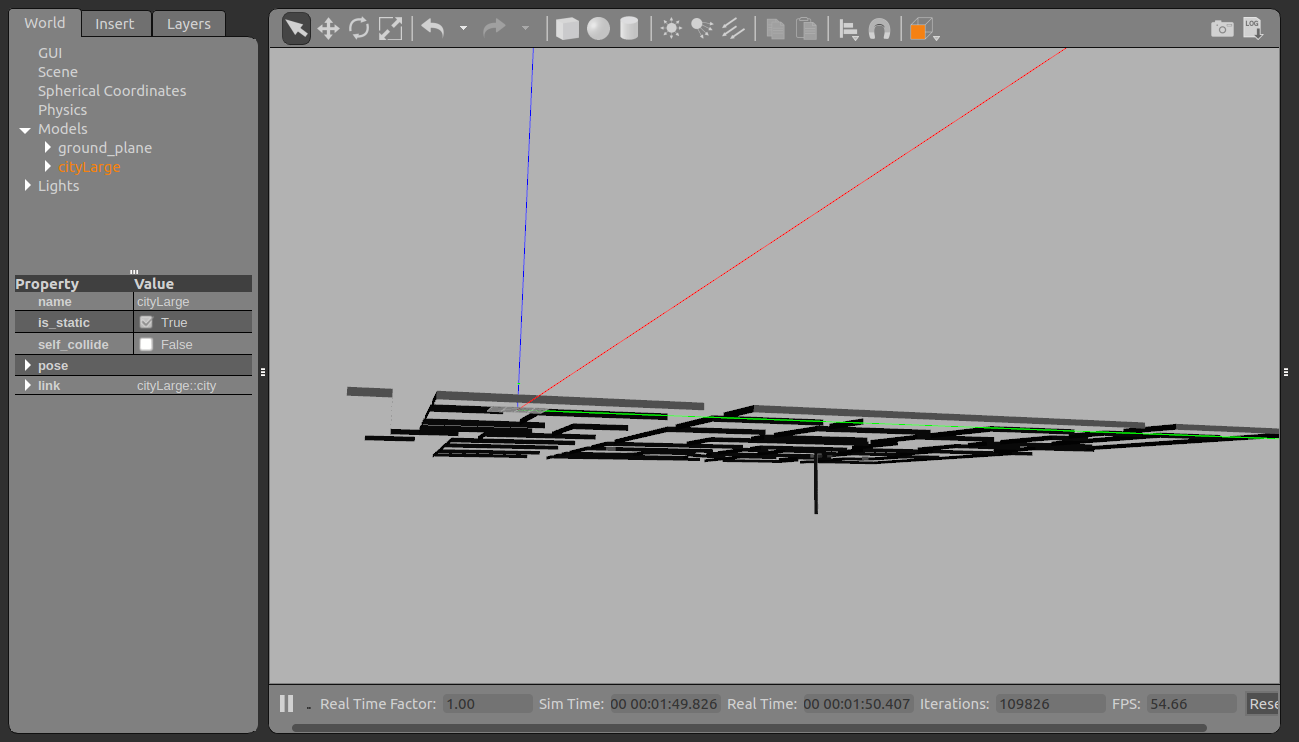


Figure 4.3: Modelo antiguo cityLarge

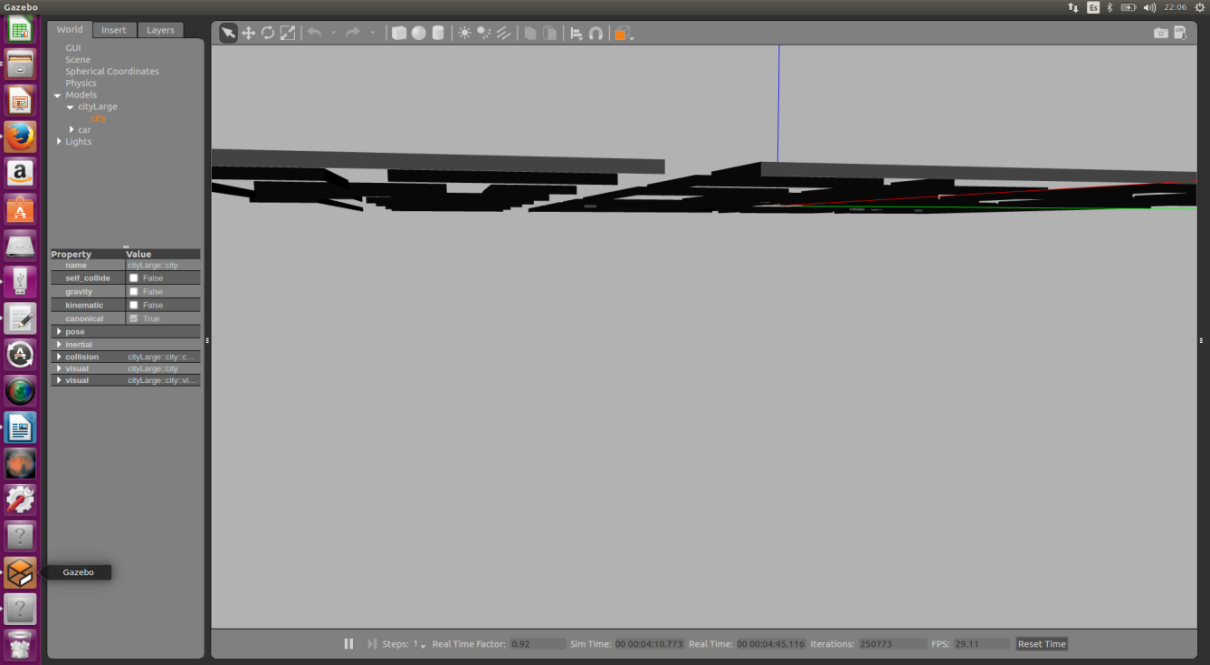


Figure 4.4: Modelo nuevo cityLarge

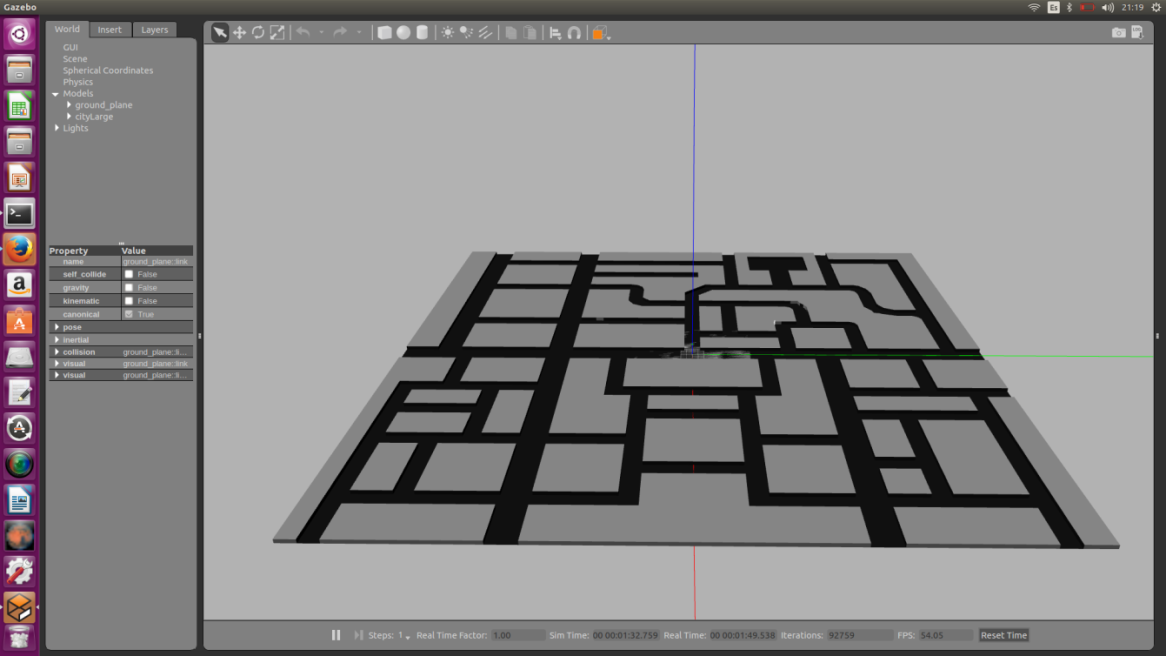


Figure 4.5: Modelo cityLarge desde arriba

* + 1. Mundo de Gazebo

Los mundos que se simulan con Gazebo son mundos 3D. Estos mundos se cargan en ficheros con extensión .world, que no son más que ficheros XML definidos en el lenguaje Simulation Description Format (SDF). Este lenguaje contiene una descripción completa de todos los elementos que tiene el mundo y los robots, incluyendo:

* Escena: Luz ambiente, propiedades del cielo, sombras, etc.
* Modelo: Links, joint, objetos de colisión, sensores, etc.
* Físicas: Gravedad, motor físico, paso del tiempo, colisiones, inercias, etc.
* Plugins: Sobre un mundo, modelo o sensor.
* Luz: Los puntos y origen de la luz.

Todos estos elementos se han tenido en cuenta en el desarrollo de la práctica. Por ello se ha creado un mundo en Gazebo (cityLarge.world) compuesto por el modelo de la ciudad (cityLarge) y el modelo del taxi (taxi\_holo). El antiguo mundo en vez de utilizar el modelo taxi\_holo empleaba el modelo yellow\_taxi. El archivo cityLarge.world tiene el siguiente aspecto:

<?xml version="1.0" ?>

<sdf version="1.4">

<world name="cityLarge">

<!-- My city -->

<include>

<uri>model://cityLarge</uri>

</include>

<!-- My robots -->

<include>

<uri>model://taxi\_holo</uri>

<pose>0 0 0 0 0 0</pose>

</include>

<!-- A global light source -->

<include>

<uri>model://sun</uri>

</include>

</world>

</sdf>

* + 1. Archivo de configuración

En el punto 4.2 hemos mencionado la necesidad de crear un archivo de configuración para que nuestra aplicación pueda comunicarse con gazeboserver. En este fichero se indican los puertos de los plugins que usa el taxi. Este fichero (teleTaxi.cfg) en la práctica tiene el siguiente aspecto.

TeleTaxi.Motors.Proxy = Motors:default -h localhost -p 9999

TeleTaxi.Pose3D.Proxy = Pose3D:default -h localhost -p 9989

TeleTaxi.robot=Pioneer

TeleTaxi.Motors.maxV = 50

TeleTaxi.Motors.maxW = 20

Podemos ver que, en el caso de nuestro taxi, los motores emplean el Puerto 9999, mientras que la pose3d emplea el Puerto 9989. Además, se puede ver en este archivo que se indica al robot la velocidad máxima de tracción y de rotación.

* 1. Aplicación gráfica

En esta sección vamos a explicar qué es la interfaz gráfica o también llamada GUI (Graphic User Interface), el componente Grid (mencionado en el punto 4.2.1), y cómo es nuestra interfaz gráfica.

* + 1. Qué es la interfaz gráfica

La interfaz gráfica de usuario, conocida como GUI, es un programa informático que actúa cómo interfaz de usuario para representar la información y acciones en la interfaz. Esta interfaz suele estar compuesta por imágenes y objetos gráficos que facilitan la interacción con el usuario. Su principal objetivo consiste en facilitar un entorno visual sencillo para permitir la comunicación entre el usuario y el ordenador.

Las interfaces gráficas pueden estar escritas en numerosos lenguajes y emplear múltiples herramientas para su creación. En el caso de nuestra práctica empleamos el lenguaje Python y PyQt5.

* + 1. Componente Grid

El componente Grid se creó en la anterior versión de “global\_navigation”, pero se han modificado algunos aspectos en esta versión.

Este componente es el que se encarga de capturar información del mundo, crear una rejilla donde se almacenará el campo de la expansión y comunicarse con la interfaz gráfica. Este componente será lanzado al ejecutar la práctica y lanzar la GUI.

El componente se crea mediante una clase (Grid) que será instanciada en el programa principal (globalNavigation.py). Cuando es lanzado este componente, se crea una rejilla del tamaño de la imagen del mapa que tiene la interfaz gráfica (400 x 400 píxeles). Esta rejilla se crea para poder guardar información acerca del mundo. Será utilizada por el alumno para guardar los valores del campo del gradiente (será explicado en el punto 4.6) y ayudará a realizar el pilotaje del robot. Además, se inicializa otra rejilla, llamada path para almacenar la ruta más corta, la cual se pintará en verde sobre el mapa.

Al ejecutar la práctica también se inicializa la variable map. Esta variable se inicializa desde la interfaz gráfica y es una imagen binaria del mundo. Esta imagen tiene representados en negro (valor 0) los píxeles que forma parte de obstáculos, mientras que los píxeles que forman parte de la carretera serán blancos (valor 255). La imagen será una imagen de tres canales, aunque para la solución de la práctica bastará con usar un canal. Este mapa ayudará al alumno a realizar la solución, ya que se puede extraer mucha información de la misma. Esta imagen del mapa ha sido cambiada en la versión actual, debido a que la anterior estaba algo torcida y faltaba una parte del mapa en la parte inferior. Se puede ver en la siguiente imagen la diferencia entre la imagen antigua y la actual.

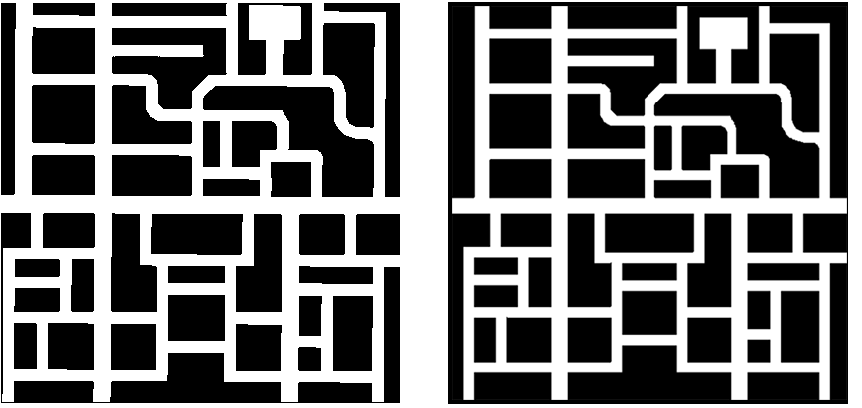


Figure 4.6: Imagen antigua del mapa(izquierda) e imagen actual (derecha)

Este objeto posee una función (llamada setDestiny) que será despertada una vez hagamos click sobre la interfaz gráfica para seleccionar el destino deseado. Esta función almacenará las coordenadas del destino en una variable (destiny). Esta variable será de gran utilidad para la resolución de la práctica.

El objeto Grid cuenta con funciones que le permiten relacionarse con el mapa del mundo. Estas funciones son las siguientes:

* grid.getMap(): devuelve la imagen del mapa que se está mostrando.
* grid.getDestiny(): devuelve el destino seleccionado en la interfaz gráfica. Este destino se devuelve como una tupla (x,y).
* grid.getPose(): devuelve la posición respecto al mapa. También será una tupla (x,y).
* grid. showGrid(): crea una ventana en la que representa los valores del campo que se le han asignado a la rejilla. Los valores más pequeños del campo tendrán un color más cercano a negro, y se irán haciendo más claros a medida que se trate de valores superiores.

Este objeto Grid, también posee funciones que le permiten interactuar con la rejilla (donde se apunta la distancia al destino en ella). Los valores de esta rejilla son de tipo float. Las funciones son:

* grid.setVal(x, y, val): Esta función establece el valor val en la posición indicada (x, y).
* grid.getVal(x, y): devuelve el valor de la posición (x, y) del grid.

El objeto Grid consta de funciones que interactúan con la rejilla que posee la ruta más corta. Los puntos de la rejilla con valor 0 serán ignorados, mientras que los valores superiores a 0 serán considerados parte del camino. Las funciones para interactuar con esta rejilla son:

* grid.setPathVal(x,y, val): establece el valor val en la posición indicada (x, y).
* grid.getPathVal(x,y): devuelve el valor de la posición indicada (x, y).
* grid.setPathFinded(): indica que se ha encontrado el camino para que empiece a pintarse.

Además, este componente tiene funciones para poder pasar de coordenadas del mundo a coordenadas del mapa y viceversa. Estas funciones son necesarias para la solución de la práctica, y son las siguientes:

* gridToWorld(gridX, gridY): recibe las coordenadas x e y del mapa (gridX, gridY) y devuelve una tupla con las coordenadas equivalentes en el mundo (worldX, worldY).
* worldToGrid(worldX, worldY): recibe las coordenadas x e y del mundo (worldX, worldY) y devuelve una tupla con las coordenadas equivalentes en el mapa: (gridX, gridY).

Las funciones gridToWorld y worldToGrid han sido modificadas en la versión actual, puesto que estas funciones tenían un sistema de conversión apropiado para cómo está hecha la práctica, pero si modificáramos esta práctica sería difícil adaptar este sistema de conversión de coordenadas. Esto se debe a que emplea un sistema de conversión por casos, en vez de emplear matrices de rotación y traslación para pasar de un sistema de referencia a otro. Por eso se ha modificado empleando estas matrices. Además, uno de los motivos del cambio fue que no era correcta la conversión, sino que era aproximada. Lo que provocaba una desviación que afectaba al sistema de pilotaje. Si por ejemplo, teníamos una ruta pintada por el centro de la carretera en la GUI, al realizar el cambio a coordenadas del mundo nos daba como resultado una posición de Gazebo que implicaba que el coche iba pegado a las paredes, y por ello se chocaba con ellas. En las siguientes imágenes podremos ver la imagen del mapa con la antigua versión (a la izquierda) y la misma con la nueva versión (a la derecha). En este mapa se ha pintado un píxel en rojo en la posición (200, 200), que corresponde con el centro de la imagen, y con la posición donde se encuentra el taxi al lanzar la práctica.

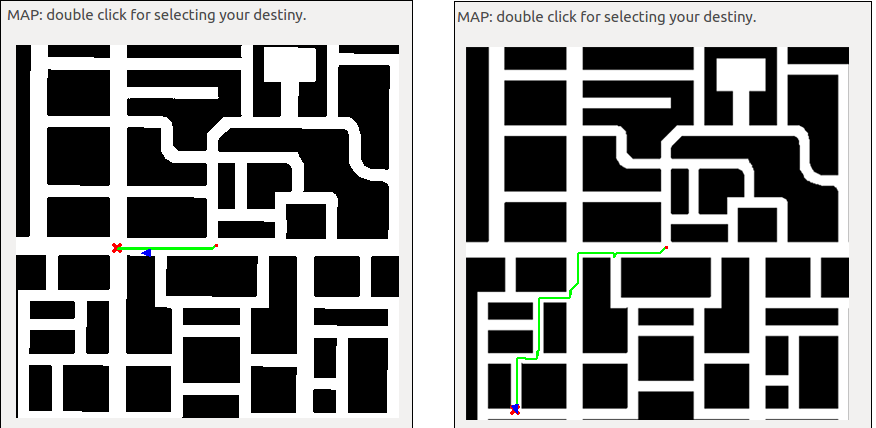


Figure 4.7: Imágenes con el centro incorrecto (izquierda) y el centro correcto (derecha)

Las matrices de rotación y traslación son necesarias puesto que queremos pasar de un sistema de referencia del mundo en 3D a un sistema de referencia del mapa en 2D o, al contrario. Estos cambios de sistema de referencia son necesarios porque en la práctica si realizamos la expansión del campo del gradiente sobre una rejilla deberemos saber la relación entre cada celdilla de la rejilla con el mundo para poder resolver la práctica. Lo primero que tenemos que conocer es cómo están situados los sistemas de referencia de cada sistema. Nuestros sistemas de referencia están situados de la siguiente forma:

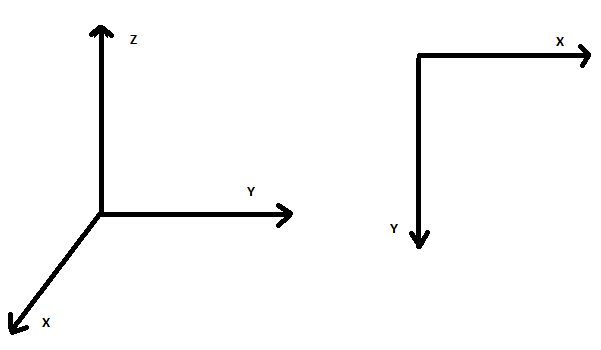


Figure 4.8: Sistema de referencia del mundo (izquierda) y sistema de referencia del mapa (derecha)

Para poder pasar de un sistema a otro deberemos aplicar la rotación y la traslación necesarias. Por ejemplo, en la función WorldGrid tenemos que pasar de una coordenada (x, y) en un mundo 3D a una coordenada (x’, y’) en 2D. Para ello primero deberemos aplicar una matriz de rotación de pi grados sobre el eje x para que el sistema de coordenadas del mundo rote de la siguiente forma:

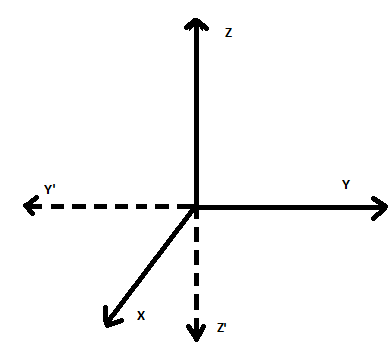


Figure 4.9: Rotación sobre el sistema de referencia del mundo (sistema punteado)

Por si tenemos que aplicar alguna traslación además de la rotación, tenemos una matriz de rotación y traslación para el eje x. Si no queremos realizar una traslación los puntos tx, ty, y tz tendrán valor 0. La traslación se debe a que queremos tener el punto (0,0) de la imagen en la esquina superior izquierda. El cambio lo realizamos de la siguiente forma:

Lo siguiente que tendremos que hacer es aplicar una rotación de –pi/2 grados sobre el eje z y una traslación si los puntos están un poco desviados. La traslación será de 200 píxeles (ancho de la imagen/2) en x y -200 píxeles (-alto de la imagen/2) en y. Por lo tanto, tx será en este caso 200, y ty será -200, tz será 0. La matriz de rotación y traslación la aplicamos de la siguiente forma:

Para poder realizar la conversión de sistemas de referencia para pasar de la imagen al mundo deberemos aplicar las matrices de rotación inversas a estas. Con esto obtendremos el resultado que deseábamos tener para poder cambiar de un sistema a otro cuando sea necesario.

* + 1. Interfaz gráfica de la práctica

La interfaz gráfica (GUI) de la práctica es un objeto de ayuda al alumno para poder realizar la solución a la práctica. Esta interfaz se realizará en PyQt5, dado que nos permite realizar interfaces con numerosos objetos gráficos (imágenes, botones, etc).

La GUI de la práctica contiene una imagen del mapa del mundo de Gazebo a la izquierda. En esta imagen que se muestra en el visor podemos seleccionar el destino al que deseamos que el taxi llegue. En ella se pinta la posición y orientación del taxi mediante un triángulo azul. El pintado del triángulo se ha modificado puesto que al haber una desviación en la conversión del sistema de referencia del mundo a la imagen o al revés, se había hecho una traslación a la hora de pintar el triángulo que no era necesaria. Por lo que esta conversión ha sido eliminada. Además, al cambiar de modelo de taxi se vio que la orientación del taxi no era correcta porque el ángulo no estaba en grados como era de esperar. El ángulo que nos devuelve la orientación del taxi está en radianes. Por lo que se suponía que se debía pasar a grados para poder realizar el pintado del triángulo, pero la conversión era errónea, ya que se multiplicaba por 100 la orientación para pasar a grados. Esto ha sido modificado obteniendo los grados al multiplicar la orientación por 180 y dividir por pi. A continuación, podemos ver dos imágenes, una con la orientación del triángulo incorrecta y otra con la orientación correcta.

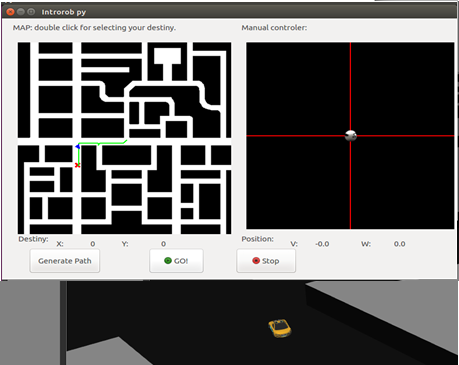


Figure 4.10: Interfaz gráfica con el triángulo que representa al taxi mal pintado

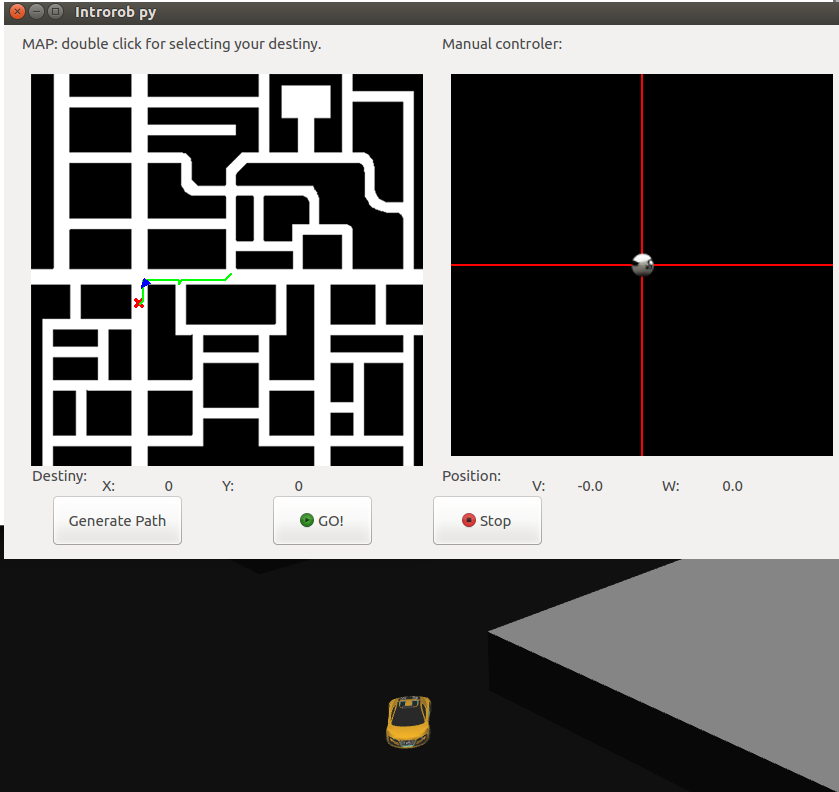


Figure 4.11: Interfaz gráfica con el triángulo que representa al taxi correctamente pintado

Esta interfaz gráfica además muestra un teleoperador a la derecha con el que se puede mover manualmente el taxi en el mundo de Gazebo si se desea. En la interfaz además se muestra debajo del mapa del mundo la posición que tiene el taxi al iniciarse la práctica en el eje X y en el eje Y. Por otra parte, debajo del teleoperador podemos ver la velocidad lineal y velocidad angular que tiene al teledirigir el taxi.

La GUI posee tres botones, de los cuales dos de ellos serán utilizados para poder ver cómo se lleva a cabo el algoritmo que ha programado el alumno. Cuando pulsamos el botón donde pone “Generate Path”, podemos ver el resultado del código que resuelve la parte de la planificación y cómo nos genera en la imagen del mapa una ruta. De ser pulsado el botón “GO!” se activará la parte de código que ejecuta el pilotaje del taxi, y podremos ver cómo nuestro taxi navega por las carreteras de la ciudad.

El tercer botón sirve para cuando estamos empleando el teleoperador. Este botón nos permitirá parar el taxi si queremos.

Se ha modificado el tamaño del visor de la imagen que muestra la GUI, ya que antiguamente no se podía ver la parte inferior del mapa en la GUI, por lo que existían puntos de la carretera a los que no podíamos ir. En las siguientes imágenes se puede comprobar la diferencia que acabamos de mencionar entre la antigua GUI y la actual:

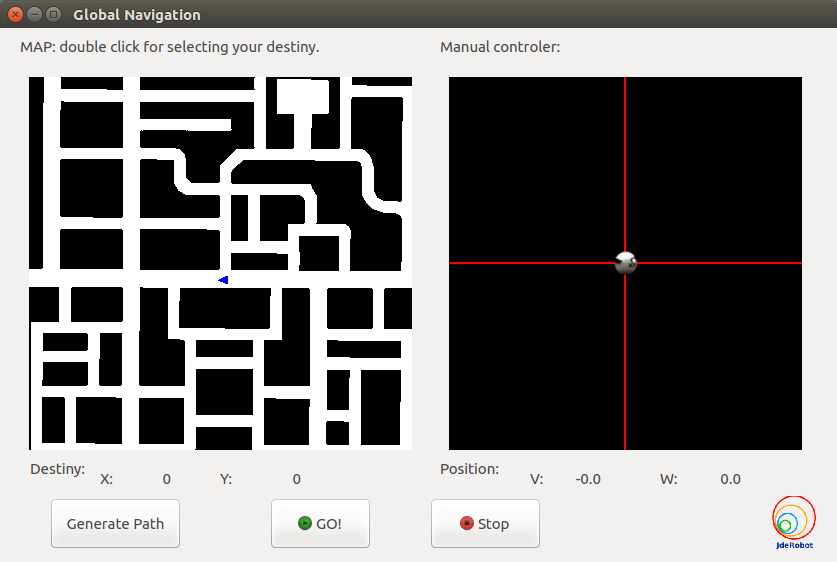


Figure 4.12: Interfaz gráfica (GUI) antigua del GPP

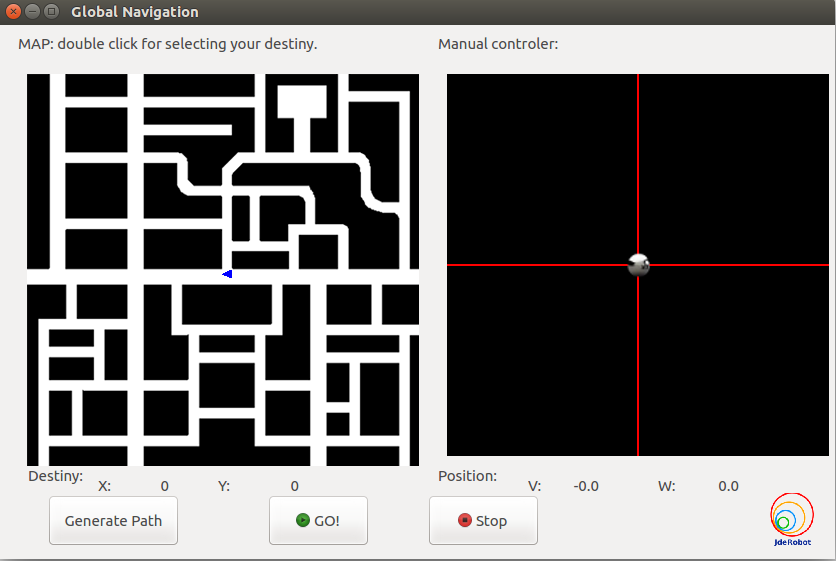


Figure 4.13: Interfaz gráfica (GUI) actual del GPP

* 1. Árbitro

La práctica consta de un árbitro que tiene en cuenta ciertos parámetros para calificar el algoritmo que programa el alumno. Este árbitro tiene una interfaz gráfica que muestra los diferentes parámetros, así como la nota final. Para poder crear el árbitro se ha empleado PyQt5. Para programar este árbitro se han creado clases diferentes para cada parámetro que queramos mostrar. Estas clases serán instanciadas en una clase principal llamada MainWindowReferee, que será la clase que contiene la ventana principal de la aplicación del árbitro.

El árbitro tiene en su visor un reloj que va mostrando los segundos que han pasado desde que se ejecuta la práctica. Este reloj tiene la apariencia de un reloj digital. Este visor de tiempo está programado en una clase (llamada timeWidget), en la cual se almacenará en una variable los segundos que el taxi está realizando el pilotaje.

En segundo lugar, se ha creado una clase (distanceWidget) para poder mostrar la distancia que existe entre la posición inicial del taxi y el destino que se ha marcado en la GUI. En todos los casos con esta clase se nos mostrará un mensaje en el visor de la aplicación. Si el destino no ha sido seleccionado todavía se nos mostrará el mensaje “Distance between robot and target: Destination not yet selected”. Por el contrario, si ya hemos escogido el destino deseado, nos mostrará el mensaje: “Distance between robot (origin) and target: X m”. Con X nos referimos a que dependiendo de donde esté colocado el destino se mostrará una distancia u otra.

En tercer lugar, tenemos un visor de la velocidad (en km/h) que alcanza nuestro taxi durante el pilotaje. Para mostrar la velocidad en km/h se ha calculado la velocidad en m/s y se ha realizado la conversión a km/h. En esta clase se realiza un pintado de un velocímetro, que marca con una aguja la velocidad que lleva nuestro vehículo. Cuando el coche está totalmente parado, la aguja aparecerá tumbada hacia la izquierda. Además, debajo del velocímetro aparecerá un mensaje con la velocidad en km/h que tiene nuestro taxi en cada momento.

Al igual que sucede con el velocímetro, tenemos un visor similar para pintar la orientación que tiene nuestro taxi en todo momento. Esta orientación irá desde un ángulo de –pi (izquierda) a un ángulo de pi (derecha). En esta clase (angleWidget), también, se mostrará un mensaje con la orientación del taxi.

Por último, tenemos la clase que nos calculará la nota final. En la interfaz gráfica podemos ver que tenemos un botón con el mensaje “Show me my mark”. Si pulsamos este botón nos mostrará nuestra nota. Si el botón se pulsa antes de que el taxi llegue al destino fijado nos indicará en un mensaje que el destino aún no se ha alcanzado. La nota final se calcula si hemos llegado al destino. Si hemos llegado a destino ya tendremos como mínimo una nota de 7, y como máximo una nota de 8, a la cual le sumaremos hasta 2 puntos como máximo en función de la velocidad media del taxi en el pilotaje. Si nos hemos quedado a una distancia máxima de 2 metros del destino tendremos un 8 de nota, a la que le sumaremos hasta 2 puntos como máximo. Por el contrario, si nos quedamos hasta 4 metros de distancia del objetivo el valor de la nota de la que partimos será 7.5; y si por el contrario nos hemos quedado hasta 5 metros del destino, la nota de partida será de 7 puntos. Si nos hemos quedado a más de 5 metros del destino se considerará que no hemos llegado aún. El cálculo del campo del gradiente no se tiene en cuenta en el cálculo de la nota, puesto que es muy difícil comprobar si se ha realizado el cálculo del campo correctamente, ya que hay diferentes modos de realizarlo. Para saber si el campo está bien habría que comprobar la imagen que tenemos como resultado del campo o emplear un vídeo de cómo se realiza la expansión.

A continuación, se muestran tres imágenes diferentes de nuestro árbitro. La primera imagen muestra cómo sería nuestro árbitro al ejecutar la práctica y no haber seleccionado aún el destino. En la segunda imagen, se muestra el aspecto que tendría el árbitro durante el pilotaje del taxi. Se puede observar que la aguja del velocímetro ya no está tumbada a la izquierda. En la última imagen podemos ver que se ha pulsado al botón “Show me my mark” y hemos alcanzado ya el destino, por lo que nos aparece un mensaje con nuestra nota.

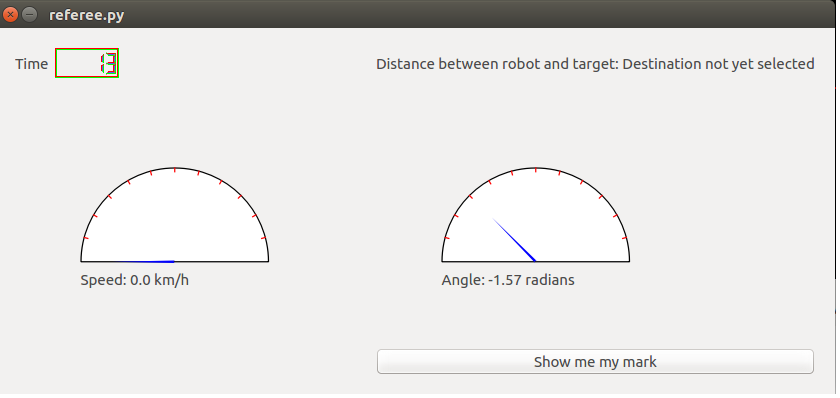


Figure 4.14: Árbitro del GPP antes de seleccionar un destino

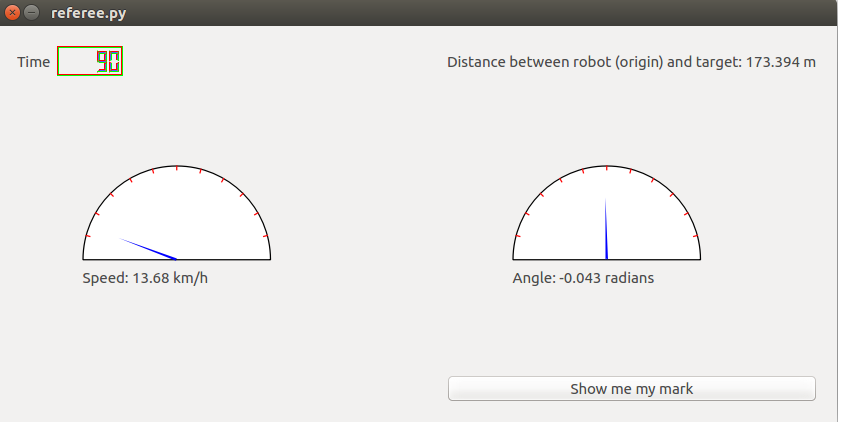


Figure 4.15: Árbitro del GPP durante el pilotaje

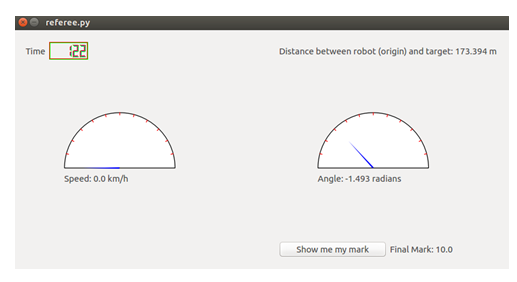


Figure 4.16: Árbitro del GPP cuando el taxi ha alcanzado el destino

* 1. Solución

Como hemos observado en el punto 4.1, el objetivo de esta práctica es proveer al robot de un algoritmo de navegación, que estará compuesto por un algoritmo de navegación global y un algoritmo de pilotaje. En este punto abordaremos una breve explicación sobre navegación global y local, una descripción de la técnica *Gradient Path Planning*, y la solución concreta de la práctica.

* + 1. Navegación global y local

El principal problema de los robots móviles es la navegación autónoma. La navegación autónoma es la capacidad que poseen los robots para ir desde un punto del espacio a otro cualquiera evitando chocarse con algún obstáculo, ya sean objetos fijos u objetos móviles inesperados. Esta es una tarea compleja, que provee a los robots de grandes capacidades. La navegación de los robots de forma habitual se divide en dos ramas: la navegación global y la navegación local.

La navegación global consiste en calcular o planificar una ruta de forma óptima inicialmente para que el robot la pueda seguir. Para poder llevar a cabo este proceso el robot debe tener un conocimiento preciso del escenario por el cual se debe mover. El robot suele tener previamente esta información mediante un mapa del entorno. De no ser así, el robot adquirirá dicho conocimiento construyendo un mapa del entorno por medio de los sensores que posee. La ruta se construye empleando técnicas de búsqueda o planificación, que requieren un tiempo considerable. Ejemplos de estas técnicas son: grafos de visibilidad y Gradient Path Planning.

La navegación local emplea las medidas obtenidas por los sensores del robot para que este sea capaz de evitar obstáculos que pudieran aparecer de improviso en cada instante. Este tipo de comportamiento dota al robot de una sensibilidad elevada en entornos altamente imprevisibles. Ejemplos de técnicas de navegación local son: Método de fuerzas virtuales (VFF), Método de carriles y velocidad (VLM) y Método de velocidad y curvatura (CVM).

* + 1. Fundamentos teóricos de Gradient Path Planning

Para solucionar el problema de la navegación global, hemos escogido el algoritmo *Gradient Path Planning* (GPP), que garantiza una trayectoria mínima entre el punto desde el que partimos hasta el punto de destino. La trayectoria calculada no se basa en la distancia euclídea mínima, sino que se incorporan los obstáculos en el cálculo de la trayectoria.

Para poder desarrollar el algoritmo *Gradient Path Planning* hemos partido del trabajo desarrollado por Kurt Konolige [14], así como de trabajos previos realizados en la Universidad Rey Juan Carlos [15], [16]. Esta técnica obtiene el camino óptimo desde el punto de partida hasta el destino.

La técnica GPP consiste en generar un frente de onda circular que parte desde la posición de destino, y que recorre el espacio libre del mapa hasta llegar a la posición del robot. El punto donde está situado el robot es el comienzo de la ruta que recorrerá el robot. En su propagación, el frente de onda asignará valores de forma creciente a cada punto libre del espacio por el que pase. Antes de comenzar la propagación del frente de onda, todos los puntos libres del espacio tienen valor 0. El frente de onda se puede expandir por todo el espacio, hasta la posición que ocupa el robot o un poco más allá de la posición del robot.

Los obstáculos generarán su propio frente de onda, lo que implica que los puntos del espacio que estén próximos a los obstáculos aumentarán su valor por defecto considerablemente. El frente de onda de los obstáculos se propaga de forma inversa al frente de onda anterior. Esto quiere decir que los puntos del espacio más próximos a los obstáculos tendrán un valor mayor a los puntos más alejados. El frente de onda de los obstáculos se expande hasta una distancia determinada, no se expanden por todo el espacio.

La expansión del frente de onda de los obstáculos evita que el robot se acerque a los obstáculos al navegar por el espacio. De no ser así, la ruta más corta estaría pegada a los obstáculos, y de esta forma el robot rozaría con las paredes de los obstáculos. Esto hará que el robot tenga mayor seguridad.

Una vez se ha generado el campo total, el robot podrá navegar hasta el destino evaluando en cada iteración los puntos de su alrededor. Dicho robot siempre se dirigirá hacia el punto de menor valor del campo calculado. A medida que avance el robot hacia su destino, el campo calculado tendrá menor valor, pues las zonas más próximas al destino son las que menor valor poseen.

El método GPP nos permite generar una ruta desde el punto de partida del robot hasta el destino deseado. Esta ruta se generará siguiendo el gradiente del campo calculado, y será la de menor distancia incorporando los obstáculos. Esta técnica de navegación global nos asegura que llegaremos al objetivo. Sin embargo, es posible que el robot durante el pilotaje no siga exactamente la ruta calculada, pues puede tener desviaciones debidas a la velocidad y rotación del robot. Por ello, el pilotaje se llevará a cabo de forma reactiva, mirando en cada momento cual es la celdilla de menor valor a la que debe dirigirse y si se encuentra próximo a algún obstáculo.

La forma reactiva se basa en el “ahora”, es decir, en cada instante de tiempo evalúa la situación y actúa. No desarrolla una solución inicial y la sigue, sino que va actuando en función de lo que ocurra en cada momento.

* + 1. Solución desarrollada

La navegación global mediante Gradient Path Planning se puede implementar de diversas formas. La solución implementada se describirá a continuación. Dicha solución se desarrolla en el fichero “MyAlgorithm.py”. En este fichero podremos observar que la solución se divide en un método en el que se construye el mapa del gradiente, y otro método que se corresponde con el pilotaje del robot.

* + - 1. Construcción del mapa del gradiente

En el método “generatePath” del fichero “MyAlgorithm.py” llevaremos a cabo el desarrollo del algoritmo que genera el mapa del campo del gradiente. Esta función se ejecutará solamente cuando pulsemos el botón “Generate Path” en la GUI. El escenario estará representado por una rejilla, donde iremos almacenando el campo calculado. Esta rejilla es proporcionada por la clase Grid.

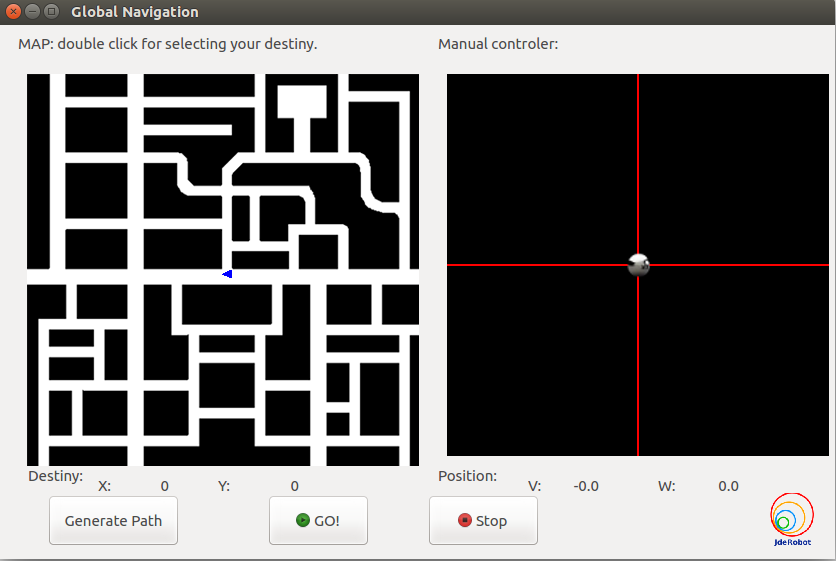


Figure 4.17: GUI del GPP

Lo primero que debemos conocer antes de comenzar a generar el campo y expandirlo por la rejilla es el mapa, la posición inicial del robot y la posición de destino deseado. En la práctica, se dispone de un objeto grid que nos permite obtener el mapa a través de la función grid.getMap(). Este mapa nos proporciona información del escenario mediante sus valores, donde el valor 0 representa a los obstáculos y el valor 255 representa la carretera. El destino podrá conocerse, una vez el usuario haya seleccionado el destino deseado en la GUI, mediante la función grid.getDestiny(). Por último, podemos obtener la posición del robot respecto al mapa mediante la función grid.getPose().

En la función “generatePath”, inicialmente, tendremos un bucle que realiza la propagación de los frentes de onda. Es decir, no tendremos un único frente de onda, sino que vamos a tener diferentes frentes de onda, los cuales se encuentran en una lista ordenada.

El bucle de la propagación de los frentes de onda se realizará hasta que se expanda el campo un poco más allá de la posición del taxi. En este caso se expandirá hasta 20 celdillas más allá de la celdilla que ocupa el taxi en la rejilla. Este número podría haber sido mayor o menor, pero en el caso de que fuera mayor la propagación tardaría más tiempo en ejecución. Si el número fuera menor tendríamos menos conocimiento de los alrededores del robot.

El punto de partida del frente de onda inicial es el destino, que es la celdilla de la rejilla que posee la distancia 0. Por lo tanto, el destino será nuestro primer nodo que expandirá el frente de onda a sus vecinos. Cada nodo expandirá el valor de la distancia a sus 8 celdillas vecinas. Este valor de distancia asignado a las celdillas vecinas será el valor del nodo (distancia) + 1 o el valor del nodo + 1.4. Cuando se encuentre un obstáculo, se almacenará la celdilla de dicho obstáculo en un array y no se le asignará ninguna distancia. De esta forma nos aseguramos de tener almacenadas las celdillas que pertenecen al borde de un obstáculo para poder posteriormente penalizar a las celdillas que se encuentren muy próximas a los obstáculos.

Cada nodo va a expandir antes el frente a sus vecinos que se encuentren a una distancia +1, y posteriormente expandirá el frente a sus vecinos que estén en una distancia + 1.4. Por lo que se podría decir que el frente de onda de cada nodo se divide en dos frentes de onda. De esta forma nos aseguramos que el frente de onda sea circular.

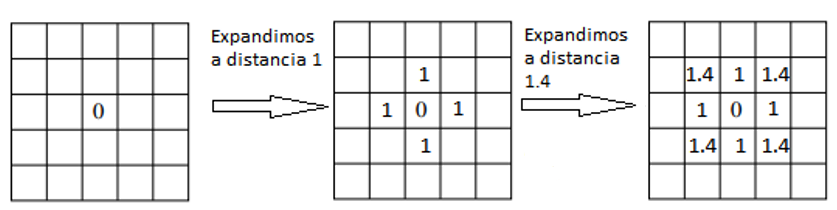


Figure 4.18: Primera propagación del frente de onda

Este proceso de expansión se irá haciendo sucesivamente hasta llegar a 20 celdillas más alejadas de la posición del taxi. A la hora de realizar la expansión si las celdillas vecinas tuvieran un valor mayor al que calculara el nodo, dicho valor se actualizaría por el valor que expanda el nodo actual. Esto permite asegurar que siempre tendremos el menor valor en cada celdilla.

Este proceso que acabamos de explicar se puede ver gráficamente a continuación:

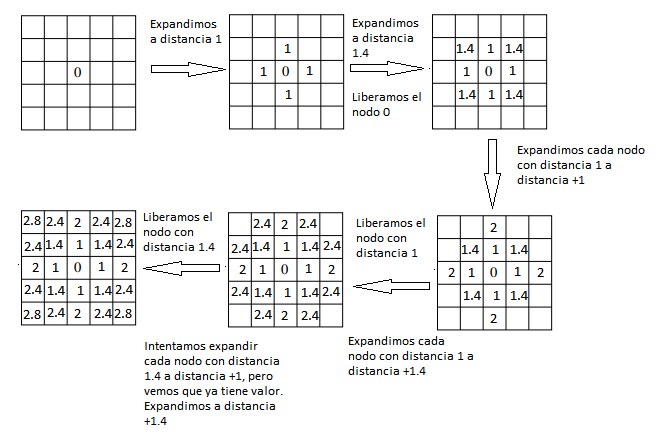


Figure 4.19: Esquema propagación frentes de onda

En la rejilla del campo que hemos generado se puede apreciar en color más oscuro los puntos más cercanos al destino, puesto que poseen un valor menor de distancia. Por el contrario, los puntos más lejanos del destino son los que poseerán un color más claro. A continuación, podemos observar una progresión de la expansión del campo.

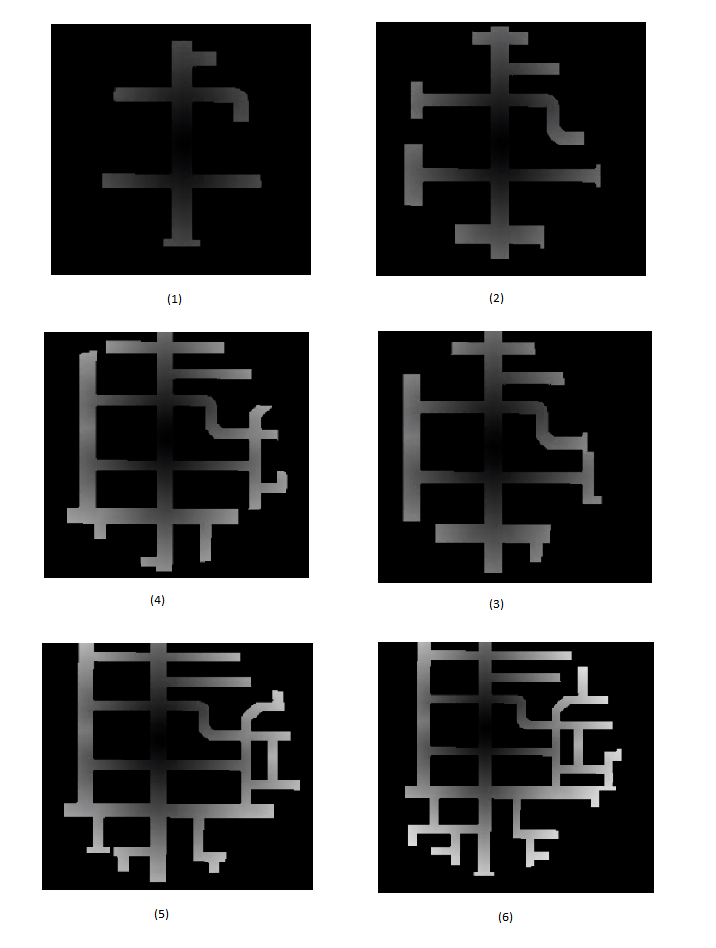


Figure 4.20: Esquema expansión del campo

El siguiente paso después de la propagación de los frentes de onda es la penalización que realizan los bordes de los obstáculos a las celdillas más próximas. Esta penalización se lleva a cabo para evitar que la ruta más corta desde el robot hasta el destino esté pegada a las paredes de los obstáculos y haga que nuestro taxi roce con las paredes. Con esta penalización nos aseguramos de que en el pilotaje halla un margen de seguridad entre el taxi y las paredes.

Como hemos mencionado antes, las celdillas que pertenecen a bordes de obstáculos están almacenadas en un array (llamado posObstaclesBorder). Estas celdas (obstáculos) sumarán una penalización a las celdillas vecinas que formen parte de la carretera en función de la distancia a la que se encuentran de la celdilla obstáculo.

Para poder llevar a cabo la penalización por obstáculos se ha creado una rejilla, la cual inicialmente posee un valor 0 en todas sus celdillas. En esta rejilla almacenaremos los valores de penalización de cada celdilla. Finalmente sumaremos la rejilla del campo y la de los obstáculos para actualizar sus valores con dichas penalizaciones.

La penalización que realizan los obstáculos se llevará a cabo mediante un bucle que recorre el array posObstaclesBorder. Cada posición de dicho array penalizará a las celdillas vecinas que están a una distancia de +1, +2 y +3 de la misma. Las celdillas que estén a una distancia de +1 del borde del obstáculo se penalizarán con un valor de 174. Las celdillas con una distancia +2 tendrán una penalización de 168, mientras que las celdillas con una distancia +3 se penalizarán con un valor de 162.

En cada penalización que adjuntemos a una celdilla antes comprobaremos su valor en la rejilla de penalización, puesto que dos celdillas del borde del obstáculo pueden querer penalizar a una misma celdilla, pero esta celdilla solamente se puede penalizar una vez. Cuando vayamos a penalizar y comprobemos si ha sido penalizada una celdilla, comprobaremos su valor. Si el valor de penalización que tiene dicha celdilla es menor que el que se iba a añadir, se sustituirá el valor de penalización por el mayor.

Estas penalizaciones harán que las celdillas próximas a los obstáculos tengan un valor mayor y aparezcan con un color más claro (blanco) al mostrar el grid. El grid lo podemos mostrar mediante la función grid.showGrid(). Esta función crea una ventana en la que representa los valores del campo que se le han asignado a la rejilla. Podemos ver a continuación una serie de ejemplos de diferentes campos calculados con la penalización por obstáculos incluida.



Figure 4.21: Representación campos calculados

El paso siguiente es el cálculo de la ruta más corta. Esta ruta se calcula para poder observar cual sería la ruta ideal que debe seguir nuestro taxi. Dicha ruta debería seguir las celdillas de menor valor de distancia.

El cálculo de la ruta más corta comienza en la celdilla donde se encuentra situado el taxi y termina al alcanzar la celdilla que posee el valor de distancia 0, es decir, el destino. Para poder ir almacenando la ruta tenemos que hacer uso de la función grid.setPathVal, la cual establece el valor en la posición que se le indica, tomando como ruta las celdillas que poseen un valor diferente a 0.

Comenzamos el cálculo de la ruta añadiendo la posición del taxi a la ruta. El siguiente paso es comprobar las celdillas vecinas de la celdilla que ocupa el taxi. Entre estos vecinos añadiremos a la ruta el de menor valor de distancia. Después, comprobaremos los vecinos de esta nueva celdilla y así continuamente hasta llegar a la celdilla destino.

Una vez que alcancemos la celdilla del destino tendremos que indicar que hemos terminado de calcular la ruta más corta y que se puede comenzar a pintar. Para ello empleamos la función grid.setPathFinded.

A continuación, podemos observar diferentes rutas calculadas en función del destino que hemos elegido. En las imágenes (1), (2) y (3), podemos ver las rutas calculadas aplicando la penalización de los obstáculos; mientras que en las imágenes (4), (5) y (6) vemos las rutas que se han calculado para los mismos destinos que en (1), (2) y (3), pero sin realizar la penalización de los obstáculos.

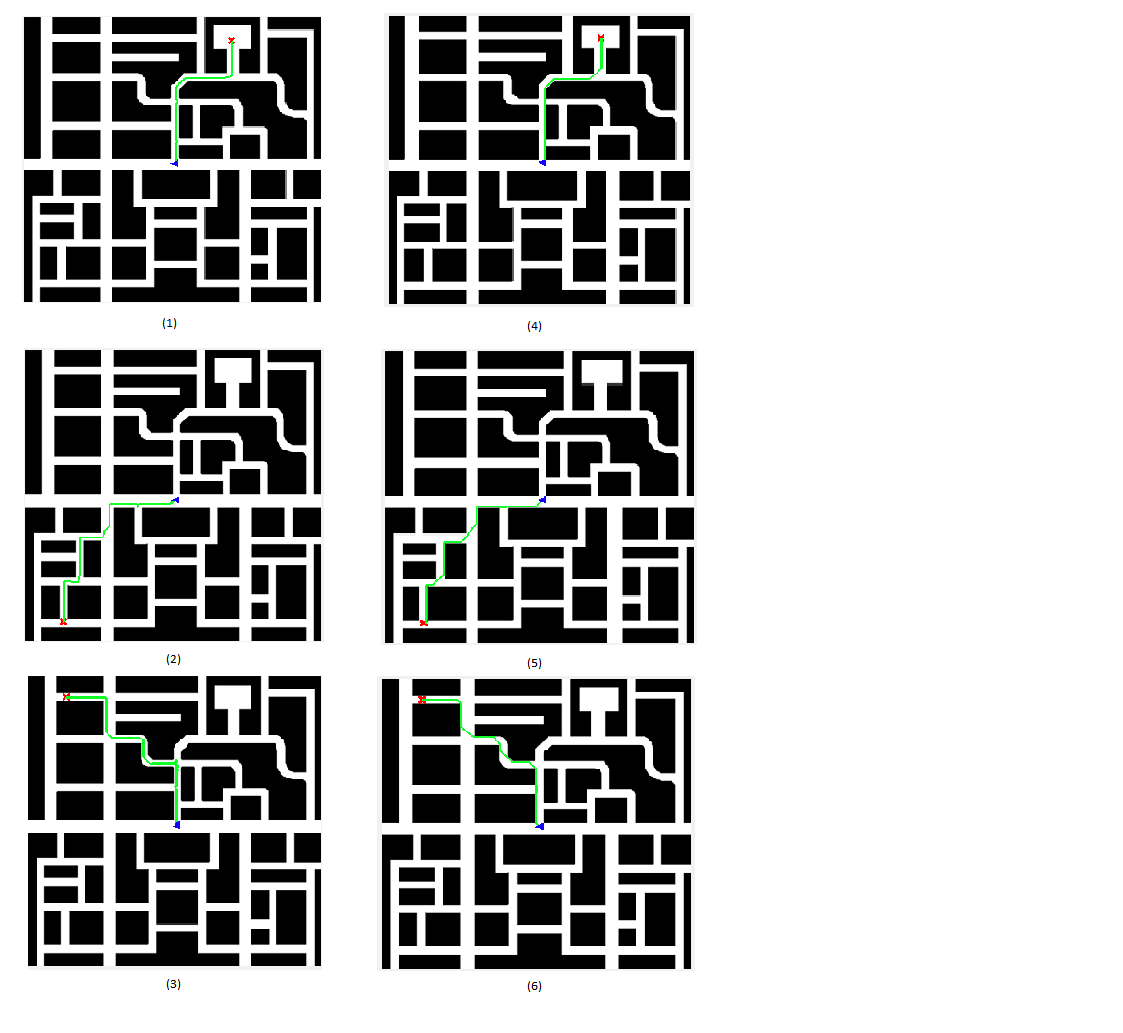


Figure 4.22: Esquema expansión del campo

* + - 1. Pilotaje del robot

En el método “execute” del fichero “MyAlgorithm.py” llevaremos a cabo el desarrollo del algoritmo correspondiente al pilotaje del taxi. Este método se ejecuta periódicamente para que el pilotaje sea un control reactivo. Es decir, este método implica que el taxi pueda mirar en cada instante cuál es su situación y así poder navegar por el escenario de manera adecuada.

En este método debemos incluir el algoritmo que se encarga de pilotar el robot desde su posición inicial hasta la posición del destino, mediante el campo calculado en el método generatePath. La dificultad está en la elección de la velocidad de tracción y la velocidad de rotación que debemos ordenar al taxi.

El pilotaje se ha realizado sin tener en cuenta la ruta más corta calculada en el punto anterior, ya que dicha ruta es el ideal a seguir, pero el taxi al seguir órdenes de velocidad de tracción y velocidad de rotación puede desviarse un poco de dicha ruta. Si intentara seguir la ruta ideal estrictamente, el taxi realizaría movimientos muy forzados hasta llegar al destino. Lo ideal es que el taxi se mueva de una forma suave, como lo haría un taxi real. En el pilotaje se ha tenido en cuenta el campo calculado. De esta forma, en cada iteración el taxi irá comprobando el valor de distancia de las celdillas que se encuentran en un radio de distancia con respecto a su posición. De estas celdillas elegirá como objetivo la celdilla de menor valor. Este planteamiento permite que el taxi tenga un comportamiento reactivo ante imprevistos y que se asemeje un poco a la navegación local, ya que no tiene en cuenta únicamente la ruta más corta calculada previamente, también tiene en cuenta la situación del taxi.

Inicialmente en el pilotaje debemos comprobar la pose de nuestro taxi mediante el sensor de posición y la posición del destino para ver cuál es la situación del taxi, ya que si el taxi ha alcanzado el destino debe detenerse en esta posición. Conocemos la celdilla que ocupa el destino en el grid, pero el sensor de posición nos devuelve la posición del taxi con respecto al mundo. Esto implica que debamos convertir las coordenadas del destino en el grid en coordenadas respecto al mundo para poder comprobar si hemos llegado a dicho destino. Para ello existe una función (grid.gridToWorld) que nos permite realizar la correspondencia de las coordenadas del grid con la posición que tendrían estas coordenadas en el mundo de Gazebo.

El siguiente paso es calcular el objetivo próximo al robot. Esto quiere decir que vamos a ir calculando en cada iteración un objetivo que se encuentra a cierto radio de distancia del robot. Esto requiere que vayamos comprobando los valores de distancia del campo que habíamos calculado en el punto anterior. Por lo que tendremos que obtener la posición del taxi en el sistema de coordenadas del grid (mediante la función worldToGrid). En nuestro caso comprobaremos las celdillas situadas a una distancia de 5 celdillas con respecto a la posición del robot y escogeremos la de menor valor como objetivo. Sin embargo, en este caso el objetivo no es exactamente el anterior mencionado, sino que vamos a calcular un segundo objetivo situado a 5 celdillas del primer objetivo, y posteriormente haremos la interpolación de estos dos objetivos obteniendo el objetivo final al que queremos llegar. El motivo de esta interpolación es obtener un pilotaje con movimientos más suaves. Además, esto nos permite que el taxi gire adecuadamente en las curvas.

El objetivo calculado está en coordenadas del grid, por lo que tendremos que hacer un cambio de coordenadas relativas al mundo para obtener el objetivo en coordenadas del mundo.

Una vez tengamos las coordenadas respecto a Gazebo del próximo objetivo, podremos calcular el vector de dirección que tendrá nuestro robot para llegar hasta este y el ángulo. Para ello debemos tener en cuenta la pose que nos devuelve el sensor de posición, así como la orientación del robot (en radianes), las cuales están en coordenadas absolutas del mundo.

Los robots tienen distintos sistemas de referencia, por lo que hay que tener en cuenta este dato a la hora de calcular su posición y orientación. Por un lado, tenemos el sistema de referencia absoluto, en el cual se pueden representar la posición del robot y la de cualquier otro objeto. Un sistema de referencia absoluto es aquel que tiene un punto de referencia fijo. El sistema de referencia absoluto en nuestro caso serán los ejes de Gazebo, por lo cual el punto de referencia será el punto de coordenadas (0, 0, 0) en el mundo de Gazebo. El otro sistema de referencia que vamos a tener en cuenta es el sistema de referencia solidario con el robot. Este sistema de referencia solidario con el robot se encontrará fijo con respecto al sistema de referencia absoluto inicialmente, pero según se mueva el taxi este sistema de referencia solidario con el robot se desplazará ya que es un sistema de referencia respecto a la posición del robot. También hay que mencionar que inicialmente el robot comienza con una rotación de –pi/2 radianes.

En la siguiente imagen podemos ver el sistema de referencia absoluto (son los ejes azul, verde y rojo que atraviesan la imagen) y el sistema de referencia solidario con el robot (flechas azul, roja y verde que salen del robot):

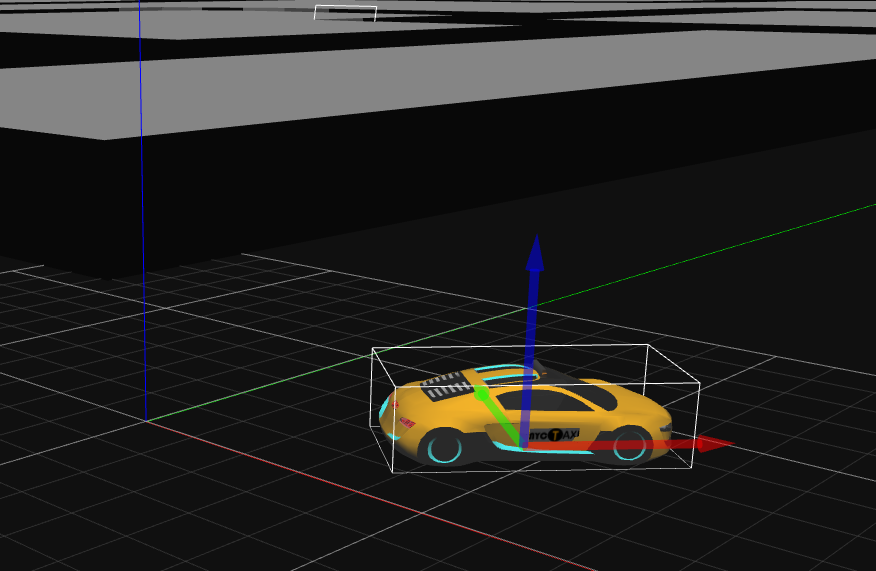


Figure 4.23: Sistema de referencia absoluto (Gazebo) y sistema de referencia solidario con el robot

En el caso de esta práctica se ha decidido definir el sistema de referencia solidario con el robot de la siguiente forma: el eje X de este sistema es el que señala al frente del robot, mientras que el eje Y es el que señala a la izquierda del robot. El origen de este sistema es un punto situado en el centro geométrico del robot.

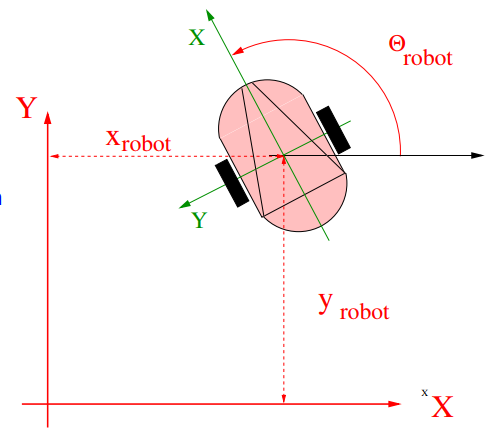


Figure 4.24: Sistema de referencia absoluto y sistema de referencia solidario con el robot

El sistema de referencia solidario con el robot se emplea para expresar las coordenadas relativas de los objetivos próximos o de obstáculos respecto al robot. Esto es necesario puesto que si tenemos un punto P que no se mueve en el espacio para el sistema de referencia absoluto no habrá movimiento, pero para el sistema de referencia solidario con el robot las coordenadas de este punto P varían.

Sabiendo las coordenadas absolutas de un punto, y las coordenadas absolutas del robot y su orientación, podemos pasar las coordenadas absolutas a relativas o al revés. En nuestro caso deberemos pasar las coordenadas absolutas de cada objetivo próximo a coordenadas relativas al sistema del robot para poder calcular la velocidad de tracción y rotación que debe tener el taxi.

Con la transformación de coordenadas absolutas a relativas obtenemos un vector de dirección, con el que podremos calcular el ángulo que hay entre el origen del sistema relativo al robot y la posición relativa del objetivo. Para obtener el vector de dirección hemos creado una función, en la que primero calcularemos la diferencia entre las coordenadas absolutas del objetivo y las coordenadas absolutas del robot (dx, dy), y después a esta diferencia le aplicamos una matriz de rotación para obtener el vector de dirección. En esta matriz de rotación tendremos en cuenta la orientación (θ) del robot.

x ' = dx cosθ − dy sinθ

y ' = dx sinθ + dy cosθ

Con este vector de dirección conocemos donde se encuentra situado el punto objetivo con respecto a nuestro origen del sistema del robot, ahora podremos calcular el ángulo que debe rotar el robot para alinearse con este punto. Este ángulo lo calcularemos mediante el cálculo del arco tangente. Conocemos la fórmula de la tangente:

Podemos ver a continuación un dibujo del ángulo que queremos calcular y el punto P (objetivo). Conociendo estos datos podemos calcular con la arco tangente.

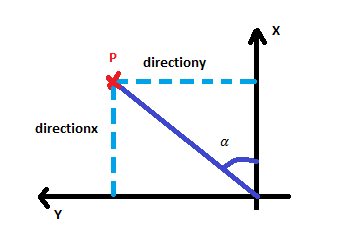


Figure 4.25: Sistema de referencia solidario con el robot y punto objetivo

El ángulo lo podemos calcular de la siguiente forma:

El ángulo calculado está en radianes. Si este ángulo es muy grande significará que debemos dotar al taxi de una velocidad de rotación alta. Por el contrario, si este ángulo es muy pequeño significa que el robot se encuentra alineado con el objetivo y que probablemente la velocidad de rotación de nuestro vehículo sea 0.

En esta solución se ha realizado un control por casos en función del ángulo calculado. Esto quiere decir que en función de este ángulo dotaremos al taxi de mayor o menor velocidad de tracción y de rotación. Si el ángulo calculado es muy elevado aplicaremos una velocidad de tracción reducida y una velocidad de rotación elevada (el coche puede estar en una curva o necesitar realizar un gran giro). Sin embargo, si el ángulo es pequeño, le daremos al taxi una velocidad de tracción elevada (ya que se encuentra en una recta) y poca velocidad de rotación.

A continuación, podemos observar una secuencia de imágenes en la que se ha calculado una ruta y podemos ver como aproximadamente el taxi sigue esta ruta, pero a veces se desvía un poco como habíamos mencionado anteriormente que podría suceder. Pero se puede ver que el taxi alcanza el destino deseado con éxito.

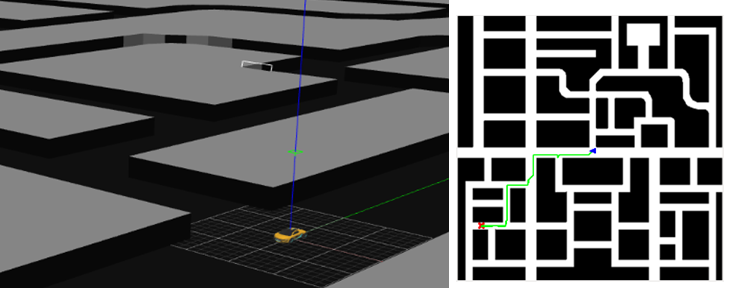


Figure 4.26: Posición 1 taxi en el mundo de Gazebo y en la GUI

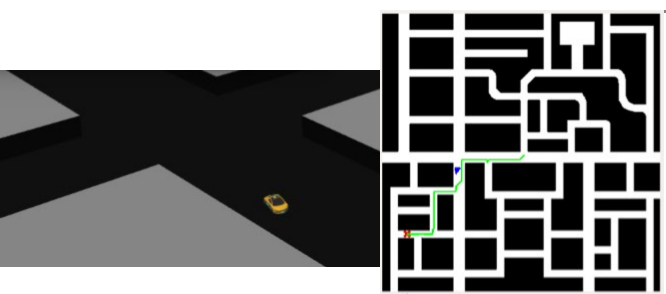


Figure 4.27: Posición 2 taxi en el mundo de Gazebo y en la GUI

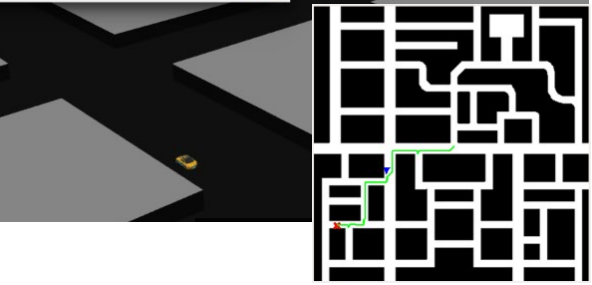


Figure 4.28: Posición 3 taxi en el mundo de Gazebo y en la GUI

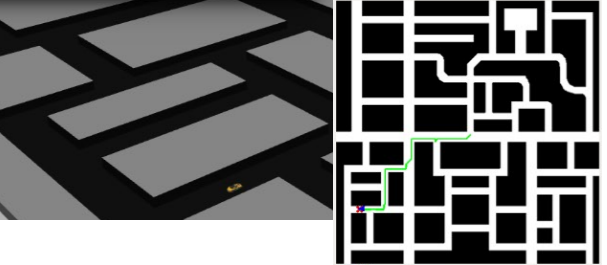


Figure 4.29: Posición 4 (destino) taxi en el mundo de Gazebo y en la GUI

* + 1. Experimentación

En la práctica es muy importante el tiempo de ejecución, ya que este tiempo influye en la nota de la práctica y cuanto más rápido sea el algoritmo mejor. Sí que es verdad que en esta práctica tenemos una ciudad muy grande y esto influirá en el tiempo, pero tenemos que intentar que sea el menor tiempo posible.

En el tiempo que tarda nuestro taxi en llegar al destino, es decir, durante el pilotaje, influirá el ordenador que empleemos. Este es un gran inconveniente, ya que quien posea mejor ordenador obtendrá tiempos de ejecución mucho menores que quien posea un ordenador sin tantas capacidades. La ejecución de Gazebo consume muchos recursos del ordenador haciendo que el taxi sea más lento.

En la parte inferior de Gazebo se puede ver el Real Time, el Sim. Time (tiempo simulado) y el Real Time Factor, los cuales tienen mucho que ver en el tiempo de ejecución de Gazebo. El parámetro Real Time expresa el tiempo real en ejecución. El factor Sim. Time expresa el tiempo simulado. Si utilizáramos un ordenador potente entonces el Sim. Time debería estar próximo al Real Time. Mientras que si usamos un ordenador sin tantas capacidades veremos que el Sim. Time es mucho menor que el Real Time. Por su parte, el factor Real Time Factor es un producto de la tasa de actualización y el tamaño del paso. Si queremos obtener un tiempo de simulación bajo deberá oscilar alrededor de 1. Si este parámetro es menor que 1 veremos que la ejecución es más lenta, y cuando se aproxima a 0.2 o menos es demasiado lenta.

En el caso del ordenador que se ha empleado el Real Time Factor es muy bajo en algunas ocasiones durante el pilotaje, lo que hace que el Real Time sea mucho mayor que el Sim. Time. En este ordenador el Real Time Factor normalmente oscila entre 0.1 y 0.75, siendo en grandes ocasiones cercano a 0.1.

En el tiempo de ejecución de la práctica se puede diferenciar el tiempo de planificación y el tiempo de pilotaje. Dependiendo del destino que elijamos ambos tiempos variarán, siendo menor si escogemos un destino cercano. Se han realizado varias pruebas con diferentes destinos para poder ver la diferencia entre el tiempo de ejecución.

* Primer destino. Hemos elegido un destino bastante alejado de la posición inicial del robot, lo cual se puede observar en la imagen de abajo. Al realizar la prueba, se ha podido apreciar que el tiempo de planificación es de tan solo 16’’. El tiempo de pilotaje es bastante mayor, alrededor de 2’ 30’’. Para tener conciencia de cómo influye el ordenador en la ejecución de la práctica, se ha comprobado el tiempo Real Time y Sim. Time (se inicializan nada más ejecutar la práctica, aunque aún no se haya comenzado a ejecutar el algoritmo) cuando el taxi alcanza el objetivo. El resultado obtenido es que el Real time es de 3’ 26’’; mientras que el parámetro Sim. Time adquiere un valor de 1’ 30’’. Podemos ver que la diferencia es excesivamente grande, casi de 2 minutos, lo que implica que en un ordenador con las capacidades al máximo el tiempo de ejecución sería relativamente corto.

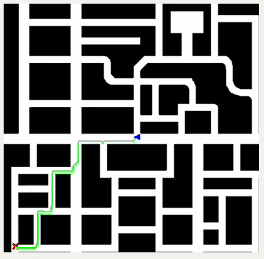


Figure 4.30: Imagen del mapa con el primer destino elegido

* Segundo destino: En esta ocasión se ha elegido un punto en la plaza, que se podrá ver en la imagen inferior. Al comprobar el tiempo de planificación se ha obtenido un tiempo de 16’, al igual que en el caso anterior. El tiempo de pilotaje es de 1’ 57’’. En esta ocasión el tiempo de pilotaje ha sido de aproximadamente 30’’ más rápido. Si comprobamos el Real Time vemos que es de 3’ 06’’; mientras que el Sim. Time es de 1’ 35’’. Es decir, en esta ocasión la diferencia entre el tiempo simulado y el tiempo real es menor, lo que nos lleva a obtener un menor tiempo de ejecución.

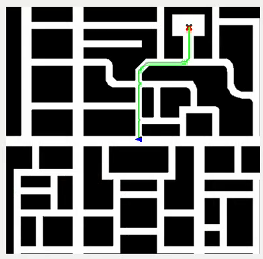


Figure 4.31: Imagen del mapa con el segundo destino elegido

* Tercer destino: El punto escogido en este caso lo podemos ver en la imagen inferior. En este caso el tiempo de la planificación ha sido de 19’, algo superior a las ocasiones anteriores. El tiempo que tarda el taxi en llevar a cabo el pilotaje es de 1’ 12’’, inferior que en el resto de ocasiones puesto que es un destino más cercano. En esta ocasión el Real Time ha sido de 2’ 19’’; y el Sim. Time ha sido de 1’ 17’’. La diferencia ha sido menor que en las anteriores ocasiones.

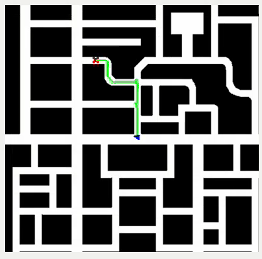


Figure 4.32: Imagen del mapa con el segundo destino elegido

Hemos podido ver cómo influyen diferentes aspectos en el tiempo de simulación: la lejanía del destino y las capacidades del ordenador que empleemos. Además, hemos podido observar que el tiempo de planificación suele ser bastante corto, esto es debido a que el algoritmo es rápido. El pilotaje por su parte ha sido más lento debido a distintos aspectos. Pero es importante mencionar que sería posible mejorar el tiempo de ejecución, empleando quizás algún otro algoritmo más rápido o mejorando el existente. Además, se podría mejorar el tiempo de ejecución dotando al taxi de una mayor velocidad.

Bibliografía

<https://gsyc.urjc.es/jmvega/documentation/2008-finalProject-guideRobot-talk.pdf>

<http://webpersonal.uma.es/~VFMM/PDF/cap2.pdf>

<https://gsyc.urjc.es/jmplaza/papers/waf2005.pdf>

<http://jderobot.org/store/jmplaza/uploads/students/pfc-gradientplanning2005.pdf>

<http://persoal.citius.usc.es/adrian.gonzalez/pdf/Gonzalez-Sieira11_thesis.pdf>

<http://jderobot.org/multimedia/pfc/jmvega/grad/documentation/guide_robot_2008.pdf>

<https://gsyc.urjc.es/jmplaza/pfcs/pfc-polly2005.pdf>

<https://cuentos-cuanticos.com/2011/11/12/navegacion-autonoma/>

<http://dspace.uah.es/dspace/bitstream/handle/10017/8032/Tesis_Doctoral_Mariano_G%C3%B3mez_Plaza_Nov-09.pdf?sequence=1>

<https://ai2-s2-pdfs.s3.amazonaws.com/dd24/af190a9235f0e2092e441354b87d5d93ae99.pdf>

<https://gsyc.urjc.es/jmplaza/activities/charla-umalaga2009.pdf>

<https://es.slideshare.net/EducaredColombia/tipos-de-movimiento-y-grados-de-libertad>

<http://www.lejosconlego.com/2013/02/lego-nxt-holonomico.html>

<http://www.arqhys.com/contenidos/quees-odometria.html>

<http://sdformat.org/>

<https://eva.fing.edu.uy/pluginfile.php/127423/mod_resource/content/1/simuladores.pdf>

[14] Kurt Konolige. A gradient method for realtime robot control. IROS, 2000.

[15] José Raúl Isado. Navegación global por el método del gradiente. Proyecto fin de carrera Ing. Informática, Universidad Rey Juan Carlos, 2005.

[16] Julio Manuel Vega Pérez. Navegación y autolocalización de un robot guía de visitantes. Proyecto fin de carrera Ing. Informática, Universidad Rey Juan Carlos, 2008.