UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIEROS INDUSTRIALES VALENCIA

TRABAJO FIN DE GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

DESARROLLO DE UNA APLICACIÓN PARA EL GUIADO AUTOMÁTICO DE UN VEHÍCULO ELÉCTRICO

AUTORA: CLAUDIA ANAÏS GONZÁLEZ MORENO

TUTORA: MARINA VALLÉS MIQUEL

COTUTOR: ÁNGEL VALERA FERNÁNDEZ

Curso Académico: 2015-16

Agradecimientos

“A mi familia,
por permanecer siempre
a mi lado”

Resumen

En este proyecto se ha realizado la implementación de un algoritmo de generación de trayectoria enfocado a maniobras de estacionamiento de vehículos de tipo coche. En primer lugar se hace una breve introducción teórica a la robótica, las diferentes configuraciones de robots y se entra a explicar con más detalle el *software* utilizado para la realización del mismo. Primero se hará un repaso de los elementos más generales de *ROS*, para ya entrar, posteriormente a los paquetes y funciones más específicos que nos permiten implementar la aplicación. También, en esta parte del documento se adjunta una descripción de los modelos matemáticos que se han utilizado para la generación de las trayectorias.

Tras este desarrollo teórico, pasamos a comentar lo que se ha llevado a cabo a nivel práctico con la realización de este proyecto. Aquí, en primer lugar se hace una presentación del material que se ha utilizado. Después, se realiza una implementación y simulación de cómo resultarían estas curvas enfocadas al estacionamiento con el *software* matemático *Matlab*. Con este mismo *software* generamos unos ficheros de puntos que podrán ser simulados con programas que nos ofrece el entorno de *ROS* (*gazebo* y *rviz)*,y una vez visualizados los resultados de esta, se implementa en *ROS*. En este documento se adjunta toda la información necesaria para hacer las simulaciones e implementaciones citadas anteriormente, además de una presentación de los resultados.

También se incluye, para cerrar el documento, unas conclusiones, bibliografía del material consultado y unos anexos, que incluyen información referente a instalación de *software* utilizado, el programa desarrollado e instrucciones para accionar el *rbcar*.

Por último, se adjunta un segundo documento que incluye el presupuesto del proyecto realizado.

Palabras clave: rbcar, ROS, generación de trayectorias, puntos, robot móvil, simulación, estacionamiento, en línea, en paralelo, paquetes, nodos, programación.

Resum

En aquest projecte s'ha realitzat la implementació d'un algoritme de generació de trajectories enfocat a maniobres d'estacionament de vehicles de tipus cotxe. En primer lloc es fa una breu introducció a la robòtica, les diferents configuracions de robots i s'entra a explicar en més detall el *software* utilitzat per a la realització d'aquest treball. Primerament, es farà un repàs dels elements més generals de *ROS*, per a entrar, posteriorment als paquets i funcions més específics que ens permeten implementar l'aplicació. També, en aquesta part del docment s'adjunta una descripció dels models matemàtics que s'han utilitzat per a la generació de les trajectories.

Després d'aquest desenvolupament teòric, passem a comentar el que s'ha dut a terme a nivell pràctic en la realització del projecte. Ací, primerament es fa una presentació del material que s'ha emprat. Posteriorment, es realitza una implementació i simulació de com resultarien estes curves enfocades a l'estacionament en el *software* matemàtic *Matlab*. Amb aquest mateix programa generem uns fitxers de punts que podran ser simulats amb programes que venen oferits per l'entorn de *ROS* (*Gazebo* y *Rviz)*, y una vegada visualitzats els resultats d'aquesta, s'implementa en *ROS*. En aquest document s'adjunta tota la informació necessària per a fer les simulacions i implementacions citades anteriorment, a més d'una presentació dels resultats.

També s'inclou, per a tancar el document, unes conclusions, bibliografia del material consultat i uns anexes, que inclouen informació referent a instalació del *software* utilitzat, el programa desenvolupat i instruccions per a accionar el *rbcar.*

Com a última cosa, s'adjunta un segon document que inclou el pressupost del projecte realitzat.

Paraules clau: rbcar, ROS, generació de trajectories, punts, robot mòbil, simulació, estacionamient, en linia, en paral·lel, paquets, nodes, programació.

Abstract

In this project we have carried out the implementation of a trajectory generation algorithm which is focused on parking manoeuvres of a car like vehicle. First, we make a short theoretical introduction about robotics, the different robot types we can find, and then we focus on explaining in more detail the software that has been used for this project's development. First of all we review the most general elements of *ROS*, to, after that, talk about it's most important packages and features which are used to perform the functionality we look for. Also, in this part of the document it is attached a description of the mathematical models that have been used for the trajectory generation.
After this theoretical part, we proceed to explain what has been carried out on a practical level. Here, first of all, we do an introduction about the materials that have been used through the whole development of the project. After that, using the mathematical software Matlab, we implement and simulate the curves we want to use for this application. Then, and using Matlab as well, we create some waypoints files which will be simulated with programs that are integrated in *ROS*, and when we have done all these tests and we have all the results, we generate a program that is compatible with *ROS*. In this document we have included all the information required to do all the simulations and implementations described above, as well as, a presentation of the final results of this project.
It's also included, to close this document, the conclusion, a bibliographical reference of all the material consulted for this document and an attached documents which includes information regarding the *ROS* installation, our node's complete code and instructions on how start the *rbcar.*
Finally, a second document which includes an economical estimate of the cost of the project is attached.

Key words: rbcar, ROS, trajectory generation, waypoints, mobile robot, simulation, parking, linear parking, parallel parking, packages, nodes, programming.

Documentos incluidos en este Trabajo de Fin de Grado.

- Memoria
- Presupuesto

Índice de la memoria

Índice de figuras

Índice del presupuesto

Memoria

Capítulo 1.Introducción y objetivos

1.1. Introducción y motivación

Vivimos en un mundo en el que la tecnología cada vez está más presente, como consecuencia a esto, los robots poco a poco se va haciendo paso en nuestras vidas para hacérnosla mucho más sencilla. Con el fin de acelerar este proceso de automatización de tareas sencillas y cotidianas nace la idea de este proyecto. No es raro ver que los coches que vemos todos los días, o aquellos que se ven en los anuncios publicitarios de la televisión ya son capaces de realizar ciertas maniobras de forma automática y autónoma. De esta forma empiezan a surgir los problemas de cómo generar este tipo de trayectorias, que para los humanos son tan sencillas de realizar en función de la situación en la que estemos, mediante algoritmos y fórmulas matemáticas que son el lenguaje que pueden entender nuestros ordenadores y robots.

Así pues, este trabajo busca, a partir de modelos matemáticos, ser capaz de generar una serie de trayectorias que resulten en la completa automatización de las maniobras que han de realizarse para estacionar nuestro vehículo en un hueco, ya sea en línea o en paralelo.

1.2 Objetivos y consideraciones

El objetivo principal de este proyecto es generar una trayectoria a partir de una serie de modelos matemáticos con el fin de realizar una serie de maniobras que permitan el estacionamiento del vehículo. Se ha enfocado de forma que se trabaja con unas condiciones de trabajo muy concretas dentro del marco de posibilidades que podemos encontrar a la hora de estacionar un vehículo.

Para poder alcanzar este objetivo global se han establecido una serie de objetivos menores que una vez han sido cumplidos todos, se podrá considerar que se ha alcanzado aquello que se esperaba de este proyecto. Estos objetivos menores son los siguientes:

- El completo entendimiento y familiarización con el entorno de *Ubuntu*, puesto que se utilizaba este sistema operativo por primera vez, se espera que para la finalización de este proyecto se comprenda su funcionamiento y las ventajas, que para un proyecto de estas características, nos ofrece respecto a otros sistemas.
- Familiarización con el entorno de trabajo de *ROS*. Este *software*, que también se utilizaba por primera vez, representa uno de los grandes desafíos de este proyecto, puesto que la forma de utilización y programación en el mismo suponen una gran diferencia respecto a los métodos tradicionales.

- Completo conocimiento del funcionamiento del sistema de navegación que se encuentra integrado en el proyecto, con el fin de poder desarrollar correctamente la trayectoria que buscamos describir.
- El desarrollo de un algoritmo que nos permita generar los puntos que posteriormente el coche habrá de seguir gracias al ya nombrado sistema de navegación que tenemos en él. Se busca que este algoritmo pueda ser implementado en *ROS*, para poder así ser utilizado sin ningún tipo de problema en nuestro vehículo.
- La integración del algoritmo expuesto en el punto anterior con el sistema de navegación del vehículo.

También, es importante considerar que para el proyecto haremos los siguientes supuestos:

- Se supondrá que se conoce el punto final en el que acabará el coche.
- El hueco en el que estacionaremos el vehículo es lo suficientemente grande para el mismo.
- El hueco, para el caso en paralelo, ha sido detectado (ya sea de forma visual por la persona que se encuentra dentro del vehículo o vía sensor) a una distancia del mismo.
- El hueco, para el caso en línea, se encuentra en las proximidades de nuestro vehículo y este está situado de forma correcta para realizar la maniobra (esto es, delante del hueco a una distancia en horizontal de un coche).

Capítulo 2. Desarrollo teórico.

2.1 Robótica

2.1.1 Definición y clasificación general.

En primer lugar, definimos el concepto robot. Un robot es, según la RAE: *"Máquina o ingenio electrónico programable, capaz de manipular objetos y realizar operaciones antes reservadas solo a las personas"*[1] Este concepto cabe ampliarlo más, puesto que tenemos una gran cantidad de robots hoy en día y en el alcance de este proyecto se trabajará con uno muy concreto.

En primer lugar podemos diferenciar entre las distintas clases de robots:

- Robots estáticos, por ejemplo brazos robóticos en las industrias, son robots que no pueden desplazarse en el espacio, pero que poseen articulaciones y mecanismos que permiten cierto tipo de movimiento.
- Robots móviles son aquellos robots que cuentan con un sistema de tracción que les permite moverse en el espacio.
- Los robots humanoides o zoomórficos, que se tratan de máquinas que buscan imitar los movimientos y la apariencia de seres humanos y animales respectivamente. Estos pueden ser móviles o estáticos[2]

Figura 2.1[30] Robot móvil

Figura 2.2[31] Robot estático

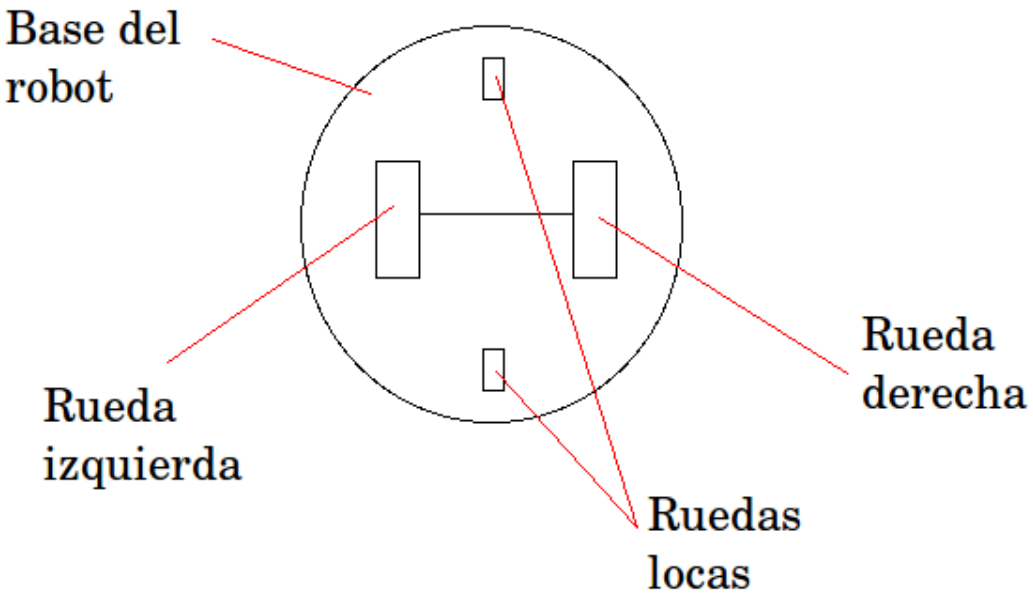
Figura 2.3[32]Robot humanoide

2.1.2 Clasificación robots móviles

Ya que lo que tenemos es un robot móvil, es necesario entrar a explicar los diferentes tipos de configuración, haciendo más hincapié en la que nos ocupa.

En primer lugar tenemos la configuración diferencial. Ésta es considerada la más sencilla de todas, consta de dos ruedas motrices independientes puestas a lo largo de un eje perpendicular a la dirección del motor. Cada una de estas ruedas será solidaria a un motor y podrán establecerse velocidades distintas en ambas para poder establecer trayectorias con giros (dándole más velocidad a la rueda derecha para girar a la izquierda y viceversa).

Como mantener el equilibrio del robot con solo dos ruedas nos resulta muy complicado a nivel de programación, en este tipo de configuración se añade, como mínimo, una tercera rueda, llamada “rueda loca” que no está solidaria a ningún eje, sino que gira libremente según la velocidad del robot, lo que nos permite mantener el equilibro y realizar giros al mismo tiempo.[3][4]



[1]

Figura 2.4[33] Robot configuración diferencial

Como ejemplo de caso concreto de la configuración anterior, tenemos la configuración en oruga. Este consiste en sustituir las ruedas por orugas longitudinales que tienen una mayor superficie de apoyo que las ruedas y esto nos permite prescindir de la “rueda loca”.
La oruga nos permite tener una mayor tracción y un mejor sistema anti-derrape, pero nos dificulta el cálculo odométrico.[3][4]

Figura 2.5 [34] Robot configuración oruga

Entre las configuraciones más comunes encontramos también la configuración en triciclo. En este caso volvemos a tener 3 ruedas, una delantera y dos traseras, pero en este caso las traseras no llevan ningún tipo de tracción, sino que es la delantera la que nos tiene que indicar la dirección (con su ángulo de inclinación) y a la que se acopla el motor.
Esto nos facilita el cálculo de la odometría, pero también nos dificulta la gestión de los giros, ya que depende de la distancia que exista entre las ruedas traseras y la delantera, y tener en cuenta que estas ruedas traseras deben ser capaces de girar de forma independiente para compensar el trayecto a recorrer. [3][4]

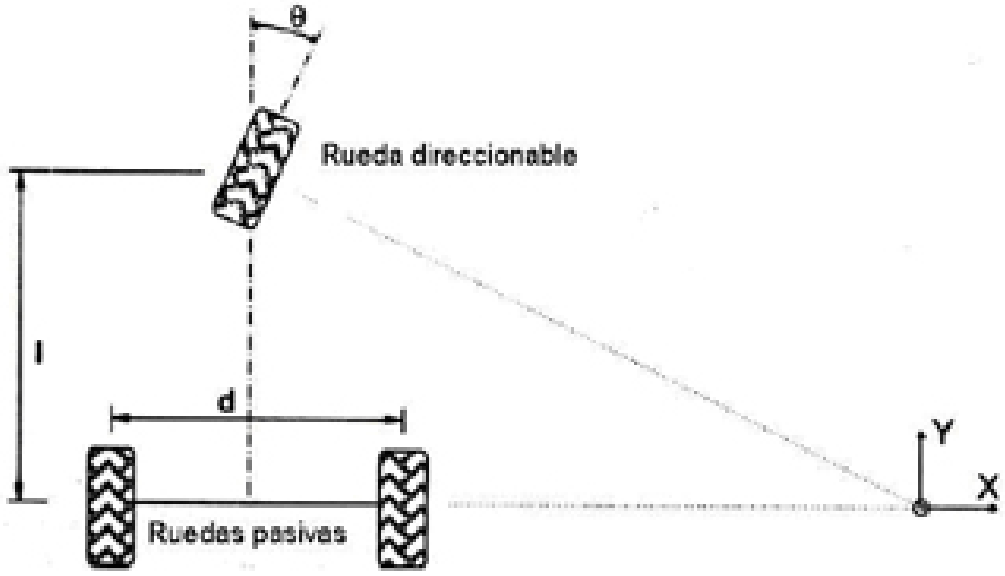


Figura 2.6 [35] Robot configuración triciclo

Otra configuración a considerar es la omnidireccional. Esta configuración permite a los robots moverse en cualquier dirección sin la necesidad de realizar maniobras de orientación previas a este movimiento, a su vez, es capaz de hacer cualquier movimiento en el plano. Pero esto supone un gran nivel de complejidad a la hora de ser utilizado e implementado. En este tipo de configuración, necesitamos como mínimo 3 ruedas activas en nuestro robot.[]
El cálculo de la odometría con esta configuración es mucho más complicado y además el coste de implementación es considerable. [4][5]

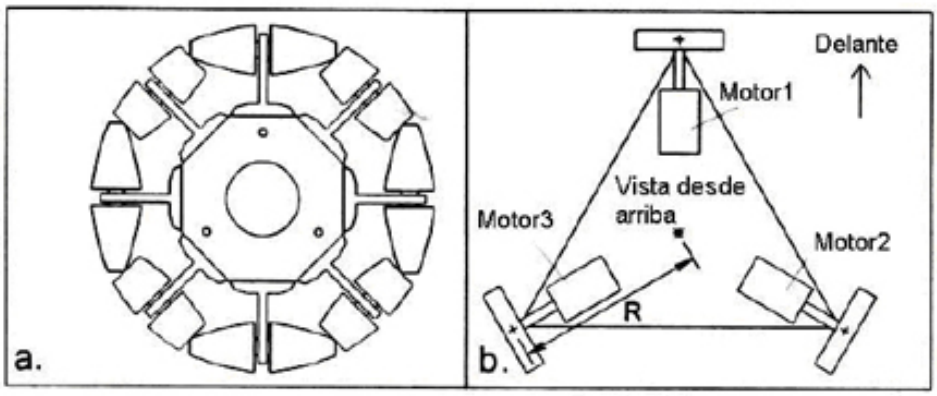


Figura 2.7[36] Robot configuración omnidireccional

La configuración síncrona es una de las más innovadoras. Se trata de tres o más ruedas todas conectadas entre sí mediante mecanismos y todas están dotadas de un sistema de tracción, de forma que consigamos que todas apunten en la misma dirección y giren a la misma velocidad. Nos plantea el problema de que necesita una gran sincronización y mecanismos muy complejos para poder llevarla a cabo adecuadamente. [4]

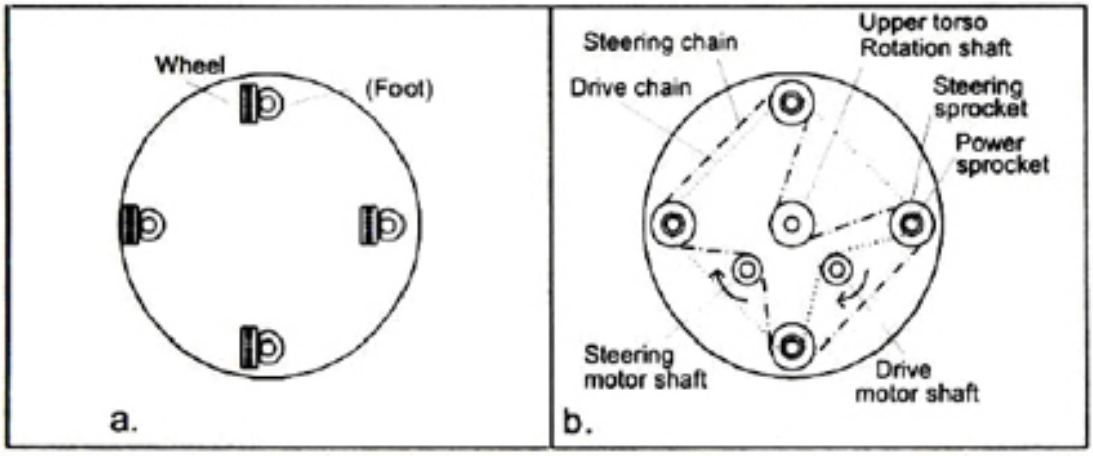


Figura 2.8[37] Robot configuración síncrona

Y ya por último, la configuración que encontramos en la mayoría de coches, y la que nos ocupa en este proyecto, la configuración *ackerman.* En este caso el robot tiene 4 ruedas de las cuales solo 2 son de tracción, y su singularidad es que las dos ruedas delanteras son las que son direccionadas para realizar los giros, pero para poder realizarlos hay que tener en cuenta que el giro de ambas ruedas no puede ser el mismo, sino que la rueda interior al giro presenta un ángulo más agudo que la rueda exterior.
Las normales de ambas ruedas se cortan en un punto que pertenece a la prolongación del eje de las ruedas traseras. Con este dato es posible probar que ambas ruedas describen, para ángulos de giro constantes, trayectorias concéntricas.

La relación entre los ángulos viene dada por la ecuación de *Ackerman*:

$$\cot(\theta 1) - \cot(\theta 2) = \frac{d}{l}$$

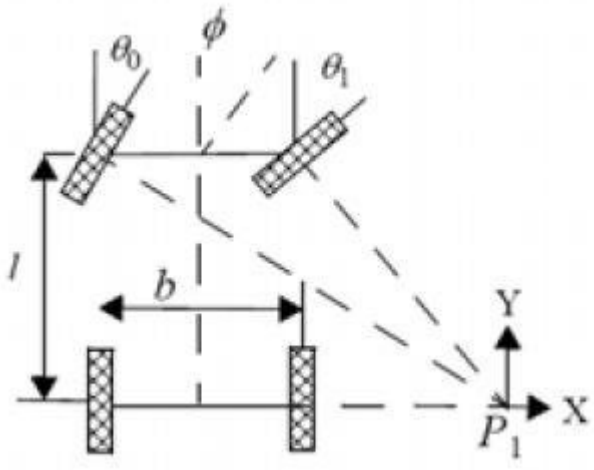


Figura 2.9[38] Robot configuración *ackerman*

Esta configuración nos facilita el cálculo de la odometría respecto a algunos de los modelos anteriores y a su vez nos asegura un buen sistema de tracción con independencia de la inclinación del terreno, sin embargo, cabe mencionar que complica de forma exponencial la construcción mecánica del robot.

También nos permite utilizar las ecuaciones cinemáticas del modelo del triciclo, ya que podemos "simplificar" las dos ruedas delanteras como una única rueda cuyo ángulo de giro sea la media de los dos.

2.2 ROS.

2.2.1 Definición de ROS.

ROS (*Robot Operating System*) es un sistema de programación modular especialmente enfocada para robots y sistemas móviles. Es un *software* libre que, hasta la fecha solo se encuentra disponible en sistemas operativos con base *Linux* como *Ubuntu* y en *Mac OS X*, se está trabajando en el desarrollo del mismo para *Windows*, pero de momento este no se encuentra disponible aún.

Se trata de una colección de librerías, herramientas y convenciones que tienen como objetivo simplificar la creación de programas que permiten establecer de forma segura y robusta el comportamiento, que puede ser más o menos complejo, de un robot, permitiéndonos trabajar a su vez en una gran variedad de modelos.

Esto, explican en la propia página de *ROS*, es debido a que es muy difícil conseguir que los robots hagan ciertos movimientos o respondan a ciertas situaciones que para nosotros los humanos son muy sencillas u obvias, y que, sin embargo, desde el punto de vista del robot puede resultar muy complicado, si, como programadores, hemos de enfrentarnos a la programación de los mismos de forma individual.[6]

Este software nos ofrece los mismos servicios que un sistema operativo, como puedan ser una capa de abstracción de *hardware*, control de dispositivos a bajo nivel, implementación de funcionalidades más utilizadas, mensajes entre procesos y gestión de paquetes además de las librerías y herramientas mencionadas anteriormente.[7]
ROS, a su vez, está diseñado para ser lo más ligero posible, y permitir que el código utilizado en *ROS* pueda ser utilizado e integrado a otros softwares de control de robots.
También se busca que sea implementable con cualquier lenguaje de programación moderno. Funciona correctamente con *Python*, *C++* y *Lisp*, y ahora mismo existen y están en desarrollo librerías experimentales con *Java* y *Lua*.
ROS, así mismo, destaca también por su facilidad para realizar pruebas y ensayos ya que tiene integrada una herramienta para ello *(rostest*).

2.2.2 Conceptos generales de *ROS*.

2.2.2.1 Filosofía de *ROS*

Para comprender con mayor facilidad la filosofía en la que se basa este *software* es importante entender aspectos más concretos de la misma, como que está basado en un comportamiento *peer-to-peer*, o lo que es lo mismo, todos los pequeños programas que pueden formar un sistema *ROS* se comunican mediante mensajes que circulan de uno a otro sin pasar por una rutina de servicios central.
A su vez, se trata de un sistema basado en herramientas. Para el caso de *ROS* tenemos herramientas pequeñas que realizan tareas muy específicas, y éstas pueden ser modificadas o sustituidas por implementaciones mejores que realicen mejor la tarea deseada.
Otra de las cosas que destacan de *ROS* es, como ya se ha mencionado antes, que soporta un gran número de lenguajes de programación. Esto permite que para cada tipo de tarea o de función que se busca podamos utilizar el lenguaje más óptimo para ello, permitiendo así una mayor eficiencia en los programas realizados.
Por último, mencionar que se trata de código de acceso libre y gratuito. *ROS* está bajo una licencia *BSD*, que nos permite un uso tanto comercial como no.

2.2.2.2 Terminología

Antes de adentrarnos más en el mundo de *ROS* es necesario explicar parte de la terminología y algunos conceptos básicos[8] de este *software*:
· Los paquetes, que son la forma básica y principal de agrupar la información dentro de *ROS*. En estos paquetes podemos encontrar nodos, las librerías dependientes de *ROS*, conjuntos de datos, archivos de configuración o cualquier tipo de archivo o información que sea útil junto al resto de contenidos del paquete. Son la unidad más básica indivisible que se puede crear y publicar en *ROS*.
· Los metapaquetes, que se tratan de paquetes especializados cuya función es agrupar paquetes relacionados entre sí o que tienen una funcionalidad común.
· Los manifiestos: se tratan ficheros con extensión *.xml* que nos dan metadatos sobre los paquetes como por ejemplo: el nombre, la versión, la descripción y más información adicional.

· Mensajes: son los ficheros que describen las estructuras de datos para los mensajes enviados en *ROS*, esto es, son los ficheros que contienen la información que se intercambia en cualquier proceso de *ROS*.
· Servicios: son los ficheros que definen las solicitudes y las estructuras de datos de respuesta requeridas por los procesos de *ROS*.

2.2.2.3 Sistema de computación a nivel gráfico de *ROS*.
Todos los procesos de este sistema están conectados mediante una red *peer-to-peer* en la que todos los elementos de la red son capaces de procesar datos y a su vez están todos conectados entre sí. [8]
Los conceptos más básicos son los siguientes:
· Nodos: Son los procesos que realizan los cálculos. *ROS* está diseñado para trabajo modular detallado, por lo que el proceso de control de un robot está formado por diversos nodos. Estos nodos se escriben mediante la utilización de funciones de biblioteca de *ROS* como *roscpp* y *rospy*.
· *Master*: Es el que da nombre y un "almacén" al resto del proceso. Sin el *Master*, no sería posible para los nodos encontrarse unos a otros, enviarse mensajes o invocar servicios.
· Servidor de parámetros: Forma parte del *Master*, permite el almacenaje de datos en una posición centralizada utilizando claves.
· Mensajes: Son el medio de comunicación de los nodos, se tratan de estructuras simples de datos que constan de variables. Los tipos de variables estándar están permitidas, así como matrices de este tipo de variables.
· *Topic*: Son el modo de transporte de los mensajes basados en un sistema de publicación/suscripción semántico. Un nodo envía un mensaje publicándolo en un *topic*, el *topic* es el nombre que se le da al contenido de mensaje con el fin de identificarlo. Para recibir la información que necesita, un nodo ha de suscribirse al *topic* apropiado. En los *topics* puede haber tantos nodos que publican como suscriptores, sin número máximo. A su vez, un nodo puede estar publicando en diversos *topics*.
· Servicios: El método de publicación/suscripción es una forma de comunicación muy flexible, pero al ser "de muchos a muchos" y solo permite comunicación en un sentido, no es apropiado para solicitudes y respuestas, que suelen ser necesarios en un sistema distribuido, por lo que para esto *ROS* utiliza los servicios, que se definen como un par de estructuras de mensaje, una para solicitud y otra para respuesta.

Todo estos estos conceptos serán ampliados y contextualizados más adelante.

2.2.2.4 Gráfico de *ROS*
Es una ilustración que nos da la información de cómo, en un sistema de *ROS* los diferentes programas o nodos que lo forman, se intercambian los datos y mensajes entre sí.
Es una herramienta muy potente puesto que nos permite representar cualquier sistema de *ROS* sin importar la complejidad de este, ya que lo único que cambiará es el número de conexiones y nodos.

En los gráficos que nos devuelve *ROS* consideramos los nodos como piezas de *software* que están recibiendo o enviando información y las flechas como mensajes que se están transmitiendo en el sentido que nos indica.

2.2.3 Componentes básicos de ROS.

2.2.3.1 Servicio *roscore*.

Roscore es un servicio que nos ofrece ROS y que es lo que permite que los nodos puedan comunicarse entre sí mediante el envío de mensajes. Es necesario para cualquier aplicación de *ROS* que tengamos abierto un *roscore*.
Cada nodo se conecta al *roscore* en cuanto se inicia y le envía los detalles de los mensajes que desea enviar y a los que se quiere suscribir, también establece las conexiones *peer-to-peer* con otros nodos cuando uno nuevo aparece, por lo que podemos afirmar que no se trata de una conexión *client/server* como las que tenemos para servicios web, pero tampoco una red *peer-to-peer* pura, en la arquitectura de *ROS* tenemos algo intermedio. [9]

Cada nodo informa al *roscore* de a qué mensajes quiere suscribirse y qué mensajes envía y es entonces cuando *roscore* informa de las direcciones a aquellos interesados en dichos mensajes.
roscore también nos provee de un servicio de parámetros (*parameter server)* que es utilizado por los nodos para su configuración. Permite a los nodos almacenar y recuperar datos arbitrarios como pueden ser descripciones de robots, parámetros para algoritmos, etcétera. Para esto, como para la mayoría de servicios de *ROS*, tenemos un comando sencillo *rosparam*.

2.2.3.2 *Catkin* y *workspaces*.

Para poder trabajar con *ROS*, además de todos los conceptos explicados anteriormente, es importante conocer cómo son los espacios de trabajo en los que se desarrollarán los programas.
Catkin es el sistema de construcción de ROS desde la versión *Groovy* (antes se utilizaba el comando *rosbuild*) que mediante la combinación de macros *CMake* y *scripts* en lenguaje *Python* nos permite la generación de *targets* a partir de puro código y permite que pueda ser utilizado por un usuario final. Estos *targets* se traducen en librerías, programas ejecutables, interfaces o cualquier cosa que no sea código estático. Estos códigos se agrupan por funcionalidades comunes en los ya mencionados paquetes.

Para construir estos paquetes, el sistema de construcción necesita información como la localización de los componentes de la cadena de herramientas (como el compilador de *C++*), la localización de los códigos fuente, las dependencias tanto de los códigos como externas, donde se encuentran esas dependencias, qué *targets* se deben construir y dónde. Esta información es la que, con el sistema *CMake*, encontramos en los ficheros llamados habitualmente *"CMakeLists.txt"* que se encuentran en todos los paquetes de *ROS*.

Catkin es un sistema de construcción a medida que permite que *CMake* maneje las dependencias entre paquetes.

Esto es necesario para *ROS*, ya que éste está formado por un conjunto de paquetes libremente federados, por lo que podemos tener muchas dependencias entre paquetes a priori independientes que además utilizan diferentes lenguajes de programación, por esto, el proceso de construcción de un *target* puede variar mucho entre sí, incluso dentro de un mismo paquete. El objetivo de *catkin* es facilitar la construcción y el funcionamiento de código *ROS* mediante la utilización de herramientas y convenciones para simplificar el proceso.
Esta herramienta es en la que basamos la construcción de los espacios de trabajo (*Workspaces* a partir de ahora).

Antes de realizar cualquier programa en *ROS* es necesario crear un *workspace* donde se guardará y se trabajará con el código. Un *workspace* simplemente es un conjunto de directorios en los que una agrupación de códigos es almacenada. Se puede tener dentro de una máquina una gran cantidad de *workspaces*, pero solo se puede trabajar con uno cada vez.
Existen comandos para la creación de estos *workspaces*, y pueden encontrarse todos en los tutoriales que la propia página de ros nos ofrece http://wiki.ros.org/ROS/Tutorials, sin embargo y para mayor comodidad, se ha adjuntado una guía rápida de instalación (basada en estos tutoriales) en el Anexo I de este documento.
Destacar los más importantes que en este caso son el *catkin_init_workspace* que nos crea el fichero *CMakeList.txt* (se ha de ejecutar cuando estamos dentro del directorio *src* de nuestro *workspace*) y el comando *catkin_make* que se podría comparar con el proceso de compilación de los lenguajes de programación convencionales. *Catkin_make* nos genera dentro del *workspace* dos nuevos directorios: *build* y *devel*. El primero es donde almacenaremos los resultados de la ejecución del catkin, como puedan ser ejecutables o librerías, mientras que en el segundo tenemos una gran cantidad de ficheros y directorios de configuración del workspace. [9]

2.2.3.3 Paquetes

Una vez conocidos los *workspaces*, se profundizará en el concepto de paquete de *ROS*.
Como ya se ha comentado anteriormente, un paquete es la unidad fundamental de agrupar la información en *ROS*. Estos paquetes se almacenan dentro del directorio *src* que se ha creado previamente en el *workspace* y es imprescindible que contengan tanto un fichero *CMakeLists.txt* como un *package.xml*. Estos archivos describen al comando *catkin* cómo debe interactuar con ellos cuando es ejecutado.
En estos paquetes es donde almacenamos los programas realizados en otros lenguajes de programación y los ficheros que son necesarios para que la funcionalidad del paquete pueda ejecutarse correctamente.

2.2.4 Comandos importantes de *ROS*.

2.2.4.1 *Roscore*, *rosrun* y *roslaunch*

Ya se ha explicado anteriormente el comando *roscore*, que es el primer comando que se ha de lanzar a la hora de ejecutar cualquier aplicación de *ROS* y su importancia dentro de cualquier proceso de ROS, pero no es la única instrucción que se utilizará con frecuencia. Aquí un repaso de las más importantes[9]:

Rosrun es un comando que nos permite lanzar los programas sin que exista la necesidad de buscar la ubicación de los mismos dentro del sistema de archivos de *Ubuntu* (esto puede ser tedioso si no se está familiarizado con el uso de la terminal). Únicamente utiliza el nombre del paquete y el programa concreto que se desea ejecutar, sin importar dónde se encuentre la terminal situada dentro del sistema de ficheros. La única condición que se ha de cumplir previamente es que se haya ejecutado el comando *roscore* y éste esté activo.

Roslaunch se trata de un comando que nos permite lanzar varios nodos de forma simultánea. A la hora de ejecutarse, se hace de forma muy similar al *rosrun*, pero en este caso se necesita un tipo de ficheros especiales con extensión *.launch*. Son ficheros *XML* que almacenan toda la información necesaria de los nodos que se quiere lanzar. También nos permite ejecutar programas de forma remota en otros ordenadores a través del protocolo *ssh*, esto se utilizará posteriormente en la comunicación con el *rbcar*.

2.2.4.2 Otros comandos.

Tenemos otra serie de comandos, mucho más sencillos, pero no por ello menos importantes que nos ayudarán a realizar todas las acciones que deseamos y nos permitirán navegar por el sistema de archivos de *ROS*:

- *rospack*: Este comando nos permite recibir información sobre los distintos paquetes que se encuentran en nuestra máquina, como por ejemplo, que devuelva por pantalla la ubicación de un paquete con el argumento '*find*' tras el comando.
- *roscd*: Este comando es el equivalente a *cd* para *ROS*. Nos permite la navegación por las distintas carpetas o paquetes desde la terminal de *Ubuntu*.
- *rospd*: Similar al comando anterior, pero este comando recuerda la última ubicación en la que estábamos.
- *rosls*: Nos da una lista de los paquetes que podemos encontrar en el directorio en el que nos encontramos actualmente
- rosls: Similar al anterior, pero esta vez devuelve una lista de todos los archivos que podemos encontrar en el directorio.
- *rosed*: Nos permite la modificación de un fichero que se encuentra dentro de un determinado paquete desde la terminal.
- *roscp*: Este comando es utilizado para copiar un fichero de un paquete a otro.

Estos comandos se encuentran dentro de un paquete que está incorporado a *ROS*, llamado *rosbash*, y que además nos permite la opción de, a la hora de trabajar con ellos, autocompletar cuando presionamos la tecla tabulador, lo cual ahorra mucho tiempo y errores de escritura al trabajar desde la terminal directamente.

2.2.5 Sistemas de comunicación entre nodos:

2.2.5.1 *Topics*

Topics: la forma más común de comunicación entre nodos. Un *topic* es un nombre que se le da la retransmisión de mensajes de un tipo definido. Se utilizan junto con un mecanismo de comunicación basado en publicaciones/suscripciones. Antes de enviar la información los nodos han de informar (como ya se ha mencionado anteriormente, al *roscore*) del nombre del *topic* y del tipo de datos que se van a transmitir, entonces pueden realizar el envío de información. Para realizar la suscripción es algo similar, pero informando de qué *topic* estamos recibiendo información.

Dentro de los nodos es donde se indica a qué *topics* se publica/suscribe. Cada lenguaje de programación establece el tipo de mensaje y la forma de transmisión de datos como corresponde.

Dentro de un programa se puede tener un gran número de *topics* transmitiendo mensajes, además de que puede ser necesario conocer cierta información referente a éstos, por lo que *ROS* habilita también un comando, *rostopic*, que simplifica mucho la tarea de trabajar y conocer los parámetros de los *topics.* [9]

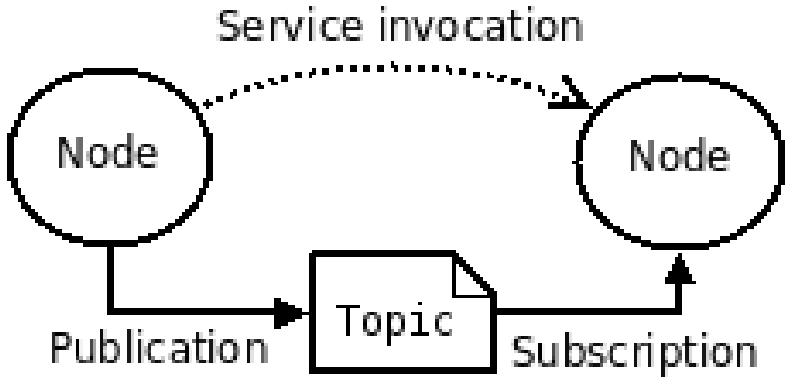


Figura 2.10[39] Esquema funcionamiento *topics/services* en *ROS.*

2.2.5.2 *Services*

Services: Otro sistema de transmisión de datos que nos ofrece *ROS*. Se tratan de llamadas a procedimientos remotas y sincronizadas. Nos permiten que un nodo realice la llamada a una función que ejecuta otro nodo. La información recibida y enviada mediante este procedimiento se define de forma muy similar a los mensajes. El servidor (aquell que envía el servicio) especifica una *callback* (una retrollamada) para lidiar con la solicitud, y anuncia el servicio. El cliente (el que solicita el servicio) entonces accede al servicio mediante un *proxy* local.

Las llamadas a servicios son muy apropiadas para funcionalidades que, a lo largo de nuestra aplicación utilicemos un número reducido de veces y precisen obligatoriamente de un tiempo de proceso, por ejemplo, algún tipo de cálculo que se quiera distribuir a otros ordenadores. Otras funciones discretas que pueden realizarse perfectamente con services pueden ser el encendido de un sensor o la realización de una fotografía con una cámara. [9]

2.2.5.3 *Actions*

Actions: La mejor forma de las que nos ofrece *ROS* para implementar interfaces que se extienden a lo largo del tiempo o consisten en alcanzar un objetivo. El mejor ejemplo para esto es un programa en el cual el vehículo tiene que alcanzar una posición determinada. Al contrario que los *services*, las *actions* son asíncronas, sin embargo presentan un comportamiento similar a la solicitud y respuesta de los services. Una *action* utiliza un *goal* (objetivo) para iniciar un comportamiento en el robot y envía un resultado cuando este ha sido alcanzado. Además, las *actions* utilizan también *feedbacks* que van actualizando el progreso del comportamiento frente a ese objetivo, y a su vez, nos permiten poder cancelar estos *goals*. Las actions se implementan a sí mismas mediante la utilización de topics. Una *action* es, básicamente, un protocolo de alto nivel que especifica cómo han de utilizarse una determinada combinación de *topics*.[9]

2.2.6 Simuladores

2.2.6.1 Definición e importancia

Los simuladores son una parte clave de la robótica. Permiten poder depurar los códigos y los modelos matemáticos antes de implementarlos en el robot, evitar ciertos errores que pueden provocar que la vida útil del robot quede reducida muy considerablemente. Además, facilita la tarea de elegir un robot para una función determinada sin necesidad de hacer ningún tipo de inversión previa y con unos errores mínimos, por lo que es interesante conocer aquellos con los que se va a trabajar. En este caso se va a utilizar tanto *Gazebo* como *Rviz*.

2.2.6.2 *Gazebo*

Gazebo es un software de simulación libre, únicamente disponible en sistemas operativos de base *Linux* y se presenta como una herramienta potente, con posibilidad de simular tanto entornos exteriores como interiores, con una interfaz programable y gráfica.

Nos permite modificar de forma muy sencilla modificar la rigidez y los motores físicos de la simulación, lo cual permite que lo que se ensaya pueda ajustarse a mayor número de situaciones.[10]

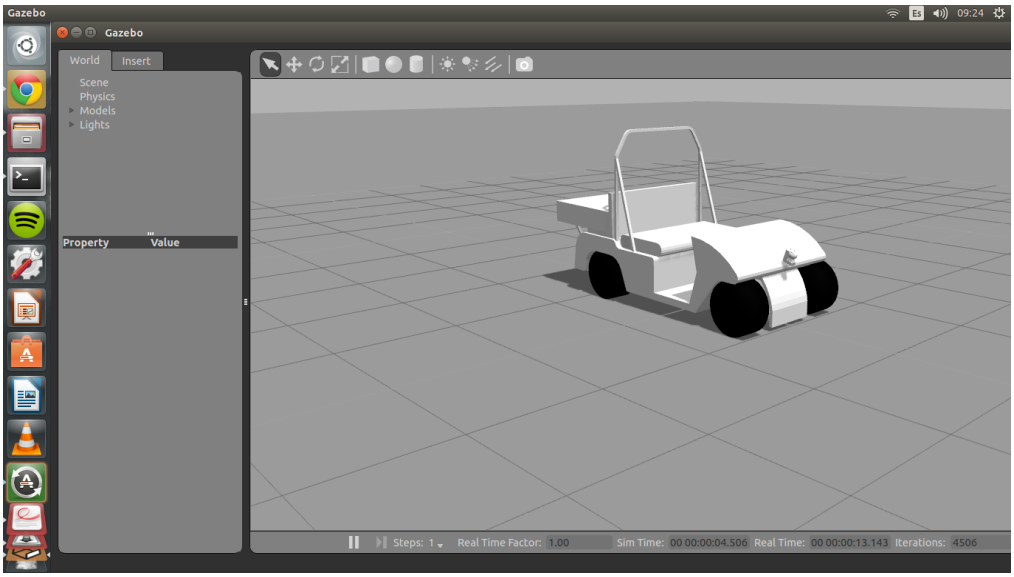


Figura 2.11 *Rbcar* en el entorno *Gazebo*

2.2.6.3 *Rviz*

Rviz es un visualizador gráfico que nos permite la representación en 3D de entornos para robots, sensores y algoritmos. Es fácilmente configurable para cualquier tipología de robot y nos permite de forma sencilla mostrar en pantalla o en un entorno gráfico un considerable número de variables de *ROS*, con énfasis especial en las variables 3D. Estos datos están siempre expresados respecto a un sistema de referencia.

Rviz también nos ofrece un gran número de plugins y paneles que son configurables y modificables para satisfacer la función que queramos realizar con él. [11]

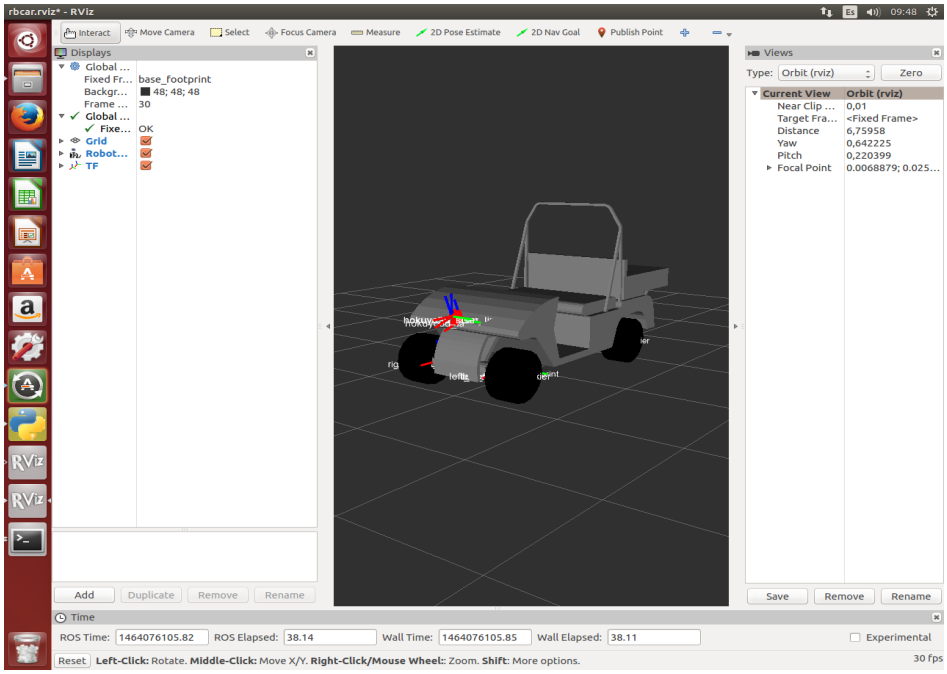


Figura 2.12 *Rbcar* en el entorno *Rviz*

2.3 Transformación de coordenadas

2.3.1 Justificación

Dentro de la robótica, y concretamente dentro de la robótica móvil, la utilización y gestión de los distintos sistemas de referencia que tenemos en nuestro sistema es de vital importancia. Con el fin de facilitar esta tarea, *ROS* incorpora el paquete *tf*.

En primer lugar, justificar la existencia y utilización de este paquete. A la hora de trabajar con robots es importante la diferenciación de distintos espacios en los que se va a trabajar, como el entorno por el que se va a mover el robot, la base de este y las diferentes partes de las que consta. Cada elemento de los citados anteriormente consta de un sistema de referencia distinto y, en algún momento de nuestra programación, necesitaremos conocer la información que viene dada de uno de estos sistemas a otro. Esta es la principal función del paquete tf, que la comunicación entre distintos sistemas (o *frames*) sea lo más sencilla y directa posible.

Así pues, *tf* puede definirse como la parte del *software* de *ROS* que se encarga de mantener y controlar todos los *frames* que haya a la hora de implementar la funcionalidad del robot a lo largo del tiempo.[12] Establece la relación que existe entre los diferentes *frames* de nuestro robot y la representa en estructuras con forma de árbol en función del tiempo, ya que estos sistemas cambian respecto a él.

Tf opera en un sistema distribuido, esto es, toda la información de los sistemas de referencia con la que se trabaja desde este paquete está disponible para todos los componentes de la aplicación de *ROS*, desde cualquiera de las máquinas que se están utilizando para ello. Se utilizan *topic*s para la transmisión de los datos de las transformadas, lo cual nos permite que la publicación de esta información sea realizada por cualquier nodo, pudiendo seleccionar nosotros el más conveniente o aquel en el cual nos sea más fácil encontrar o procesar los datos que deseamos publicar. También nos permite elegir en qué nodos se realizan las transformaciones de unos sistemas a otros.[9]

2.3.2 Funcionamiento y estructura

Este paquete se enfocó de forma que su uso fuera lo más sencilla e intuitiva posible, por lo que hay ciertas características que ayudan a la hora de procesar los datos de las transformadas, por ejemplo, para trabajar con un cierto dato solo es necesario conocer el nombre del sistema de referencia en el que se encuentran. Otra de las características de *tf* es que solo se producen las transformaciones necesarias en el momento que se solicitan, por lo que aumentamos la eficiencia y simplicidad del proceso. También, destacamos que no necesita configuración previa al ser ejecutado y es fácilmente modificable una vez está en marcha, lo cual nos da un indicativo de lo versátil que es este paquete.

Sin embargo, *tf* también presenta algunas restricciones en su uso. La más recalcable, quizás, es que a pesar de que es capaz de guardar ciertos datos de sucesos ocurridos con anterioridad, únicamente nos puede guardar los 10 últimos segundos, no obstante, esto suele ser más que suficiente para la mayoría de aplicaciones.[12]

Como ya se ha comentado anteriormente, trabajamos con gran cantidad de sistemas de referencia en la mayoría de aplicaciones, por lo que es muy importante ser capaces de distinguirlos unos de otros. Cada sistema de referencia ha de poseer un nombre único para poder ser diferenciados sin problemas.
Es importante, a su vez, conocer la relación que puede existir dentro de estos sistemas de referencia a la hora de realizar aplicaciones con ellos. Para poder transmitir la información de un *frame* a otro es necesario que los nodos realicen las transformadas correspondientes y publiquen el mensaje en el sistema en el que queremos trabajar. En este caso el sistema que envía la información se conoce como padre y el que la recibe es hijo. Una de las cosas más importantes a considerar a la hora de trabajar con este paquete *tf* es que un hijo únicamente puede tener un padre, esto es, un sistema solo puede recibir la información transmitida por un nodo que convierte o transporta la información de otro sistema de referencia, pero sólo uno. Sin embargo sí es posible para un padre retransmitir información a más de un hijo, que es lo que encontraremos en el árbol de transformadas que conforma nuestro proyecto y que se entrará en detalle en él más adelante.

2.3.3 Convención de nombres de sistemas y definición

Los sistemas utilizados en nuestro proyecto forman parte de una convención que es utilizada por gran parte de los usuarios de ROS con el fin de fomentar y ser capaces de reutilizar código y fomentar la compatibilidad entre dispositivos, programas y robots.

Los sistemas más importantes a la hora de trabajar con robots móviles son [13]:

/map: Es el que podríamos considerar como sistema de referencia global. Se trata del *frame* del mundo virtual o el entorno sobre el cual estamos trabajando con nuestro vehículo. Se encuentra fijo y el resto de elementos varían su posición respecto a este de forma discreta. Para ayudar a la localización de los elementos en este sistema, se utiliza la información dada por los sensores que puedan existir en el vehículo. Útil como referencia global a largo plazo pero los saltos discretos lo hacen poco apropiado para actuación local en función de la información de los sensores.
/odom: Este sistema de referencia, así como el anterior, se considera fijo. La posición de un vehículo móvil respecto a odom varía de forma continua sin los problemas que podíamos tener en */map* al ser un sistema discreto. La información de este sistema de referencia suele venir dada por algún tipo de sensor incorporado en el vehículo, como pueda ser un *encoder* en las ruedas o distintas formas de cálculo de la misma. */odom* se utiliza para cálculos en sistemas locales y a corto plazo.
/base_link: Este sistema de referencia se encuentra adherido al cuerpo del robot. Habitualmente se halla situado en el centro de masas del mismo, aunque esto no es obligatorio y el usuario tiene la libertad de ponerlo donde considere, existen convenciones para situar este sistema en las páginas de documentación de *ROS*.
/base_footprint: Se trata de una proyección en el suelo del espacio en el que nos encontremos del sistema /base_link, esto es, este sistema y el anterior comparten tanto la coordenada X como la Y en todo momento y la única que los diferencia es la Z que en el caso de */base_link* es un valor distinto a 0 en la mayoría de casos y en */base_footprint* es igual a 0.

A partir de aquí surgen los distintos controladores de las partes del robot como pueden ser las ruedas, los sensores láser o el *gps* integrado en el coche.

2.4 Sistema de Navegación

Para entender lo que queremos que el coche realice es imprescindible presentar y explicar los paquetes de *ROS* que nos permitirán implementar la funcionalidad que buscamos.
Paquete *Navigation*:
El paquete *Navigation* está compuesto por una serie de paquetes y nodos que nos permiten la implementación de conducción autónoma en el *rbcar*. De estos paquetes únicamente se explicarán aquellos más importantes para el desarrollo del trabajo.[14]

2.4.1 *Move_base*

Así pues, una de las partes más importantes de *navigation* es, sin duda, el *move_base*. Este paquete podría definirse como una caja o un contingente de los paquetes que nos dan la funcionalidad buscada. Estos paquetes que están dentro del *move_base*, están relacionados entre sí y a partir de la información que recibimos, o bien de los sensores o del fichero donde tenemos el mapa por el que vamos a navegar, podemos generar las velocidades que serán enviadas al controlador del coche.[15]
Pero, para entender cómo hacemos esto, vamos a desglosar el *move_base* en sus partes más básicas.

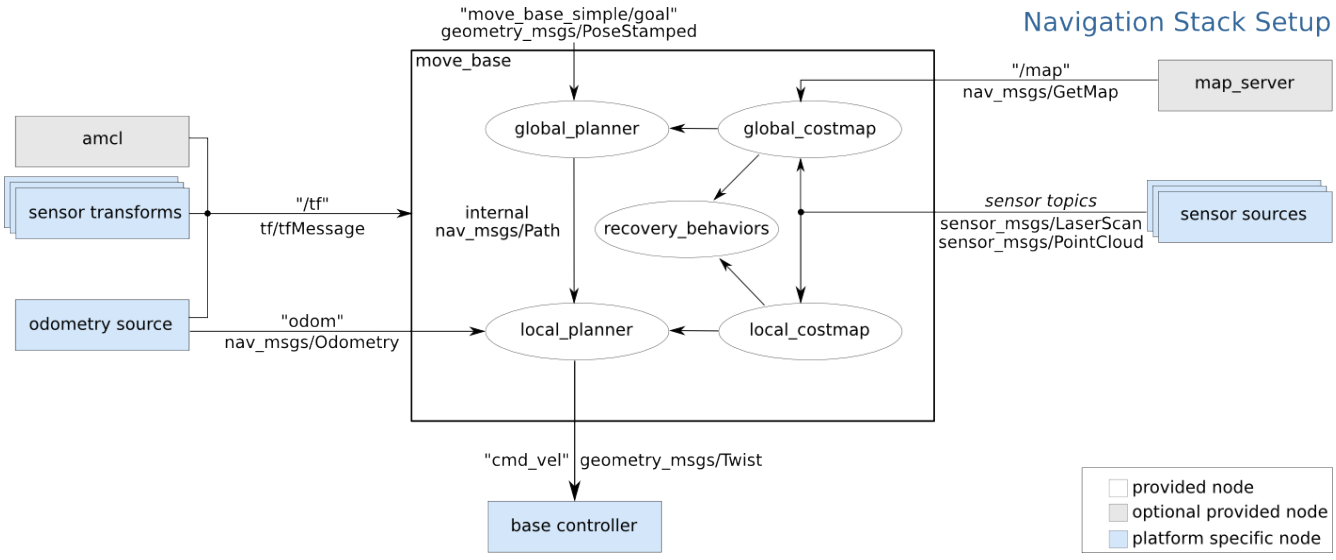


Figura 2.13 [40] Esquema *move_base*

El *move_base* está formado por, como puede verse en la figura, el *global_planner*, *local_planner*, *recovery_behaviors*, *global_costmap* y *local_costmap*.

2.4.1.1 *Costmaps*

En primer lugar, se indagará en los *costmaps*, tanto en el local como en el global. La función de estos paquetes es, a partir de la información recibida del mapa (en caso del global) y del sensor (para ambos), establecer una especie de margen de seguridad frente a obstáculos que el coche puede encontrarse durante su ruta y penaliza si se incumplen. Esto nos permite evitar accidentes y colisiones. En caso del global lo que hace es tener en cuenta los obstáculos fijos, como paredes o pilares y los "infla", esto es, les da un margen el cual el coche está programado para no infringir, o para que en caso de hacerlo sea lo mínimo posible. El local, al recibir únicamente la información de los sensores, se centra en los obstáculos móviles y que no están representados en el mapa en un primer lugar, como puedan ser personas que circulan por el mismo espacio que nuestro robot. [16]

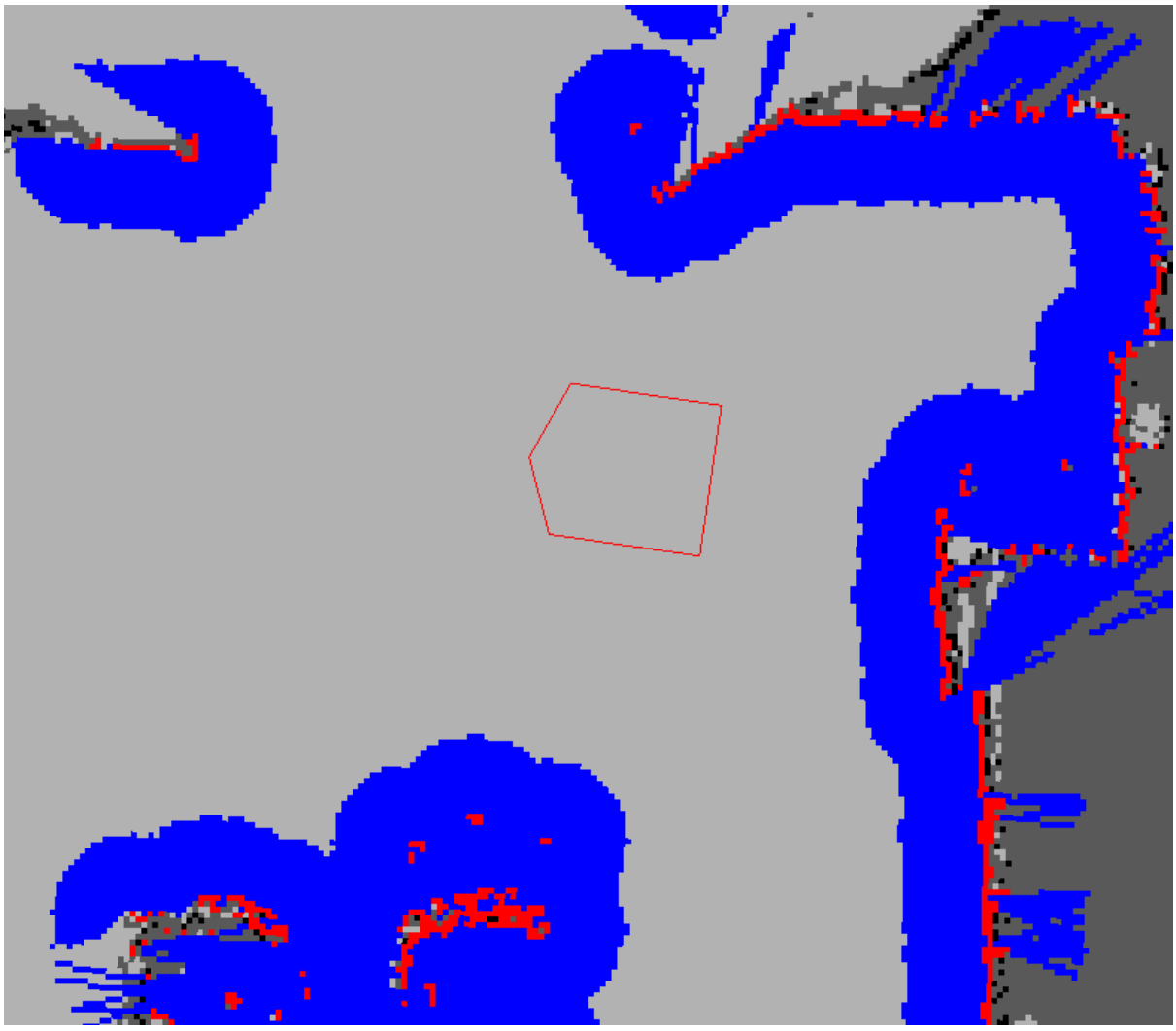


Figura 2.14 [41] Ejemplo de *costmap*

En este ejemplo de *costmap*, tenemos el vehículo representado como el polígono rojo. Las celdas rojas en el entorno representan obstáculos para el vehículo y las celdas azules son la distancia de seguridad a esos obstáculos.

2.4.1.2 Planificadores

Una vez el *costmap* ha producido y procesado esta información se envía al planificador global que generará la trayectoria del vehículo y, el planificador local con esto, calculará las velocidades del vehículo que, posteriormente, se envían al controlador del coche.

El planificador global (*global_planner*) es el que, una vez recibida la información de los *costmaps*, y tras haber recibido un punto de origen y un punto de llegada, genera los distintos puntos que formarán la trayectoria del vehículo, esto es, nos genera una serie de posiciones que ocupará el robot para poder ir del punto de origen, que por lo general viene dado por la posición en la que se encuentra el robot en ese momento y el punto de llegada,[17] que en este caso vendrá generado por otro paquete del que se hablará posteriormente llamado *rbcar_wpts*.

El planificador local (*local_planner*) es el que una vez ha recibido las posiciones que han sido generadas en el planificador global las convierte en velocidades que son enviadas al controlador del coche. Este planificador también es el que nos permite la evasión de obstáculos no planeados y/o móviles como personas u otros vehículos, pudiendo modificar los puntos que venían dados por el planificador global para evitar colisiones.

Para cumplir las mismas funciones, tenemos varias opciones de planificadores y este paquete nos da la posibilidad de cambiarlos, ya sea para pruebas o porque ya se ha testeado que cierto tipo de planificador funciona mejor con el tipo de robot que estamos utilizando.

2.4.1.2.1 Planificadores globales

En este caso se ha sustituido el planificador global que viene de serie con el navigation por dos, que son el *sbpl* y el *navfn*.

El *sbpl*, que se encuentra en el paquete *sbpl_lattice_planner*, genera la trayectoria para ir desde el punto inicial hasta el objetivo mediante la utilización de “movimientos primitivos”. Éstos se podrían definir como un conjunto de movimientos cortos y posibles para nuestro robot. La planificación se realiza en dos dimensiones, esto es, x e y, pero que a su vez también tiene en cuenta la inclinación en la misma, cosa que nos resulta muy interesante, ya que, sobre todo en el *rbcar,* esta inclinación ha de tenerse en cuenta a la hora de llegar a nuestros objetivos, y tiene en cuenta, a la hora de programar las rutas, las limitaciones que podamos tener con nuestro vehículo cosas, como por ejemplo, el ángulo máximo de giro.[18]

El *navfn* asume nuestro robot como un cuerpo circular y realiza la generación de las trayectorias a partir del algoritmo de *Dijkstra*, o bien por métodos más empíricos, en este caso el algoritmo de búsqueda *A** (A estrella).[19]

2.4.1.2.1 Planificadores locales

También se ha cambiado el planificador local, se ha sustituido el *dwa_local_planner* que venía de serie con el paquete navigation por el *teb_local_planner*. Este cambio se debe a que, mientras que el *dwa (Dynamic window approach)* lo que hace es, a partir de los datos del entorno del robot tomados del sensor y velocidades y direcciones calculadas, realiza una simulación de todos los posibles caminos que puede llevar a cabo el robot en las condiciones actuales y las evalúa en función de una serie de parámetros, como la proximidad a los obstáculos, proximidad al objetivo, etc, después de haber descartado aquellas opciones que desemboquen en una colisión y elige aquella que haya conseguido una mayor puntuación tras la evaluación [20].

El *teb (Timed Elastic Band)* lo que hace es, a partir de la trayectoria calculada en el planificador global, la optimiza para ser ejecutada en el menor tiempo posible, sin embargo, en el momento en el que durante esta trayectoria el robot se encuentra con un obstáculo, la trayectoria se recalcula y esta empieza a actuar como una banda elástica en la que el obstáculo actúa como una fuerza que tira de ella [21], así pues, y como es de esperar, para el tipo de aplicación que se busca en nuestro robot, parece más lógico la utilización del *teb* ya que al ser un planificador local *online*, nos permite la actualización de la trayectoria durante la conducción y el comportamiento que nos ofrece es más práctico para nuestros propósitos que el *dwa*.

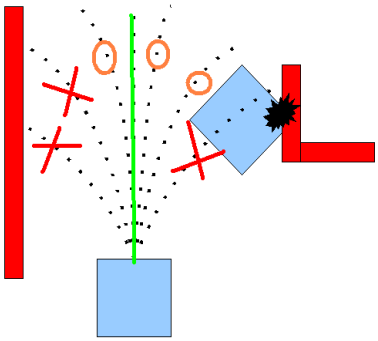


Figura 2.15[42] Comportamiento *dwa*

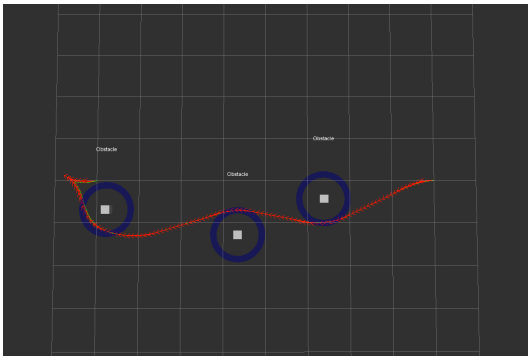


Figura 2.16 Comportamiento *teb*

Lo explicado anteriormente queda reflejado en estas dos imágenes. La Figura 16 representa el *dwa_local_planner*, las líneas de puntos representan las posibles trayectorias, aquellas que se han tachado en rojo son trayectorias que nunca se realizarían por terminar con el vehículo colisionando. Las marcadas con amarillo son aquellas posibles pero con mayor penalización y la verde es la que se realizaría.
La Figura 17 nos muestra la línea roja, que sería la trayectoria que realizaría el vehículo al verse en la situación de encontrar 3 obstáculos en las mismas posiciones cuando implementamos el *teb_local_planne*r en nuestro navigation. La circunferencia azul que rodea a los objetos es un margen de seguridad alrededor de los obstáculos para evitar colisiones.

2.4.1.3 *Recovery_behaviors*

Por último y ya para terminar con el *move_base*, tenemos los *recovery_behaviors*. Se trata de una serie de comportamientos programados para el caso en el que el robot se encuentre en una situación de la que no es capaz de encontrar una solución. No se comentará mucho más sobre este concepto, ya que en el caso de este proyecto, este paquete se encuentra inhabilitado.

2.4.2 *Waypoint*

Como ya se ha mencionado anteriormente, para que el planificador global sea capaz de calcular los puntos que generarán la trayectoria que seguirá nuestro coche y que posteriormente irán al planificador local para generar las velocidades, hemos de ser capaces de enviarle antes dos puntos de referencia, uno inicial, que por lo general suele ser la posición actual del coche, y uno final, que es el destino al que queremos llegar.

Esta es la función del paquete rbcar_wpts, que no forma parte del paquete navigation pero lo incluimos como un complemento a éste, poder enviarle al planificador global una serie de puntos que han sido generados previamente, o bien mediante un fichero de matlab o bien a través de un nodo de *ROS*.
El funcionamiento de este paquete es el siguiente, en primer lugar se carga el fichero que se ha generado con anterioridad (de extensión .*yaml*) que contiene la información de los puntos a los que se quiere llegar con el coche. Esta información es el identificador del punto, que suele ser el número del mismo, el mapa sobre el que se está trabajando, esto es, puede trabajarse sobre un punto de un mapa, que viene dado en función de los ejes de referencia de este mapa, o en otras palabras, trabajando con la posición absoluta de un punto respecto al mapa, o bien podemos trabajar desde la odometría del punto, es decir, puntos que están relativos a la posición actual del coche, y por último la posición del punto al que se quiere llegar, los valores de X eY, y la orientación que debe tener el vehículo al alcanzar dicho punto.
Tras haberse cargado toda la información necesaria, *rbcar_wpts* le pregunta a la *action* si ésta se encuentra disponible. Esta comprobación se realiza de forma cíclica hasta que el paquete recibe una respuesta afirmativa, y en este momento lo que se hace es mandarle al planificador global el punto inicial y el punto final. Aquí, el paquete lo que hace es preguntarle de manera constante al *move_base* si ha llegado ya al punto final, cuando éste le devuelve que ya se ha alcanzado la meta, se envía el siguiente punto de la misma forma, y así sucesivamente hasta que, o bien nos quedamos sin puntos porque no se ha seleccionado que se repita la ruta de forma cíclica, o bien interrumpimos de forma manual el programa porque el bucle está activado.

2.5 Generación de trayectorias a partir de modelos matemáticos

La generación de trayectorias a partir de modelos matemáticos supone una herramienta interesante. Permite describir el camino que va a seguir el robot a partir de una posición inicial y una posición final. Para ello, se generarán puntos auxiliares por los que la trayectoria puede pasar o no (en función del tipo de algoritmo de generación que se seleccione) y, que serán los que la determinen.

2.5.1 Tipos de algoritmos de generación de curvas

Dentro de estos algoritmos existen dos tipos:
Interpolación: que son aquellos que generan la trayectoria a partir de los puntos auxiliares y la trayectoria pasa obligatoriamente por éstos. Un ejemplo de algoritmos de este tipo puede ser la *spline* cúbica natural. [22]

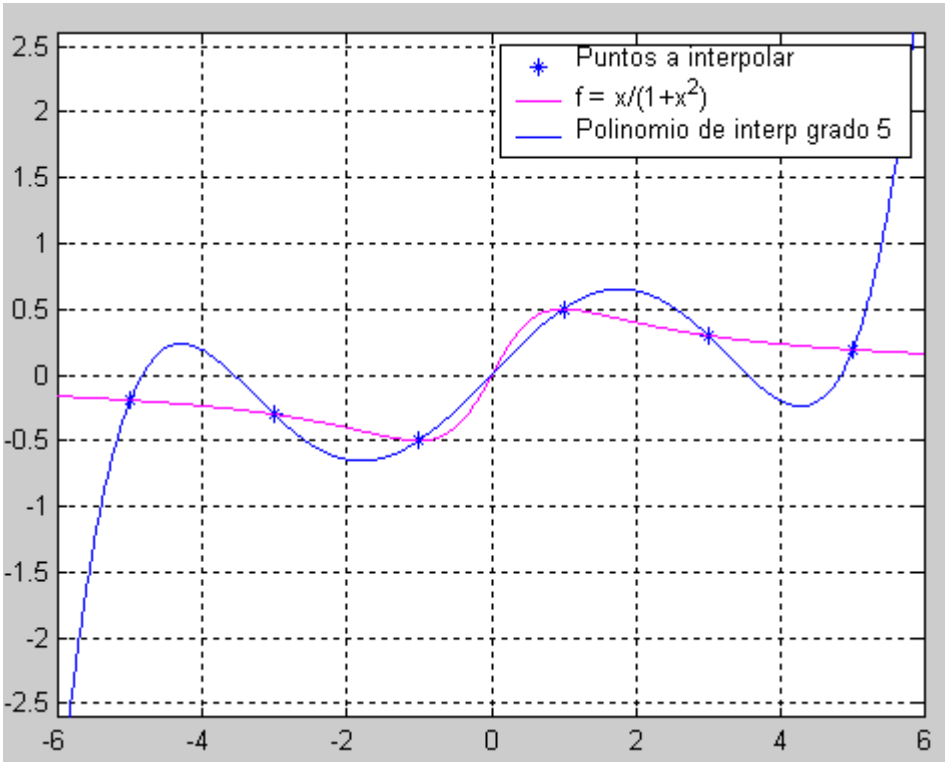


Figura 2.17[43] Ejemplo función interpolación

Aproximación: La trayectoria no pasa exactamente por los puntos marcados, pero estos influyen de forma que la línea descrita por el desplazamiento del robot se desvía para pasar cerca. Como ejemplo de algoritmos de este tipo tenemos las curvas de *Bézier*. [24]

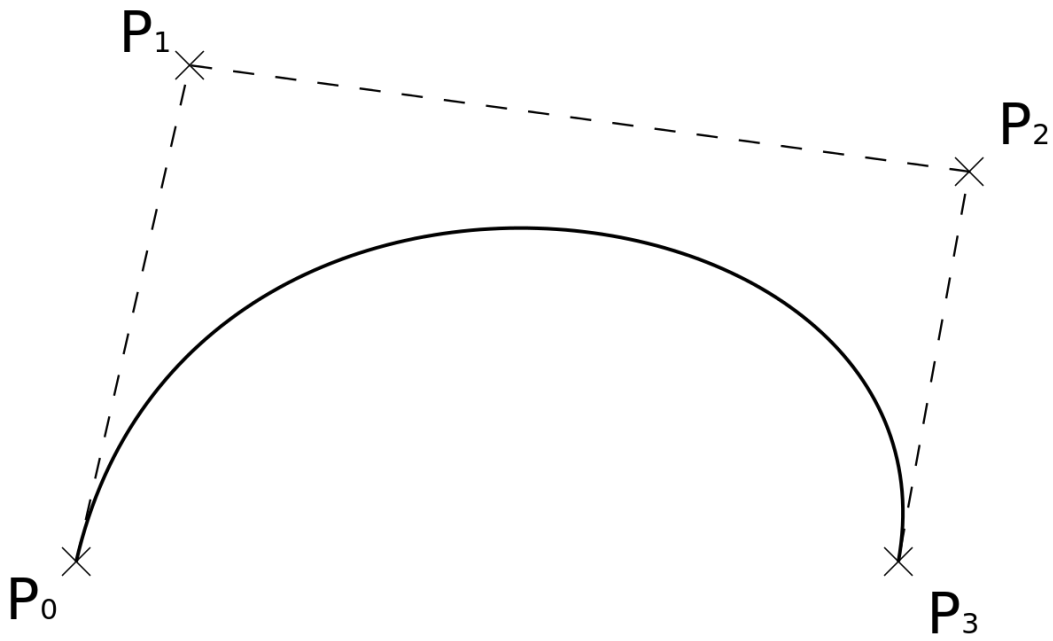


Figura 2.18[44] Ejemplo función aproximación

2.5.2 *Splines*

Las *splines* cúbicas naturales se trata de un caso concreto de interpolación segmentaria cúbica. Cada polinomio que generamos entre dos puntos es de grado 3, esto es:

$$r(t)=a+bt+ct^2+dt^3 \qquad \forall\, 0\le t\le 1$$

Se consideran m+1 puntos para generar las curvas correspondientes a m tramos de la trayectoria que queremos generar. Sin embargo, para poder aplicar este método, tenemos que cumplir una serie de condiciones, una de ellas es que el extremo y el inicio de dos curvas coincidan en el mismo punto, y cabe añadir, que exista continuidad en la primera y segunda derivada en segmento, esto es, que el valor de la derivada tanto primera, como segunda para último punto de un tramo y el primero del siguiente coincida.

Esto es:

Para los m-1 nodos interiores:

1. Continuidad en los segmentos: $r_{i-1}(1)=r_i(0)$
2. Continuidad en la primera derivada: $r'_{i-1}(1)=r'_i(0)$
3. Continuidad en la segunda derivada: $r''_{i-1}(1)=r''_i(0)$

Para los nodos extremos:

1. Valor segunda derivada en el primer segmento: $r''_1(0)=0$
2. Valor segunda derivada en el último segmento: $r''_m(1)=0$ [22][23]

Esto únicamente es válido para todos los puntos menos para el primero y el último que son considerados en la curva, por lo que esto nos deja con menos ecuaciones que incógnitas, concretamente dos menos de las que son necesarias para determinar todos los parámetros necesarios. Por lo que se realiza la siguiente consideración, que es la que distingue el *spline* cúbico natural del *spline* cúbico sujeto, igualamos la segunda derivada de r en ambos extremos a 0.

De esta forma se tiene lo siguiente:

$$r(t)=a+bt+ct^2+dt^3$$
$$r'(t)=b+2ct+3dt^2$$
$$r''(t)=2c+6dt$$

Aquí, como se puede ver lo realmente interesante es poder calcular para cada tramo los coeficientes a, b, c y d de cada curva para generar su correspondiente curva. Si despejamos estos coeficientes para t=0 en cada tramo:

$$a=yi$$
$$b=Di$$
$$c=3(y_{i+1}-y_i)-2D_i-D_{i+1}$$
$$d=2(y_i-y_{i+1})+D_i+D_{i+1}$$

Pero aquí se ve que surge otro inconveniente, y es que el valor de las derivadas en los puntos comunes en los tramos tampoco es conocido. Despejamos entre sí los coeficientes:

Partimos de que tenemos continuidad en la segunda derivada entre los distintos tramos:

$r''_{i-1}(1)=r''_i(0)$, a partir de aquí:

$$2c_{i-1}+d_{i-1}=c_i$$

$$2\left[3\left(y_i-y_{i-1}-2D_i\right)\right]+6\left[2\left(y_{i-1}-y_i\right)+D_i+D_{i-1}\right]=2\left[\left(y_{i+1}-y_i\right)-2D_i-D_{i+1}\right]$$

Desarrollando y reorganizando términos, nos queda:

$$D_{i-1}+4D_i+D_{i+1}=3\left(y_{i+1}-y_{i-1}\right)$$

Y queda un sistema de n ecuaciones con n incógnitas en el que éstas son las derivadas de los extremos de los segmentos.

$$\begin{bmatrix} 2 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & 4 & 1 & 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & 1 & 4 & 1 & 0 & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & 0 & 1 & 4 & 1 & 0 \\ \dots & \dots & \dots & \dots & \dots & 0 & 1 & 4 & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ \dots \\ D_{m-2} \\ D_{m-1} \\ D_m \end{bmatrix} = \begin{bmatrix} 3\left(y_1-y_0\right) \\ 3\left(y_2-y_0\right) \\ 3\left(y_3-y_1\right) \\ \dots \\ 3\left(y_{m-1}-y_{m-3}\right) \\ 3\left(y_m-y_{m-2}\right) \\ 3\left(y_m-y_{m-1}\right) \end{bmatrix} \quad [22]$$

Sin embargo, como se puede observar, es necesario para el cálculo de este sistema determinar el valor de la inversa de la matriz. Esto si es con una matriz de reducida dimensión no causa demasiado problema, pero, cuando se empieza a tener matrices mayores de 4x4 puede resultar en cálculos muy difíciles y que consuman mucho tiempo computacional, cosa que puede provocar que nuestro robot pierda capacidad de respuesta o que esta reacción se produzca demasiado tarde, por lo que se opta por implementar el método del pivote, que nos permite el cálculo de las derivadas que se necesitan de forma óptima y sin mucho consumo, tanto de tiempo como de recursos.

2.5.2.1 Método del pivote

Se parte del sistema anterior, y el objetivo es dejar la diagonal principal de la matriz de coeficientes a uno, y para esto, en primer lugar, dividimos toda la primera fila entre dos, de forma que ya tenemos un uno en el primer dígito, y posteriormente multiplicamos esta columna por el coeficiente correspondiente y se resta a la columna anterior de forma que nos queden unos en la diagonal principal. Esto, como es natural también afecta al resto de la matriz y al vector de términos independientes.

El resultado de la aplicación del algoritmo es:

$$\begin{bmatrix} 1 & e_0 & & & & & & \\ & 1 & e_1 & & & & & \\ & & 1 & e_2 & & & & \\ & & \cdot & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \cdot & & \\ & & & & & & 1 & e_{m-1} \\ & & & & & & & 1 \end{bmatrix} \begin{bmatrix} D_0 \\ D_1 \\ \cdot \\ \cdot \\ \cdot \\ D_{m-1} \\ D_m \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \cdot \\ \cdot \\ \cdot \\ f_{m-1} \\ f_m \end{bmatrix}$$

Donde:

$$e_0 = 1/2$$
$$e_i = 1/(4 - e_{i-1})$$
$$e_m = 1/(2 - e_{m-1})$$

$$f_0 = \frac{3}{2}(x_1 - x_0)$$
$$f_i = (3(x_{i+1} - x_{i-1}) - f_{i-1}) * e_i$$
$$f_m = (3(x_m - x_{m-1}) - f_{m-1}) * e_m \quad [22]$$

Cabe destacar que estas derivadas que se calculan aquí, sólo es necesario hacerlo una vez, mientras que para cada tramo necesitamos calcular los coeficientes de la curva ya que vienen en función tanto de los puntos como de las derivadas que están asociados a este tramo.
En este caso este algoritmo ha de aplicarse de forma vectorial ya que se trabaja en un espacio bidimensional, y por tanto necesitamos en todo momento el valor de la componentes (x,y), pero esto lo único que supone es que se ha de implementar todo lo mencionado anteriormente dos veces. También es fácilmente extrapolable a casos tridimensionales.
Sin embargo, es menester también mencionar los inconvenientes que presentan estas curvas en su uso para trayectorias, y es que no tenemos control sobre las curvas, cualquier modificación hace un cambio en toda la curva, estos cambios no son intuitivos, esto es, no es fácil cambiar la curva para que cumpla lo que buscamos y también puede producir oscilaciones no deseadas.

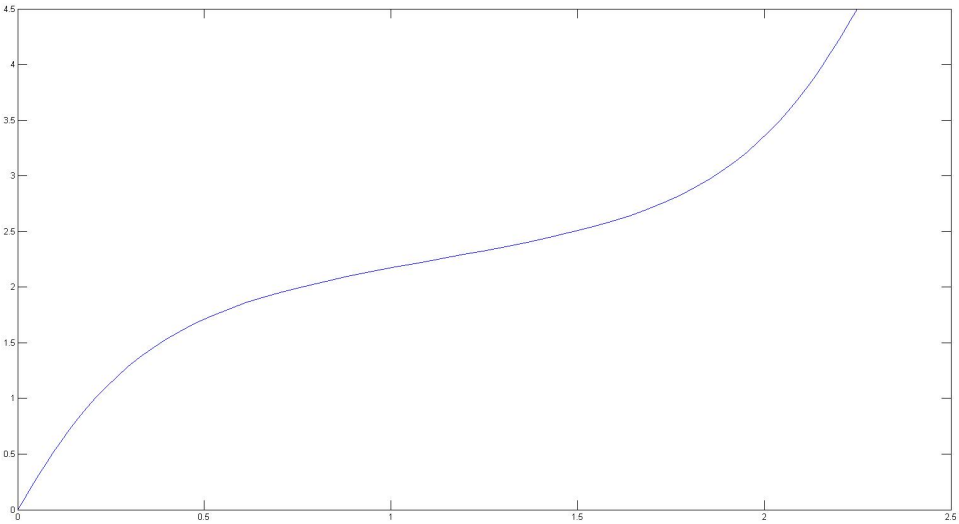


Figura 2.19 *Spline* cúbica natural

2.5.3 Bezier.

En el caso de aproximaciones, se utilizarán las curvas de *Bézier*. Fueron creadas en los años 60 por Pierre Bézier y fueron ampliamente utilizadas por la casa Renault en el diseño de diferentes partes de coche.[25]

La idea de este tipo de algoritmos es unir dos puntos en lugar de por una recta por una curva. Esta curva viene dada por un tercer punto que atrae la línea que une a los dos puntos sin pasar por él, sin embargo, se puede modificar este grado de atracción de la línea al punto si en la ubicación del tercer punto añadimos más.

Realmente, lo que se pretende es interpolar los dos primeros puntos, es decir, que la trayectoria pase necesariamente por ellos, mientras que el resto de puntos son aproximados por la función que resulta de la aplicación del algoritmo.[24]

Para n-1 puntos P_i independientemente de las dimensiones de estos, tenemos la siguiente fórmula:

$$r(t)=\sum_{i=0}^{n} P_i f_i(t)\, t\in[0,1]$$

Siendo $f_i(t)$ la fórmula dada por los polinomios de *Berstein*:

$$f_i(t)=B_{i,n}(t)=\binom{n}{i}t^i(1-t)^{n-i}$$

Si desarrollamos esto para 3 puntos, esto es, n=2:

i=0 (Primer término para P_0):

$$\binom{2}{0}t^0*(1-t)^2=\frac{2!}{0!(2-0)!}*t^0*(1-t)^2=(1-t)^2$$

i=1 (término para P_1):

$$\binom{2}{1} t^1 * (1-t)^1 = \frac{2!}{1!(2-1)!} * t^1 * (1-t)^1 = 2t(1-t)$$

i=2 (término para P_2):

$$\binom{2}{2} t^2 * (1-t)^0 = \frac{2!}{2!(2-2)!} * t^2 * (1-t)^0 = t^2$$

Quedando la ecuación final de la siguiente forma:

$$r(t) = \sum_{i=0}^{n} P_i f_i(t) = (1-t)^2 * P_0 + 2t(1-t) * P_1 + t^2 * P_2 \quad t \in [0,1]$$

Las curvas generadas por esta fórmula no presenta grandes oscilaciones, siguen al polígono de control (polígono generado por los puntos que debe aproximar la curva) y la pendiente en los extremos viene definida por sus segmentos o aristas (en caso de ser cerrado). La modificaciones como en el caso anterior son globales, el cambio de un punto modifica la totalidad de la curva, pero por el contrario estas curvas son mucho más intuitivas a la hora de ser modificada mediante el polígono de control. [24]

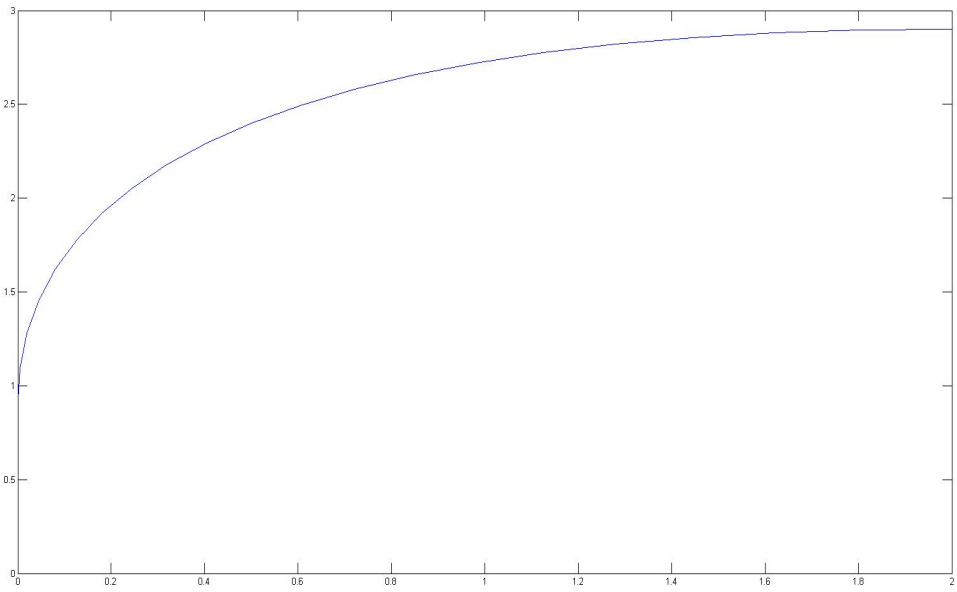


Figura 2.20 *Bezier*

Capítulo 3. Desarrollo práctico

3.1 Presentación del material a utilizar

3.1 Presentación del material utilizado
Para la realización de este proyecto se han utilizado una serie de materiales que serán descritos a continuación:

3.1.1 Ordenador

El ordenador que se ha utilizado para la realización del proyecto se trata de un ordenador portátil Sony Vaio modelo SVF1531B4E. Las características técnicas son las siguientes [26]:

- *"Sistema Operativo: Partición de Windows 8.1 y Ubuntu 14.04*
- *Procesador: Intel Core i7-4500U*
- *Núcleos: 2x1.8GHz*
- *Disco duro: 750GB (140 GB en la partición de Ubuntu)*
- *RAM: 8GB*
- *Tarjeta gráfica: NVIDIA GeForce GT 740M con tecnología NVIDIA Optimus*
- *Pantalla: 15'5" con retroiluminación LED*
- *Puertos USB: 2 x USB 3.0 + 2 x USB 2.0*
- *Conexiones: HDMI, RJ45, audio, micrófono*
- *Conectividad Wireless: 802.11 b/g/n, Bluetooth 4.0 + HS"*

Se ha empleado *Linux*, concretamente, *Ubuntu 14.04* porque, como se ha explicado anteriormente, *ROS* se encuentra disponible únicamente en esta plataforma.
La versión de *ROS* que se ha utilizado para este proyecto es la *ROS Indigo*.

3.1.2 Listado de paquetes de *ROS* a instalar.

Exceptuando los paquetes indicados, todos los paquetes de la siguiente lista se descargarán con el siguiente comando ya que no se realizará ninguna modificación sobre ellos:

```
$sudo apt-get install ros-indigo-nombre-del-paquete
```

Los paquetes son:
navigation
teb_local_planner
gazebo
rviz
rbcar_robot
robotnik_msgs
robotnik_sensors

El resto de paquetes se han de descargar desde la página github, que es la página de repositorios donde los desarrolladores cuelgan los códigos que crean para que todo el mundo pueda acceder a ellos. Para ello se ha de escribir en la terminal, dentro del directorio en el que queremos guardar los ficheros, en este caso, el catkin_ws que hemos creado previamente, en la carpeta src:

```
$git clone "enlace-de-la-página-de-github"
```

Que pueden encontrarse aquí:

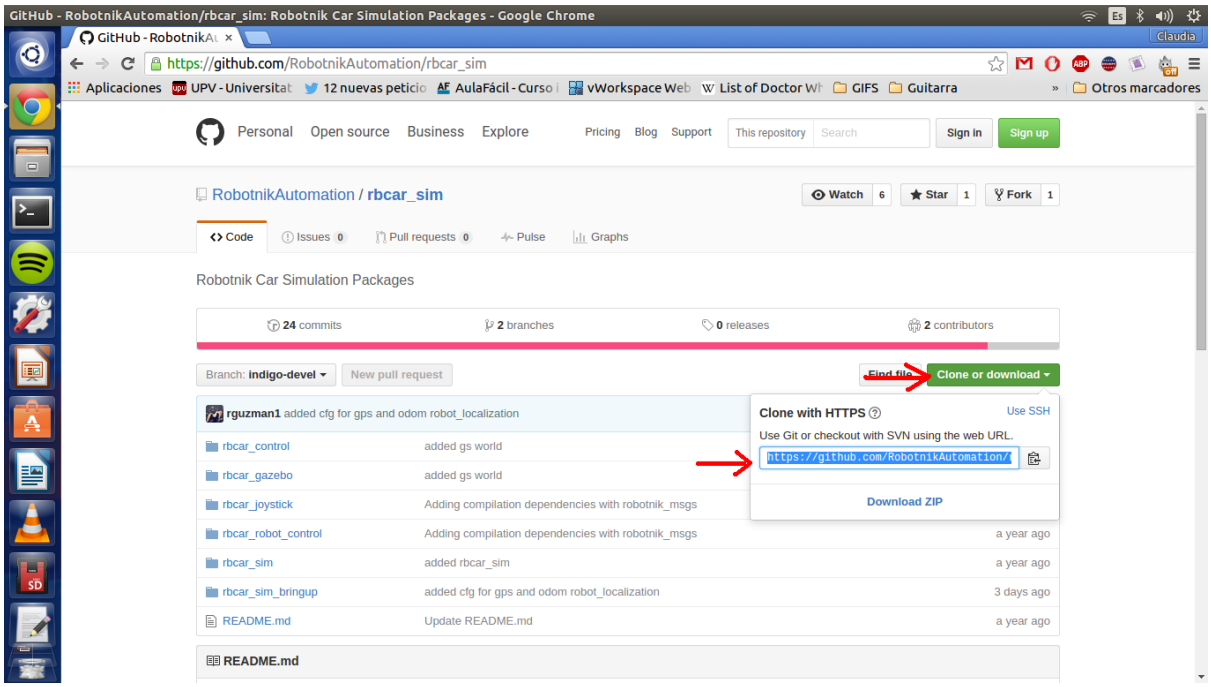


Figura 3.1 Localización paquetes en *github.com*

Los paquetes que hay que descargar de la siguiente forma son:

rbcar_common
rbcar_sim
robotnik_purepursuit_planner

Además, también se han utilizado los paquetes que venían dados por el departamento:

wpts_map
rbcar_wpts
rbcar_robot
rbcar_2dnav

Además del paquete que surge del desarrollo de este proyecto:

rbcar_path

Cabe destacar que, para que todo funcione correctamente, es necesario instalar el siguiente driver:

peak-linux-driver 7.15.2.

Se puede encontrar aquí: http://www.peak-system.com/Details.114+M56e6b5b31af.0.html?&L=1

Y el propio archivo comprimido trae un .pdf en el que explica cómo realizar su instalación.

3.1.3 *Rbcar*

El *rbcar* se trata de un coche eléctrico que ha sido diseñado y creado por la empresa Robotnik y que además de las funciones estándar de cualquier vehículo, como puedan ser transportar personas y mercancías, nos permite enviarle programas o módulos para implementar en él sistemas de conducción autónoma o implementación de sistemas de tele-control.
El *rbcar*, a su vez, nos facilita también la conexión y el control de diferentes tipos de sensores que pueden ser utilizados para la detección de obstáculos, permitiendo evitar accidentes cuando el coche se encuentra en modo automático no tripulado.

Figura 3.2 *Rbcar*

Este robot presenta una configuración *ackermann*, esto es, las ruedas de detrás son las que ofrecen la fuerza de tracción al coche, mientras que las delanteras son orientables y son las que nos permiten establecer la dirección del coche.
También consta de un ordenador y un *router*, cuyo objetivo es permitir la conexión con otros ordenadores y la ejecución de los programas diseñados. Gran variedad de sistemas de conectividad interna y externa, entre las cuales podemos encontrar *USB, RJ45* o toma de *12V DC*.

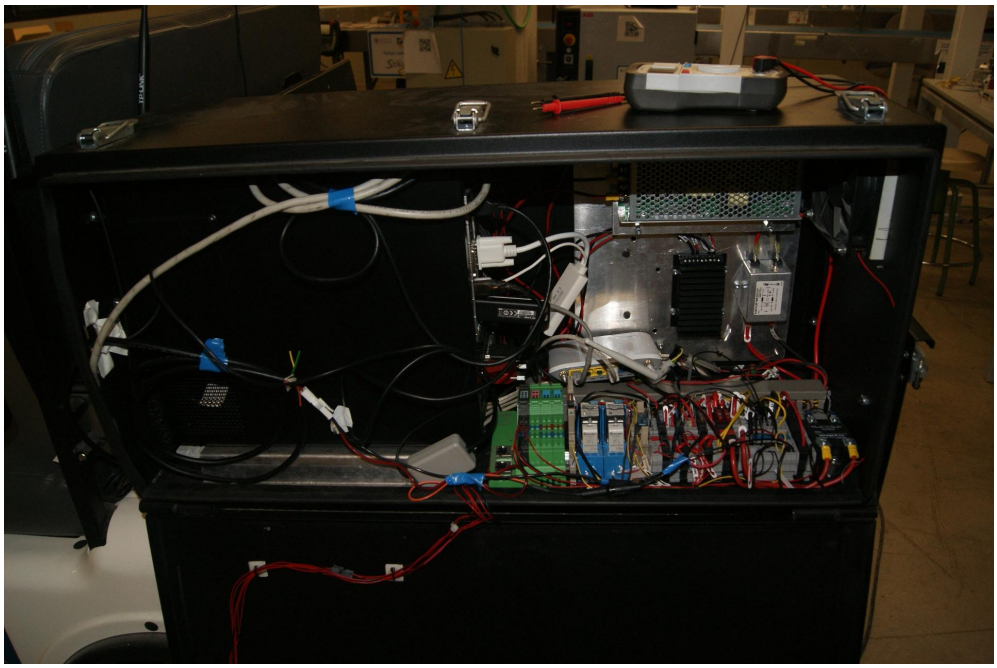


Figura 3.3 Interior ordenador *Rbcar*

Para el desarrollo de aplicaciones para *rbcar*, es necesario utilizar la arquitectura de control abierta y modular de *ROS*, ya que es el sistema que está implementado en nuestro robot.

Figura 3.4 Panel control *Rbcar*

1.Se utiliza para desbloquear las ruedas delanteras.

2. Se trata de la seta de emergencia, ésta debe estar sin pulsar para poder inicializar el coche.

3. El selector de marcha. Cuando está pulsado hacia arriba, el coche avanza; hacia abajo, el coche retrocede y en medio tenemos el punto muerto.
4. El interruptor de los faros del coche.
5. El interruptor de las luces de emergencia, nos enciende los 4 intermitentes.
6.
7. Al girar esta llave a hacia la derecha se activa el control automático del coche.

Las características técnicas, tal y como pueden encontrarse en la página de Robotnik son:
"*Mecánicas:*

- *Dimensiones: 2660 x 1230 x 1720 mm*
- *Peso con baterías: 690 Kg*
- *Caja de carga: 790 x 1100 mm*
- *Capacidad de carga: 2 personas, 150 Kg en caja*
- *Velocidad: 32 km/h*
- *Clase de protección: Galvanizado*
- *Motor: 3,3 kW AC 48V*
- *Autonomía: 70 km*
- *Frenos: Hidráulicos*
- *Indicador de estado de baterías: Sí*
- *Carrocería: ABS Termoformado*
- *Máxima pendiente: 25%*

Control

- *Controlador: Arquitectura abierta ROS PC empotrado con Linux*
- *Comunicación: WiFi 802.11n*
- *Paro de emergencia: Remoto"*[27]

El procedimiento para encender y tener el coche listo para ser utilizado viene explicado en el Anexo III de este documento.

3.1.4 Sensores integrados en el *Rbcar*

Láser: *Sick LMS 291-S05*.
Uno de los sensores que se utilizarán para esta aplicación es el sensor láser *Sick LMS 291-S05*, y se utilizará para la detección de obstáculos durante la conducción autónoma del vehículo.

Figura 3.5 Sensor láser *Sick LMS 291-S05*

El sensor tiene un rango de detección de 180º si mantenemos una resolución angular mayor o igual a 0.5º, siendo las posibilidades 1º o 0.5º, y de 100º cuando queremos una resolución angular de 0.25º.

Otras características técnicas:

- *Rango máximo: 80 m*
- *Temperatura de trabajo: 0ºC a 50ºC (modificable)*
- *Dimensiones máximas: 156x155x186 mm*
- *Voltaje necesario: 24V DC ± 15%*
- *Consumo de potencia < 20W* [28]

A su vez, el *rbcar* también cuenta con un dispositivo de localización por *GPS*. En este caso, el modelo de dispositivo es *IG-500N* de *GBS systems*. Del cual destaca su reducido tamaño, 27 x 30 x 14 mm, y peso, 44 gramos, y su bajo consumo de potencia, 800 mW, lo cual nos permite conectarlo al coche sin reducir de forma drástica su autonomía.[29]

Figura 3.6 GPS *IG-500N*

Y por último, para los cálculos de odometría y facilitar las labores de programación, el coche cuenta con *encoders* en las ruedas, y otro en el volante, lo cual nos permite saber en qué posición se encuentra el mismo a la hora de enviarle órdenes.

3.2 Generación de trayectorias con Matlab

Tras realizar el estudio de los modelos de curva que se van a utilizar para nuestra aplicación, el siguiente paso a realizar es generar una serie de pruebas en el software *Matlab* para diseñar las trayectorias que buscamos con nuestro vehículo.
Para ello, lo que se hará será una serie de funciones en *Matlab*, donde, en función del punto que establecemos como objetivo, se generará la trayectoria.

3.2.1 Estacionamiento en línea

A la hora de estacionar un vehículo, pueden darse dos situaciones. Podemos tener un estacionamiento en línea, para el cual necesitaremos que el coche, a partir de una posición inicial retroceda a la vez que describe una curva, para posteriormente, volver a la orientación inicial mientras continúa retrocediendo para terminar de realizar la acción de estacionamiento.

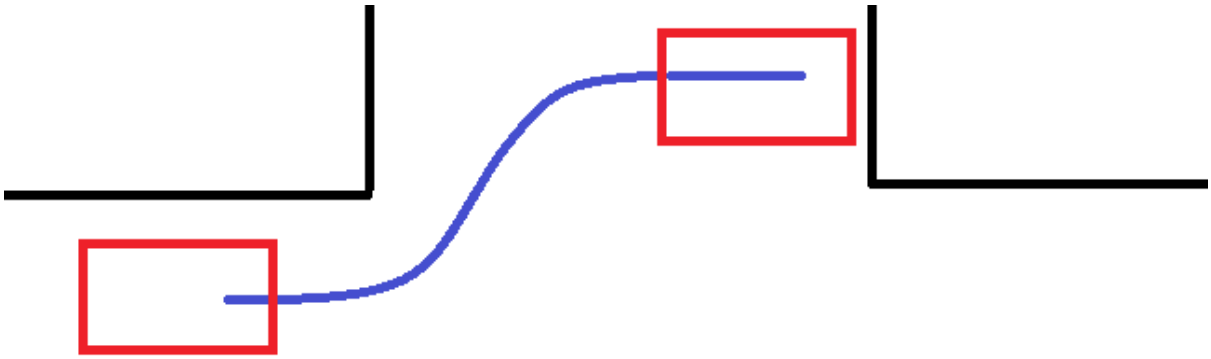


Figura 3.7 Trayectoria deseada estacionamiento en línea

Para el estacionamiento en línea, basados en la forma de las curvas que queremos describir, realizaremos un estudio comparativo entre los dos modelos de curvas expuestos anteriormente, la *spline* cúbica natural y la *bezier*, que en este caso serán ambas de cuatro puntos, para determinar cuál de las dos se asemeja más a la trayectoria de la imagen superior, que es la deseada.
Para ambos casos pondremos como destino el punto (5, 3) con el fin de realizar un estudio comparativo más efectivo.

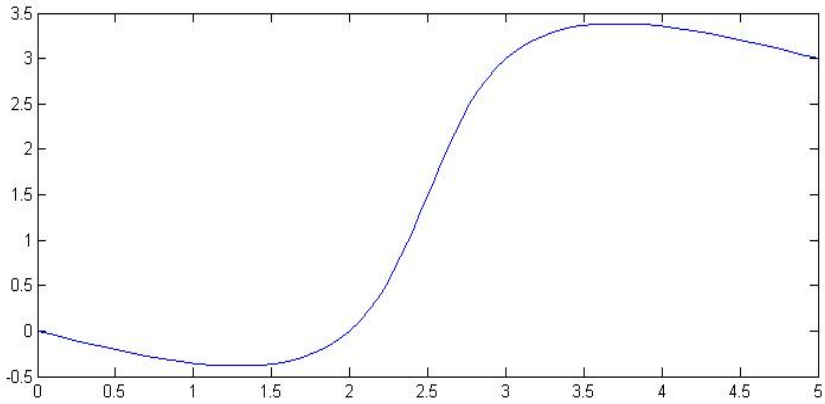


Figura 3.8 Primera prueba *spline* línea

En primer lugar estudiamos la curva que sale como resultado al utilizar las ecuaciones que generan una *spline* cúbica natural. Como puntos de interpolación de esta curva tenemos el origen y el destino, (0,0) y (5,3) respectivamente. Así pues, como se ha mencionado anteriormente necesitamos otros dos puntos que interpolar, y en este caso buscando que el trayecto se ajuste lo máximo a la imagen expuesta anteriormente, seleccionamos puntos que se encuentren en línea recta con origen y destino. Tomamos para punto 2 misma coordenada en Y que el punto inicial, en este caso 0 y que se encuentre a una distancia de ⅖ del punto que consideramos como destino en el eje X. En este caso las coordenadas de este punto 2 serían (2, 0). Lo mismo para el punto 3 pero en este caso, se encuentra a la misma Y y a ⅗ en X que el origen, siendo para este caso concreto (3, 0).

Como se puede ver, la trayectoria se asemeja bastante a lo que buscamos, pero se produce un ligera curva en el tramo final que puede ser problemática a la hora de realizar este tipo de estacionamiento ya que puede producirse una colisión con la pared. Así pues, vemos de cambiar los puntos 2 y 3 para evitar que la curva se salga tanto de la trayectoria deseada.
Para esto, mantenemos la coordenada X de los puntos igual que en el caso anterior y modificamos la Y. Para el punto 2 la ponemos a una distancia de ¼ del destino y en el 3 a ¾.
La trayectoria que se genera es la siguiente

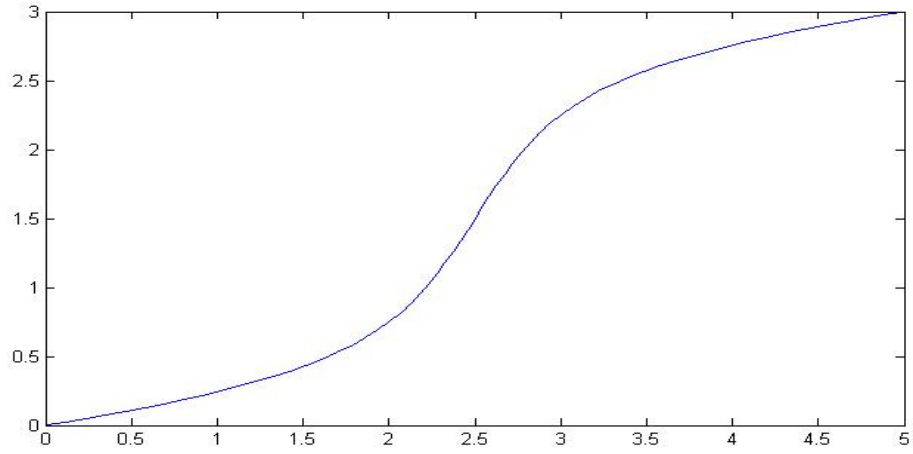


Figura 3.9 Segunda prueba *spline* línea

Lo cual se asemeja bastante más a lo que buscamos en este algoritmo.

Ahora hacemos lo mismo para la *bezier*. Establecemos origen (0,0) y destino (5,3), además de dos puntos auxiliares. En este caso la curva no ha de pasar por estos puntos, los situamos en función de lo que queremos que la curva haga. En este caso serán el punto 2: (2’5, 0) y punto 3: (2’5, 3). El resultado de generar esta curva es el siguiente:

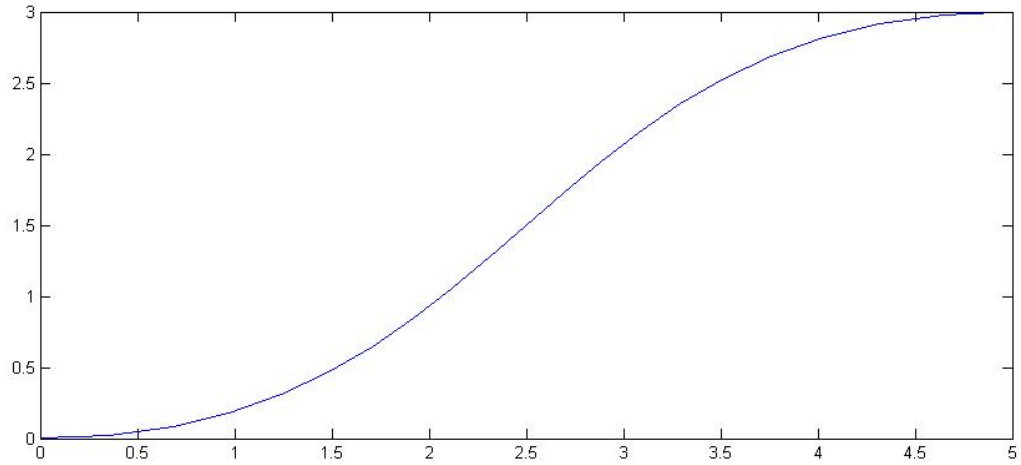


Figura 3.10 *Bezier* línea

En este caso podemos ver que la curva al principio y al final de la misma no nos da el problema de colisionar con los demás elementos que pueden existir en hueco, sin embargo, vemos que la curva es bastante menos pronunciada, y por tanto no podrá realizar la maniobra que buscamos en huecos que sean más justos para nuestro vehículo.

De esta forma, concluimos que la generación de trayectoria a partir de la *spline* satisface más lo que buscamos para este tipo de aparcamiento, así pues, implementaremos esta en nuestro simulador y posteriormente en nuestro programa.

3.2.2 Estacionamiento en batería

Para el estacionamiento en batería buscamos que el vehículo sea capaz de retroceder mientras describe una curva, partiendo de una orientación inicial de, suponemos, 0 grados, para acabar con una de 90 grados.

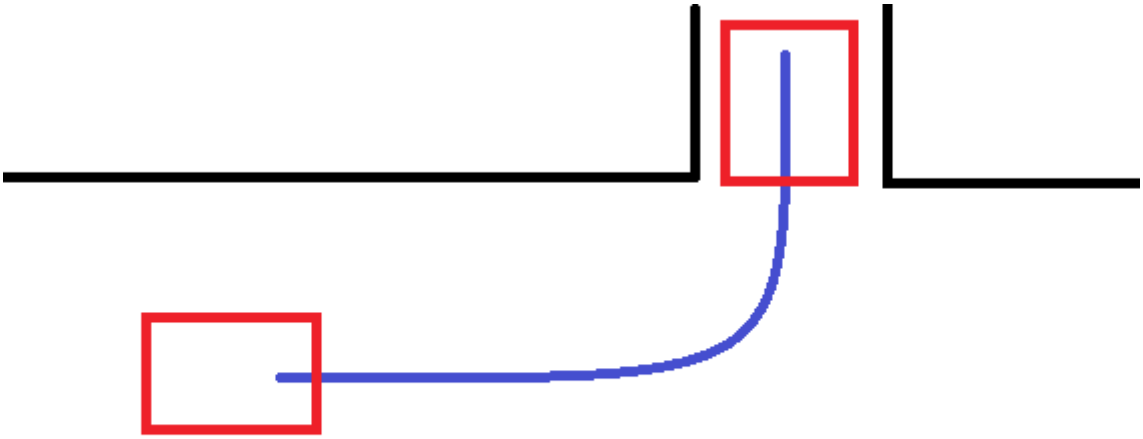


Figura 3.11 Trayectoria deseada estacionamiento en paralelo

Como hemos hecho para el caso anterior, realizamos un breve estudio comparativo entre los dos tipos de curva que hemos seleccionado para la aplicación y seleccionamos el más apropiado para el estacionamiento en batería. En este caso, las curvas únicamente están generadas a partir de 3 puntos.

En primer lugar generamos la *spline*. Volvemos a establecer como punto de inicio el origen de coordenadas, y en este caso como punto final el (6, 6). Sin embargo, como este tipo de estacionamiento, en parte, depende de la longitud del coche, hemos establecido que para coches de longitud menor a 3’5 metros, el destino se considerará como (6, 6+(3’5-longitud)). Esta condición se cumple para nuestro vehículo, ya que su longitud es de 2’6, y por tanto, el punto destino que consideramos es (6, 6’9).
Por último queda situar el punto auxiliar por el que pasará la trayectoria que generemos. Esto para la *spline* nos plantea un problema, porque en función de donde lo situemos la curva será más pronunciada o no, y el coche podrá realizar esta trayectoria con mayor facilidad o no.

Para este caso establecemos el punto a ¼ del destino en X y a ¾ en Y, siendo, pues, las coordenadas de este punto auxiliar (1’5, 5,4).
El resultado de la generación de esta curva es el siguiente:

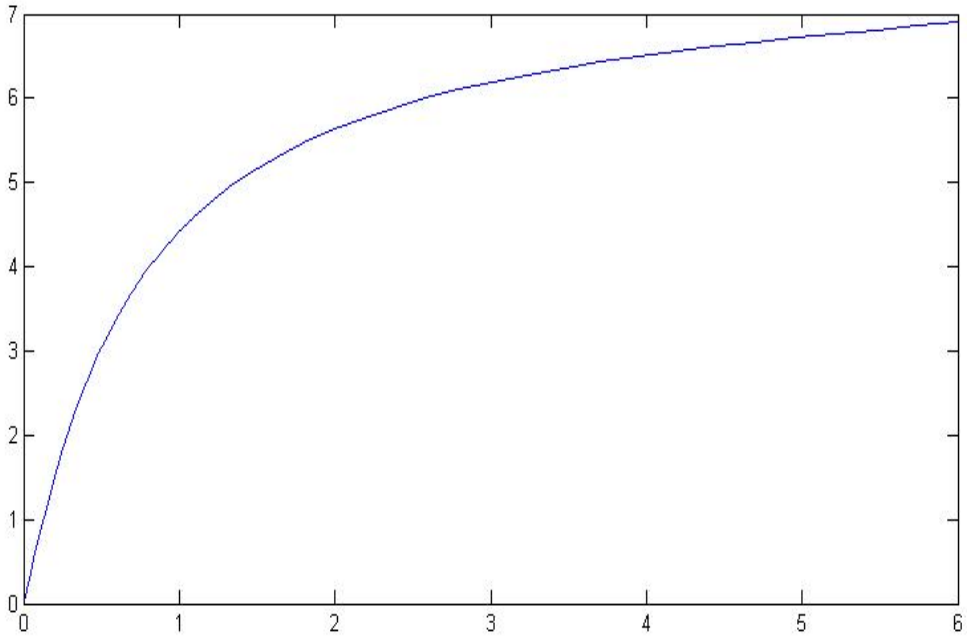


Figura 3.12 *Spline* paralelo

Esto nos plantea el problema de la orientación que asume el coche tanto al origen como en el destino, ya que, como se ha explicado previamente, se busca partir de una orientación de 0 grados y finalizar la trayectoria con una de 90, además del ya mencionado problema de situar el punto auxiliar a la hora de calcular esta trayectoria.

Y ya por último estudiamos la aplicación de la *bezier* de 3 puntos para el estacionamiento en paralelo. Como hemos hecho previamente, establecemos el origen en (0,0) y el destino en (6, 6’9) manteniendo los mismos motivos expuestos en la generación de la *spline*. En este caso, establecer el punto auxiliar es mucho más intuitivo porque, como puede verse en la foto de la trayectoria que buscamos, queremos una curva que pueda quedar contenida en un rectángulo de alto 6’9 y de ancho 6, así pues, establecemos el punto auxiliar a la misma coordenada de Y que el destino, 6’9 y la misma coordenada de X que el origen de la curva, en este caso 0.
El resultado de generar la trayectoria es el siguiente:

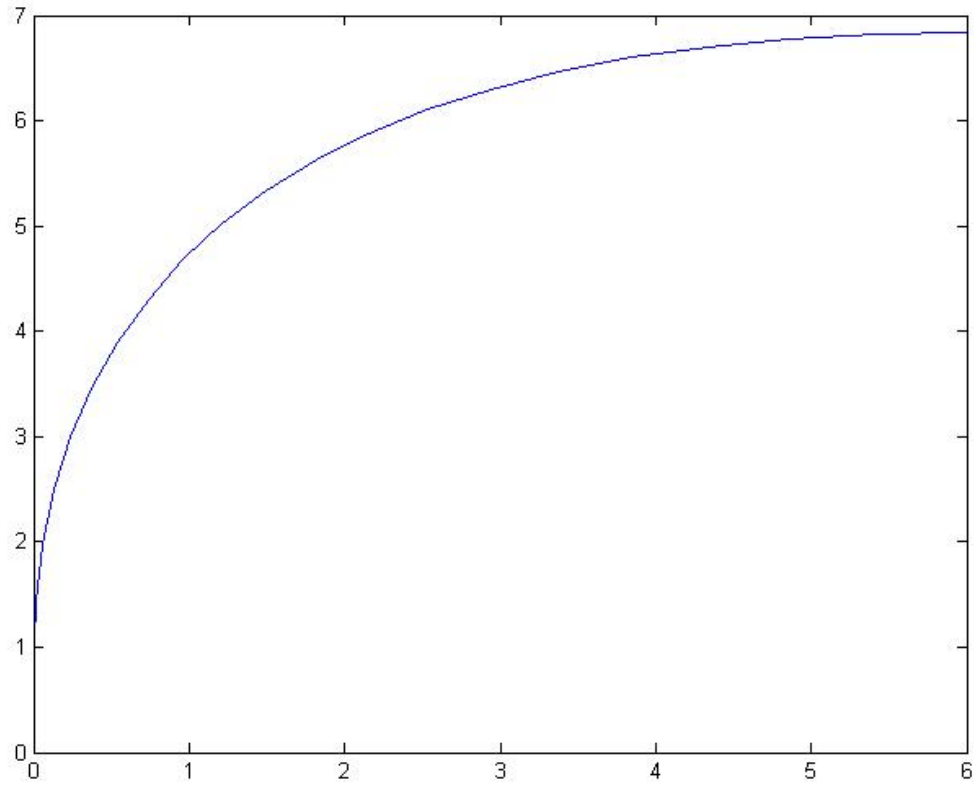


Figura 3.13 *Bezier* paralelo

La generación a partir de este algoritmo nos solventa los problemas que encontramos a la *spline*. Nos permite partir y llegar con las orientaciones deseadas y nos facilita mucho la tarea de situar el punto auxiliar.

Por lo expuesto anteriormente, es lógico que para este tipo de estacionamiento descartemos la trayectoria generada por la *spline* y nos quedemos con la *bezier* ya que es la que más se asemeja a la trayectoria deseada y además, al contrario que la *spline*, nos facilita mucho la tarea de cálculo de la curva.

Una vez hecho el estudio y selección de los tipos de curva que serán implementados en nuestro programa para cada tipo de estacionamiento, dentro de nuestras funciones implementamos la función de crear los ficheros *.txt* que utilizaremos luego en la simulación.

3.3 Simulación en *Rviz* y *Gazebo*.

Como ya se ha explicado en el desarrollo teórico de este documento, en el apartado 2.4, los simuladores son una parte muy importante de la robótica, y antes de realizar cualquier prueba sobre el robot real, es necesario ver si nuestros programas funcionan de una forma satisfactoria sobre una simulación.

3.3.1 Paquetes necesarios

Así pues, y para ser capaces de estudiar el comportamiento del coche frente a las trayectorias que hemos generado, en este proyecto se ha utilizado una serie de paquetes para realizar esta simulación. Los paquetes utilizados son los siguientes:

rbcar_common
rbcar_sim
robotnik_purepursuit_planner

3.3.2 Consideraciones previas a la simulación

Antes de realizar la simulación, necesitamos un fichero llamado *waypoints.txt.* La mejor opción para la generación de este fichero es en las funciones que realizamos anteriormente en *Matlab*, añadir las líneas de código necesarias y modificar el tiempo de muestreo a un número mayor para generar un número menor de puntos, ya que si los puntos que generamos están muy cerca unos de otros puede causar problemas en el programa y no funcionar correctamente.
También es importante respetar la forma que debe tener nuestro fichero, ya que si no la respetamos también puede causarnos algún problema.
El formato que debe tener el archivo es el siguiente:

```
p0;p0->0.4;0;0;0.4
p1;p1->0.4;0.9;0;0.4
p2;p2->0.4;5.4;1.5;0.4
p3;p3->0.4;6.9;6;0.4
```

En primer lugar, tenemos el nombre del punto, que tiene la forma p+número empezando por el 0, el origen. Después, tras el punto y coma, tenemos una especie de descripción que nos indica el nombre del punto y la velocidad a la que se va a trasladar de uno a otro. Posteriormente tenemos las coordenadas X; Y y por último la velocidad lineal a la que se moverá hasta este punto.
Otra consideración importante a la hora de trabajar con los simuladores es la forma en la que *ROS* tiene definidos los ejes de referencia. Sobre todo a la hora de generar estos ficheros y que las trayectorias realicen los movimientos deseados.

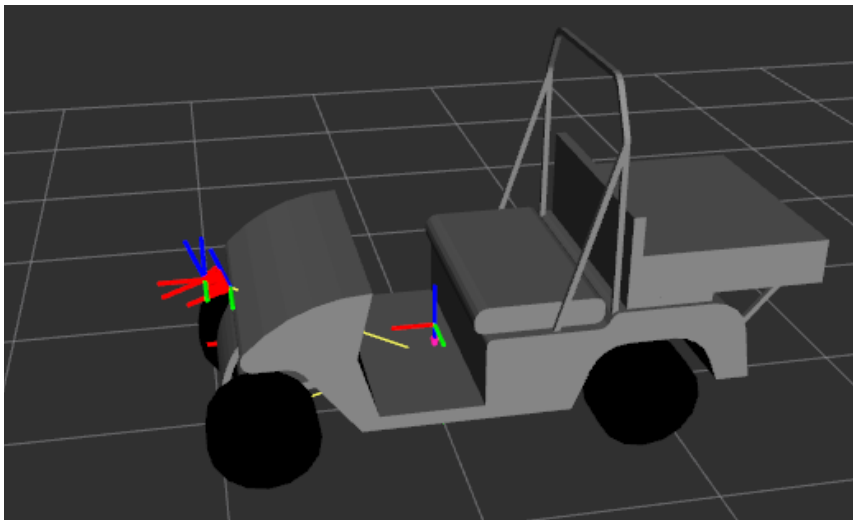


Figura 3.14 Sistema de referencia de *ROS*

Rojo: eje X, coincide longitudinalmente con el coche.
Verde: eje Y.
Azul: eje Z.

3.3.3 Ejecución de la simulación

Una vez tenemos este *waypoints.txt*, tenemos que situarlo en la carpeta correspondiente. Este directorio se encuentra dentro del paquete *robotnik_purepursuit_planner*, en la carpeta *robotnik_pp_planner*, si no lo hemos hecho ya, es necesario que aquí creemos un directorio llamado *config*, donde almacenaremos nuestro fichero de *waypoints.* En este fichero podemos tener varios archivos almacenados, pero únicamente uno con el nombre "*waypoints.txt*" que es el que nuestro programa cargará una vez lo lancemos.

Para lanzar el simulador hay que escribir los siguientes comandos en la terminal:

```
$roslaunch rbcar_sim_bringup rbcar_complete.launch
```

Este *launch* nos carga lo que es el modelo del coche en *Gazebo* y todos los *topics* necesarios para el control del mismo.

```
$roslaunch rbcar_sim_bringup purepursuit.launch
$roslaunch rbcar_sim_bringup purepursuit_marker.launch
```

Estos son los que nos permiten cargar el planificador *purepursuit* en el que simularemos la trayectoria, esto es, nos permite cargar los *waypoints* que nuestro coche seguirá posteriormente en *Gazebo.*

$roslaunch rbcar_description rbcar_rviz.launch
Esto nos carga la configuración del coche en *Rviz* y nos permite controlar los *waypoints* y el movimiento que el coche hará en *Gazebo.*

Al realizar estas instrucciones tenemos abiertos tanto *Rviz* como *Gazebo*. Ahora, para poder cargar los puntos y poder simular correctamente las trayectorias generadas anteriormente tenemos que hacer lo siguiente:
En la ventana de *Rviz*, tenemos que cambiar el sistema de referencia que tenemos como fijo. Por defecto se abre en *base_footprint*, pero para poder cargar los *waypoints* tenemos que cambiarlo a *odom*.

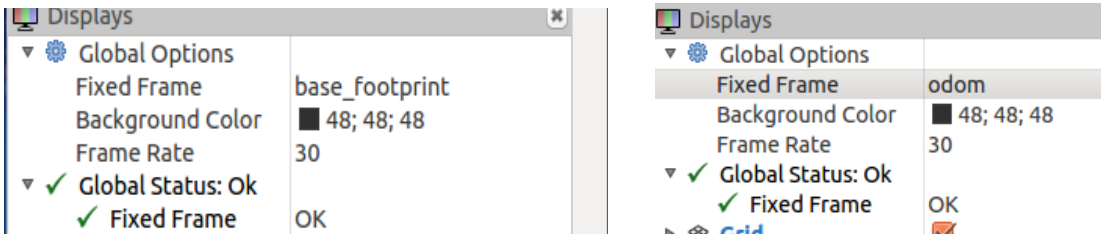


Figura 3.15 Configuración *Rviz* 1

Tras hacer esto, hemos de añadir la herramienta de *InteractiveMarkers*, y para ello, en la parte inferior de este mismo panel tenemos el botón *Add*, que nos abre otra ventana donde, en la pestaña "*By Topic*", la encontramos.

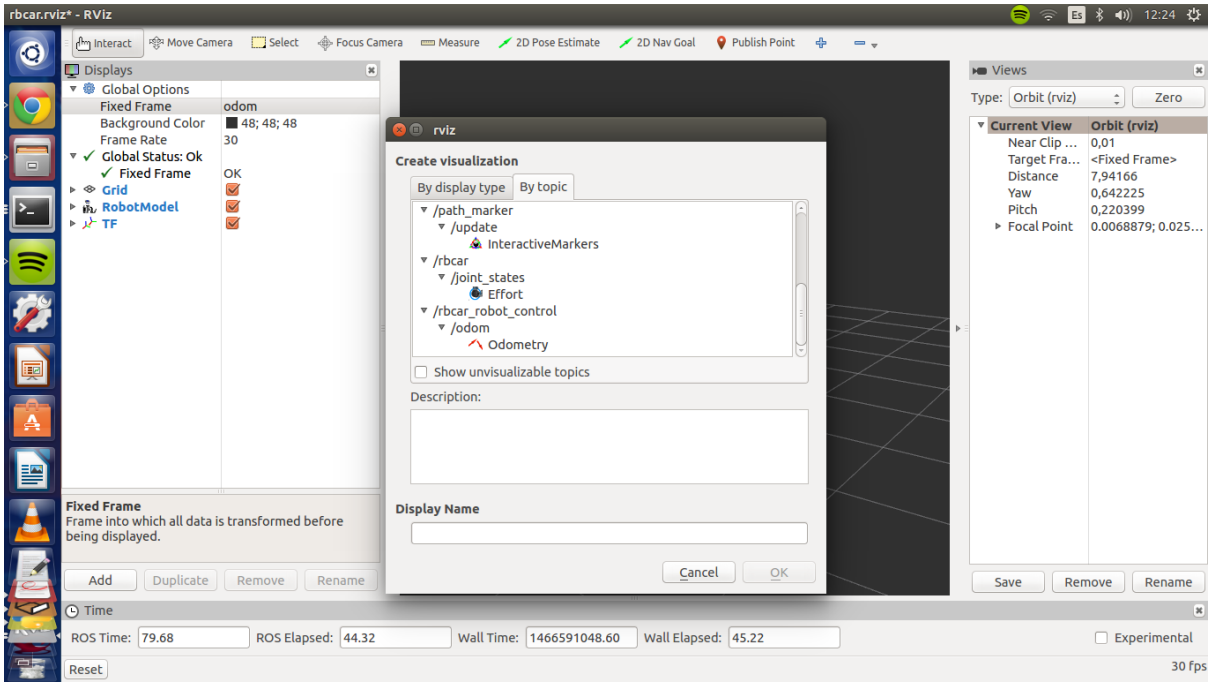


Figura 3.16 Configuración *Rviz* 2

Al seleccionar *InteractiveMarkers*, aparece una pica roja en el origen del sistema de la odometría que indica que se ha cargado correctamente. Si hacemos botón derecho sobre ella, tenemos dos menús desplegables: *Waypoints*, donde tenemos las opciones: *Create new* (y seleccionamos la velocidad a la que se moverá el vehículo), *Delete last* (elimina el último *waypoint* creado), *Delete all* (los elimina todos), *Save* (guarda en un fichero *.txt* los *waypoints* creados y su posición) y la que nos interesa, *Load* (esta es la función que nos permite cargar el fichero que generamos anteriormente).

Una vez estos han sido cargados, volvemos a hacer *click* derecho sobre cualquiera de las picas, y esta vez seleccionamos el otro menú desplegable, el de *Path:* el cual nos permite seleccionar el sentido en el que vamos a recorrer los waypoints (*Go o Go Back*) o parar el vehículo en algún momento (*Stop*).

Al indicarle al vehículo que realice algún movimiento en esta ventana, si cambiamos al entorno de *Gazebo*, vemos que sigue la trayectoria que se ha indicado al coche por *Rviz* a través de los *waypoints*, y podemos ver, sin problemas, como el coche describe las trayectorias que buscábamos.

El aspecto de la pantalla de *Rviz* y la forma de los waypoints para las dos trayectorias que queremos simular es la siguiente:

Para la *bezier* de estacionamiento en paralelo:

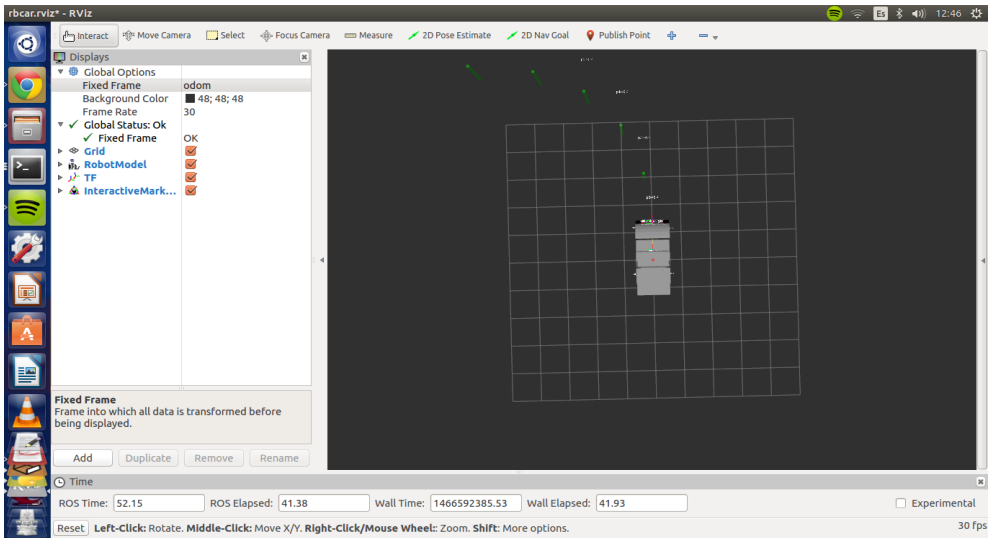


Figura 3.17 *Rviz* simulación *Bezier*

Para spline de estacionamiento en serie:

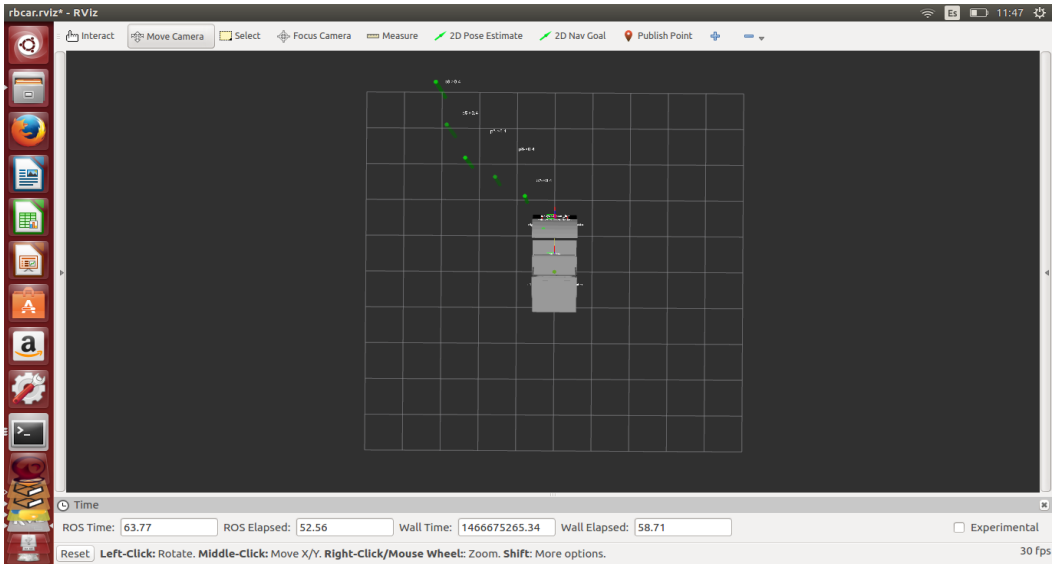


Figura 3.18 *Rviz* simulación *Spline*

3.4 Generación de trayectoria en ROS.

Una vez tenemos el resultado de las trayectorias, y los puntos generados por estas se han simulado correctamente en los programas pertinentes, se procede a la generación las mismas trayectorias directamente a partir de un nodo de *ROS*, que las envía al paquete de waypoints con el fin de alcanzar el punto que establecemos como objetivo.

3.4.1 Creación del paquete.

Lo primero, para poder hacer esta implementación es crear el paquete que contendrá el nodo que realizará la generación de trayectoria. Para esto, en primer lugar hay que situarse en el directorio *src* de nuestro *workspace* y realizar el siguiente comando:

```
$catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

Donde en *package_name* escribimos el nombre de nuestro paquete, y a su lado, separadas por espacios los nombres de los paquetes de los que dependerá el nuestro. Para el proyecto que nos ocupa, se traduce en lo siguiente:

```
$catkin_create_pkg rbcar_path tf std_msgs rospy roscpp std_srvs
```

Tras realizar esto, podremos ver en nuestro *workspace* la carpeta que contendrá todos los archivos que formen parte de este paquete, y dentro del mismo ya podemos encontrar la carpetas *src* e *include*, donde almacenaremos los distintos ficheros *.cpp* y *.h* respectivamente, además del fichero *package.xml* y el *CMakeLists.txt*.
De estos dos archivos, el más importante para nosotros será el *CMakeLists.txt*, ya que para que funcione todo correctamente, será imprescindible hacer una serie de cambios en él, pero antes, entraremos a analizar el nodo.

3.4.2 Nodo

La función básica de nuestro nodo es, a partir de la posición del vehículo que llega al nodo a través de la información de una de las transformadas con las que trabajamos, aplicamos uno de los dos algoritmos, en función del tipo de aparcamiento que queremos realizar y una vez se han generado estos puntos, se envían mediante un servicio al paquete *wpts_map* que es el que se encargará del movimiento del vehículo.

Este código se alojará en el directorio src de nuestro paquete, se encuentra en su totalidad en el Anexo II que se adjunta a este documento, pero procederemos a desglosarlo y comentar las partes que puedan ser más problemáticas:

```
#include <vector>
#include <math.h>
#include <stdio.h>
#include "rbcar_path/ParseWpts.h"
#include <tf/transform_listener.h>
#include "ros/ros.h"
#define long 2.6
```

```
#define PI 3.1416
#define ANGMAX 0.52
```

En primer lugar incluir todas las librerías necesarias para poder ejecutar las funciones que buscamos en nuestro programa, las que más llaman la atención son la de "*ros/ros.h*" que es la que nos permite trabajar con los servicios y funcionalidades que ofrece *ROS* y "*tf/transform_listener.h*" que nos da la capacidad de implementar, como indica el nombre de la propia librería, la funcionalidad de recibir información de las transformadas.
Además, también incluimos *defines*, uno para la constante *PI*, otro en el que introducimos la longitud del vehículo y el último en el que consideramos el ángulo máximo que puede girar (de forma lineal) el coche.

```
using namespace std;
std::vector<float> pto(2),pti(2),p0(2),p1(2),p2(2), a(4), b(4),c(4),d(4), D(4),beta(4),alfa(4), ay(4), by(4),cy(4),dy(4), Dy(4),betay(4),xo(4),yo(4), pf(2);
std::vector<float> y, x, ang;
float t, dt, k;
int i,j,h, w;

float xcoordenate;
float ycoordenate;
float useSpline;
```

Aquí definimos variables y le damos nombre al espacio en el que vamos a trabajar ("*std*").

Ahora entraremos en las funciones que se implementan con nuestro programa. Únicamente se entrará en detalles de lo que realizan los comandos en la primera, puesto que la segunda realiza exactamente lo mismo, pero implementando otro modelo matemático para la trayectoria.

```
bool spline(rbcar_path::ParseWpts::Request  &req,
rbcar_path::ParseWpts::Response &res){
        ROS_INFO("bezier paralelo");
```

Aquí definimos la función. Se trata de una función booleana puesto que, cuando es llamada por el servicio, lo único que tiene que hacer es devolver un *true* o un *false* en función de si esta se ha ejecutado correctamente. Recibe como argumentos (puntero) la solicitud del servicio y la respuesta, si esta ha sido ejecutada correctamente o no.
El comando *ROS_INFO* es el equivalente en *ROS* al *printf*, es decir, nos muestra por pantalla lo que introducimos dentro de ella.

```
if(req.get_wpts){
```

```
tf::TransformListener listener;
tf::StampedTransform transform;

try{
	ros::Time now = ros::Time::now();
	listener.waitForTransform("/odom", "/base_footprint",
now, ros::Duration(3.0));
	listener.lookupTransform("/odom", "/base_footprint",
ros::Time(0), transform);
}
catch (tf::TransformException ex){
	ROS_ERROR("%s",ex.what());
	ros::Duration(1.0).sleep();
}
```

Aquí entramos en el *if* en el momento en el que recibimos la solicitud de servicio (realizada por un programa externo al inicializarse), creamos dos objetos donde almacenar la información que con la estructura *try-catch* pedimos a la transformada. Aquí cogemos la información desde la transformada */odom*, y la trasladamos a /*base_footprint*, que es el sistema con el que trabajamos, el *ros:Time(0)* nos da la primera transformada disponible y lo almacenamos en el objeto *transform*.
El resto de la estructura es para cuando se produce algún error, que nos informe.

```
pti[0]=transform.getOrigin().x();
pti[1]=transform.getOrigin().y();
```

Con esto, tomamos el origen de la transformada que hemos hecho previamente y la almacenamos en un pequeño vector de dos números.
Posteriormente procedemos a realizar todos los cálculos que se han explicado previamente en el capítulo 2.5 de esta memoria, necesarios para la generación de las trayectorias siguiendo el modelo matemático de *bezier* y *spline*.
Se calculan tanto las coordenadas X e Y como la orientación que debe tener el vehículo al llegar a este punto.

```
for (k=0; k<j;k++){
	if ((ang(k+1)-ang(k))>ANGMAX){
		ROS_INFO("NO SE PUEDE ALCANZAR EL PUNTO");
		res.success=false;
		}
}
```

Siendo j el número total de *waypoints* enviado.

Se ha añadido, en ambos casos también, una comprobación de seguridad, en la que si el coche no es capaz de girar, ya que su ángulo máximo de giro es 0.55 radianes (consideramos 0.52 por seguridad), no se realiza el cálculo de la trayectoria ni el envío de *waypoints*.

Para guardar la información dentro de los parámetros que definiremos en el siguiente punto dentro de nuestro servicio, utilizamos la siguiente instrucción:

```
res.orientation.push_back(n);
```

Utilizamos esto puesto que no conocemos el número final de *waypoints* hasta que se han realizado todos los cálculos. Utilizamos el *res.nombre_variable_srv* ya que se trata de un parámetro que requiere nuestro servicio, pero este envío no se hace efectivo hasta que no enviamos la siguiente instrucción:

```
res.success=true;
```

Cuando por algún motivo no se ha recibido la solicitud de servicio o ha saltado el aviso de que el coche no puede alcanzar ese punto debido al ángulo máximo de giro del vehículo esta expresión es igual a "*false*" y el envío no se realiza.

Salimos de estas funciones de cálculo de puntos para ya entrar en lo que es el *main* de nuestra función:

```
int main(int argc, char **argv){
```

Aquí necesitamos estos argumentos para que *ROS* pueda trabajar con los argumentos que se han dado en la línea de comandos.

```
ros::init(argc,argv, "rbcar_path");
```

Esto se escribe para inicializar *ROS* y poder utilizar el resto de componentes del sistema.

```
        ros::NodeHandle nh;
        ros::NodeHandle n;
```

Estos elementos son los que nos permiten la comunicación con el sistema de *ROS*.

```
        ros::ServiceServer service
```

Aquí inicializamos el servicio.

```
if (!n.getParam("/rbcar_path/spline", useSpline))
        {
                useSpline = 1;
        }
```

Con esto, y con las dos estructuras similares que le siguen, establecemos un valor por defecto en las variables a las que les hemos dado un valor en el *.launch* (que vendrá explicado en el punto 3.4.5) en el caso de que estas no se hayan recibido correctamente o no se hayan definido en el *.launch* aún.
Estas variables que vienen dadas en el *.launch* nos definen: qué tipo de aparcamiento vamos a realizar, el valor de la coordenada x del punto de destino y el valor de la coordenada y del mismo, respectivamente.

```
if(useSpline)
        service = nh.advertiseService("get_path", spline);
else
        service = nh.advertiseService("get_path", bezier);
```

Este *if* toma el valor establecido en el *.launch* y selecciona qué función de las anteriores es la que ejecutamos cuando realizamos el servicio.

```
ROS_INFO("Waiting for rosservice call");
    ros::spin();

    return 0;
}
```

Aquí informamos al usuario con el *ROS_INFO* de que estamos esperando a que se produzca la solicitud del servicio y *ros::spin()* lo que hace es permitirnos volver a hacer una llamada al servicio sin tener que matar y reiniciar el nodo cada vez y ya hacemos el *return 0* que finaliza nuestro programa.

3.4.3 Creación del fichero *.srv*

Como se ha explicado previamente, al utilizarse un servicio, necesitamos un archivo *.srv* que es común a los dos paquetes y que envía los datos calculados en nuestro nodo, que actúa como servicio al *wpts_map*, que es el cliente. En nuestro caso se le ha llamado *ParseWpts.srv*. La forma de este fichero es la siguiente:

```
bool get_wpts
---
float64[] latitude
float64[] longitude
float64[] orientation
int8 num_wpts
bool success
```

En la que la variable *get_wpts* es a la que hay que hacer la llamada para que el servicio se active. Esto se produce en el momento en el que se le da como valor "*true*".

Tras los guiones tenemos las variables que nuestro nodo envía tras haberse calculado. Podemos ver que tenemos 3 variables del tipo *float64* que son vectores llamados *latitude, longitude y orientation*. Estos envían a *wpts_map* los valores de Y, X y la orientación del vehículo respectivamente. Después, enviamos también en un entero el número de *waypoints* que tenemos y en una booleana si el servicio se ha producido con éxito o no.

Este fichero se ha de incluir en nuestro paquete dentro de un directorio que ha de crearse con el nombre de *srv*.

3.4.4 Modificación de *CMakeLists.txt*

Una vez creado y conocido el nombre del servicio que vamos a utilizar, es momento de realizar los cambios en el *CMakeLists.txt.* En nuestro caso, las modificaciones de este fichero consisten en descomentar el trozo de código donde tenemos los servicios y añadir nuestro servicio a la lista de forma que de esto:

```
## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

Pasamos a esto:

```
## Generate services in the 'srv' folder
add_service_files(
  FILES
  ParseWpts.srv
#   Service2.srv
 )
```

También hemos de descomentar lo siguiente:

```
generate_messages(
DEPENDENCIES
std_msgs
 )
```

```
[...]
```

```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES rbcar_path
  CATKIN_DEPENDS roscpp rospy std_msgs std_srvs tf
  DEPENDS system_lib
)
```

```
[...]
```

```
target_link_libraries(rbcar_path
   ${catkin_LIBRARIES}
```

```
)
```

Y por último, descomentar lo siguiente y sustituir el nombre del fichero *.cpp* que viene de serie en el *CMakeLists.txt* por el nombre que le hemos dado nosotros a nuestro nodo, en nuestro caso queda de la siguiente forma:

```
## Declare a C++ executable
add_executable(rbcar_path src/bezier.cpp)
```

3.4.5 Creación del fichero *rbcar_path.launch*

Y ya por último, en nuestro paquete, añadimos un último directorio llamado *launch*, y aquí crearemos otro fichero, este llamado *rbcar_path.launch*, el cual nos permitirá lanzar nuestro nodo y cambiar ciertas configuraciones del mismo sin necesidad de ejecutar el *catkin_make* cada vez que queremos cambiar, por ejemplo, de algoritmo o de punto de destino.
La sintaxis de nuestro *rbcar_path.launch* es la siguiente:

```
<?xml version="1.0"?>

<launch>
  <!-- seleccionar tipo de curva a realizar: true para spline, false para bezier -->
 <param name="/rbcar_path/spline" type="bool" value="true" />

  <!-- Enviar el punto de destino -->
      <param name="/rbcar_path/coordenada_x" type="double" value="6.0" />
      <param name="/rbcar_path/coordenada_y" type="double" value="8.0" />

  <!-- launch node -->
    <node name="bezier" pkg="rbcar_path" type="rbcar_path" output="screen"/>

</launch>
```

Desglosando este programa tenemos la etiqueta <*launch*> que ha de incluirse en todos los ficheros de este tipo. Después tenemos los parámetros. En este caso, lo que tenemos es en primer lugar el selector del tipo de trayectoria que tenemos, después tenemos las coordenadas del punto de destino. Estos parámetros al lanzarse el fichero *.launch*, se envían al *roscore* y este los almacena ahí con su valor, éstos, al ser llamados en el programa se envían a las variables elegidas y ya podemos trabajar con ellos de forma normal. Para la utilización de esta etiqueta necesitamos un nombre para el parámetro, el tipo (puede ser *int, double, booleana o string*) y ya por último el valor que queremos que tome.

Después tenemos la etiqueta que nos permite ejecutar el nodo. En ella añadimos el nombre del nodo, el paquete en el que se encuentra, el tipo y por dónde queremos recibir la información.

3.5 Presentación de resultados

Como conclusión a esta parte del documento, se realizará una breve presentación de aquello que se ha conseguido tras la realización de este proyecto.

3.5.1 Gráfico proceso de *ROS*.

En primer lugar, y para entender todo lo que ocurre en el coche, se adjunta un gráfico del proceso que se lleva a cabo en el coche al accionar todos los paquetes necesarios para la navegación y la generación de *waypoints*:

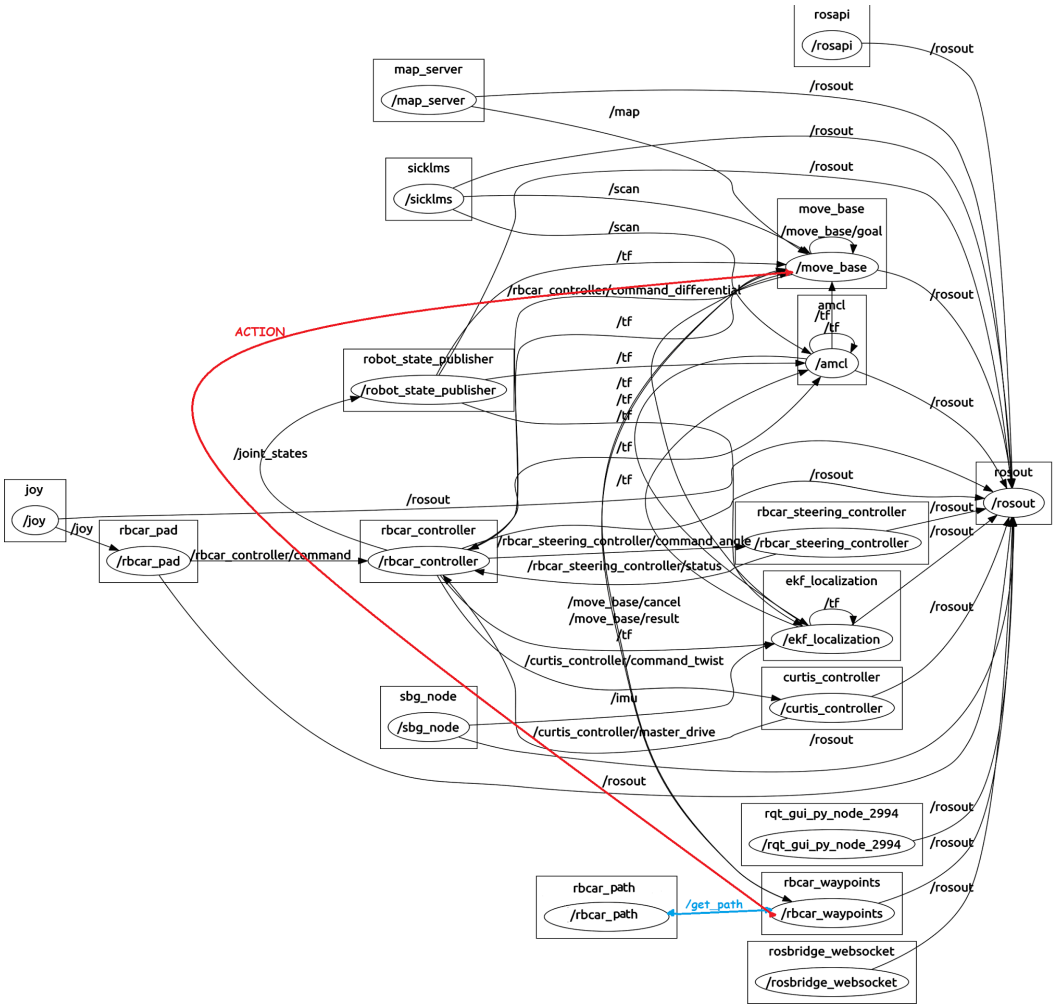


Figura 3.19 Gráfico *Ros* del proceso

Este gráfico puede ser visualizado gracias a la herramienta *rqt_graph* que viene incorporada en *ROS*.

Se ha añadido en este caso, la comunicación mediante *service* y *action*, ya que esta herramienta únicamente muestra la conexión entre nodos vía *topic*.

De todo esto, lo que realmente nos importa es nuestro paquete, *rbcar_path* que se conecta a */rbcar_waypoints* a través del servicio */get_path* una vez se han generado y desde */rbcar_waypoints* son enviados a */move_base*, y de aquí se envían a */rbcar_controller* que es el que se encarga de contactar con los nodos necesarios para producir el movimiento del coche mediante una *action*.

Por último falta cubrir la información de las transformadas que utilizamos dentro de nuestro nodo. Esta no es representada en el gráfico puesto que no se comunica por los métodos convencionales de *ROS*, sin embargo y de forma indirecta también se accede a esta información vía *topic*.

Adjuntamos un gráfico simplificado del proceso en el que se ve lo que se ha descrito anteriormente, para mayor claridad:

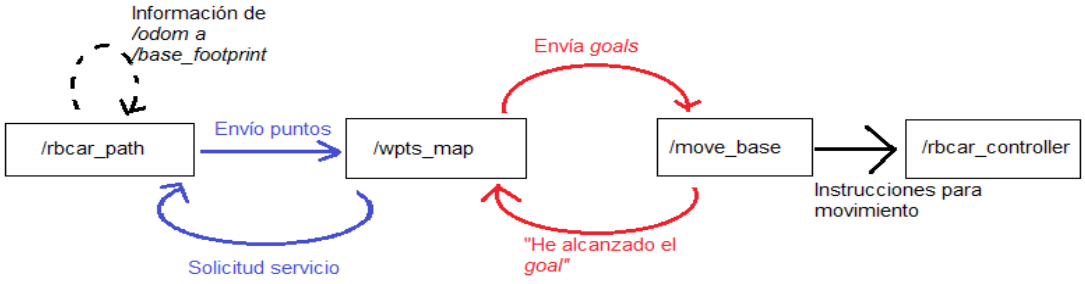


Figura 3.20 Gráfico *ROS* simplificado

Además de esto, se adjunta una captura de pantalla que demuestra que el paquete que ha sido creado envía los puntos a */rbcar_waypoints* y que estos son recibidos de forma correcta:

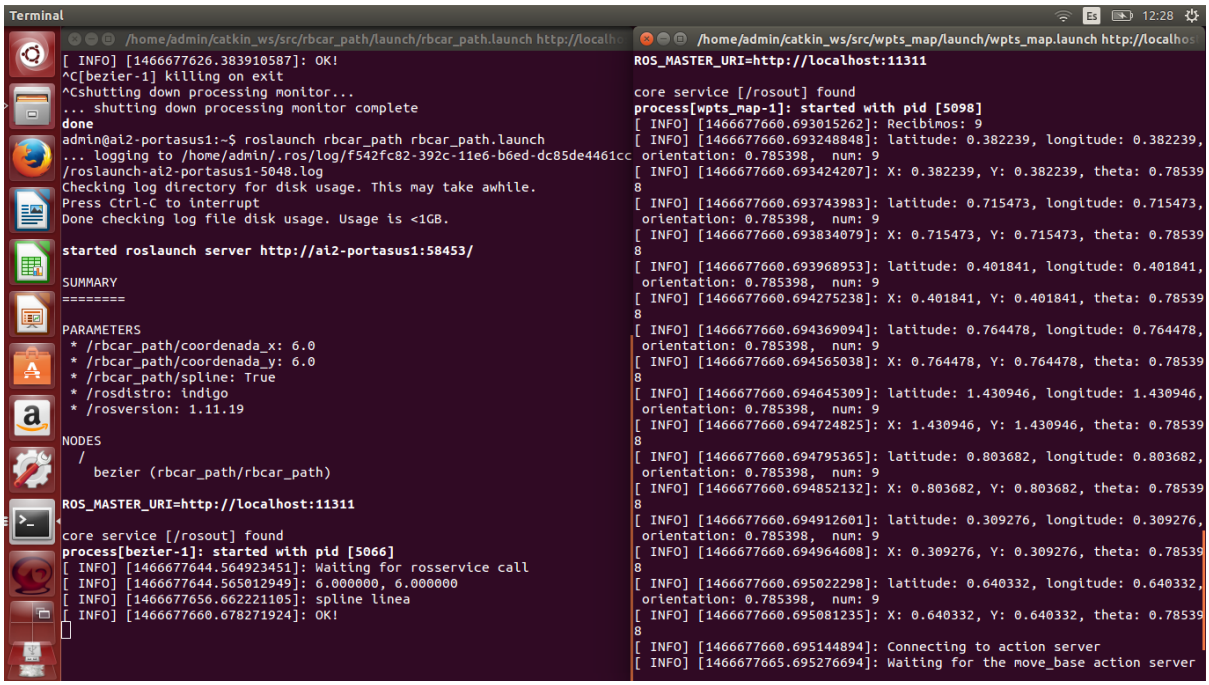


Figura 3.21 Captura de pantalla de proceso *rbcar_path-wpts_map*

3.5.2 Árbol de transformadas

También se adjunta el árbol de transformadas, que se trata de la relación que existe entre todos los sistemas de referencia que encontramos en el vehículo. En este caso, las transformadas como tal, las realizan los nodos que se encuentran representados como las flechas (ramas del árbol) y los sistemas de referencia serían los globos (hojas del árbol).

En nuestro caso, tomamos la información del sistema /odom, del cual tomamos el origen del mismo y lo movemos a nuestro marco de trabajo, */base_footprin*t, porque buscamos que los puntos que ha de seguir sean relativos al anterior que hemos alcanzado.

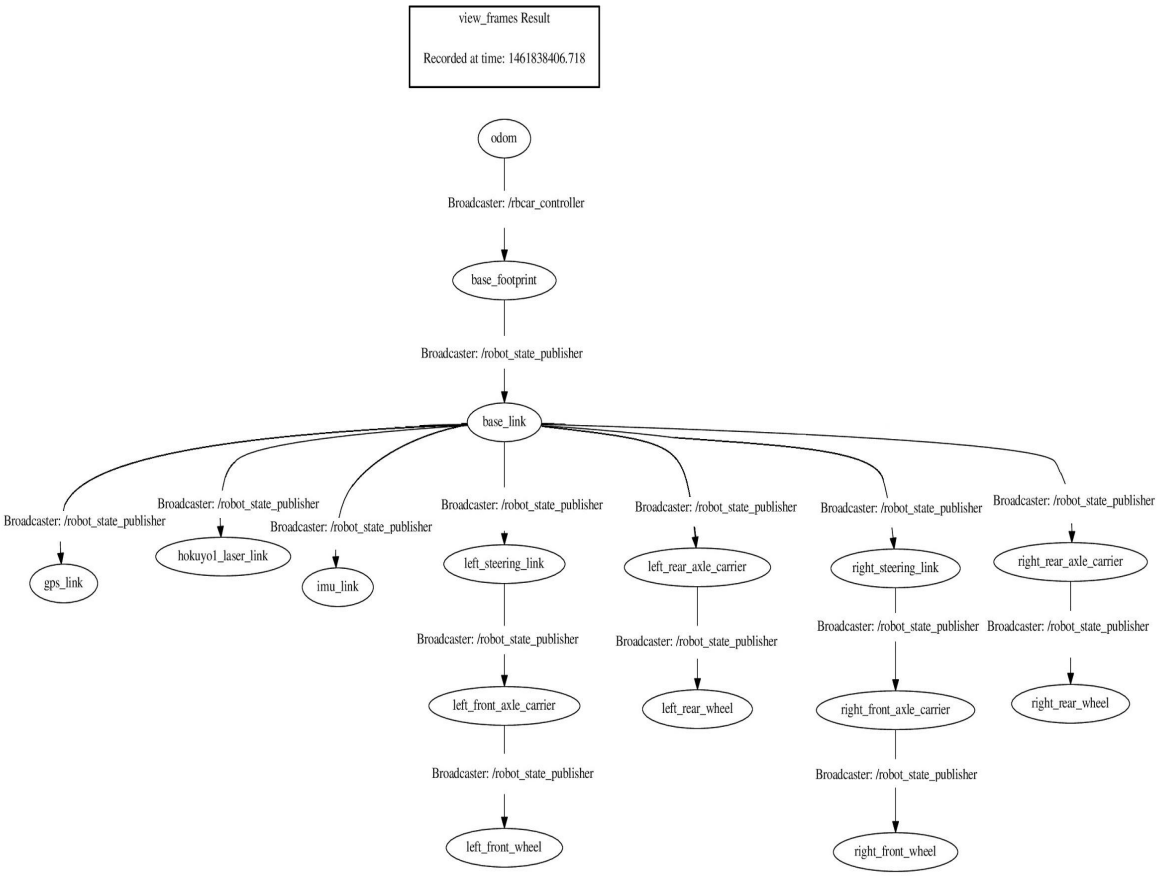


Figura 3.22 Árbol de transformadas proceso *rbcar.*

Capítulo 4. Conclusiones y trabajos futuros

Como conclusión a este documento, revisamos los objetivos que se establecieron al inicio del mismo y se comprobará si estos se cumplen.

- Se ha conseguido dominar y entender a la perfección el entorno de trabajo de *Ubuntu*, se ha comprendido que este *software* ofrece una gran cantidad de herramientas, sobre todo en lo relativo a *software* libre, de código abierto y colaborativo, que no están presentes en el resto de plataformas, y esto se tendrá en cuenta, a partir de ahora para posibles trabajos y proyectos futuros.
- Se ha dominado el entorno de *ROS*. Esto no ha sido fácil de conseguir, puesto que la información a la que se tenía acceso de primera mano (tutoriales de la página del *software*) consideran que previamente se tienen amplios conocimientos informáticos y esto suponía una dificultad añadida. Sin embargo, ha sido posible comprender el funcionamiento del *ROS*, su forma de trabajo, la filosofía y el modo de programación de este *software*. Además, el conocimiento y comprensión de *ROS* abre las puertas a posibles trabajos futuros con otro tipos de plataformas robóticas dada la estandarización en su uso para muchos robots, tanto comerciales como aquellos empleados en investigación.
- Se ha comprendido cómo funciona el sistema de navegación que se encuentra implementado en el *rbcar.*
- Se ha implementado satisfactoriamente el programa que genera la trayectoria de estacionamiento del vehículo.
- La integración del programa de generación de trayectorias con el sistema de navegación se ha realizado con éxito. El nodo que ha sido creado se comunica de forma satisfactoria mediante la utilización de un servicio de *ROS* con el nodo que envía la información necesaria al controlador del coche.

Así pues, podemos decir que se cumplen de forma satisfactoria y correcta todos los objetivos que se propusieron, y por tanto, el objetivo principal de este proyecto que consistía en “generar una trayectoria a partir de una serie de modelos matemáticos con el fin de realizar una serie de maniobras que permitan el estacionamiento del vehículo” ha sido alcanzado.

Como conclusión, se expondrán una serie de posibles proyectos futuros que pueden surgir a partir de la realización de este trabajo.

- Como se ha comentado al inicio de este documento, se han considerado unas hipótesis de trabajo muy concretas para este trabajo, por lo que, queda planteado para el futuro el desarrollar algoritmos que permitan el estacionamiento en el resto de situaciones.

- A su vez, también sería muy interesante realizar una aplicación complementaria a esta, que consiste en la detección de espacios de estacionamiento considerando la longitud del vehículo para aparcamiento en línea y el ancho del mismo para paralelo, a partir de la información que nos viene dada por el láser que está implementado en el coche.

Capítulo 5. Bibliografía

5.1 Documentación

[1] Real Academia Española. Definición de robot [en línea]. http://dle.rae.es/?id=WYRlhzm
[2] Enciclopedia de Clasificaciones. (2016). *Tipos de robots* [en línea]. http://www.tiposde.org/general/460-tipos-de-robots/
[3] Fco. Javier Campoy Garzón. Configuraciones mecánicas [en línea]. http://skiras.blogspot.com.es/2008/05/3ms-iii-configuraciones-mecnicas.html
[4] Colaboradores Muchotrasto.com. Tipos de plataformas [en línea]. http://www.muchotrasto.com/TiposDePlataformas.php
[5] Colaboradores de wikitronica. Robot Omnidireccional [en línea]. http://wikitronica.labc.usb.ve/index.php/Robot_omnidireccional
[6] Robot Operating System (ROS). About ROS [en línea]. www.ros.org/about-ros/
[7] Robot Operating System (ROS). Introduction [en línea]. http://wiki.ros.org/ROS/Introduction
[8] Robot Operating System (ROS). Concepts [en línea]. http://wiki.ros.org/ROS/Concepts
[9] Morgan Quigley, Brian Gerkey & William D. Smart. *Programming robots with ROS.* O'Reilly Media 2013.
[10] Gazebo Simulator. Why Gazebo? [en línea]. http://gazebosim.org/
[11] Robot Operating System (ROS). Rviz [en línea]. http://wiki.ros.org/rviz
[12] Robot Operating System (ROS). tf [en línea]. http://wiki.ros.org/tf
[13] Robot Operating System (ROS). REP 105:Coordinate Frames for Mobile Platforms [en línea]. http://www.ros.org/reps/rep-0105.html
[14] Robot Operating System (ROS). Navigation [en línea]. http://wiki.ros.org/navigation
[15] Robot Operating System (ROS). Move_base [en línea]. http://wiki.ros.org/move_base?distro=indigo
[16] Robot Operating System (ROS). costmap_2d [en línea]. http://wiki.ros.org/costmap_2d?distro=indigo
[17] Robot Operating System (ROS). global_planner [en línea]. http://wiki.ros.org/global_planner?distro=indigo
[18] Robot Operating System (ROS). sbpl [en línea]. http://wiki.ros.org/sbpl_lattice_planner
[19] Robot Operating System (ROS).navfn [en línea]. http://wiki.ros.org/navfn
[20] Robot Operating System (ROS). dwa_local_planner [en línea]. http://wiki.ros.org/dwa_local_planner
[21] Robot Operating System (ROS). teb_local_planner [en línea] http://wiki.ros.org/teb_local_planner
[22] Ángel Valera. Sistemas CAD/CAM. Métodos de interpolación. Diapositivas.
[23] Colaboradores de Wikipedia. Spline [en línea]. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Spline
[24] Ángel Valera. Sistemas CAD/CAM. Aproximación de puntos. Diapositivas.

[25] Colaboradores de Wikipedia. Curva de Bezier [en línea]. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Curva_de_B%C3%A9zier
[26] Fnac.es. Características técnicas Sony Vaio SVF1531B4E [en línea]. http://www.fnac.es/Sony-Vaio-SVE1512E1E-color-blanco-Ordenador-portatil-PC-Portatil/a828793#ficheDt
[27] Robotnik. Características técnicas *rbcar* [en línea]. http://www.robotnik.es/web/wp-content/uploads/2015/09/RBCAR_Datasheet_2016.pdf
[28] Sick. Hoja técnica [en línea]. http://sicktoolbox.sourceforge.net/docs/sick-lms-technical-description.pdf
[29] IG-500N. GBS Systems. Hoja técnica [en línea]. http://www.sbg-systems.com/docs/IG-500N-Leaflet.pdf

5.2 Imágenes

[30] Búsqueda google imágenes “robot configuración ackermann”. http://www.robotnik.es/web/wp-content/uploads/2015/09/RBCAR_2.png
[31] Búsqueda google imágenes “brazo robot”. http://blog.bricogeek.com/img_cms/935-video-construccion-de-un-brazo-robotico.jpg
[32] Búsqueda google imágenes “robot humanoide”. http://www.blogcdn.com/es.engadget.com/media/2011/07/nao-all-terrainaldebaran.jpg
[33] Búsqueda google imágenes “configuración diferencial robot” https://cuentoscuanticos.files.wordpress.com/2011/12/cinematica_diferencial.png
[34] Búsqueda google imágenes “configuración oruga robot” http://i.blogs.es/93566b/talon-01_650/450_1000.jpg
[35] Búsqueda google imágenes “configuración triciclo robot” http://www.muchotrasto.com/images/Tutoriales/RobotMovil/TiposDePlataformas3.jpg
[36] Búsqueda google imágenes “configuración omnidireccional robot” http://www.muchotrasto.com/images/Tutoriales/RobotMovil/TiposDePlataformas6.jpg
[37] Búsqueda google imágenes “configuración síncrona robot” http://www.muchotrasto.com/images/Tutoriales/RobotMovil/TiposDePlataformas5.jpg
[38] Búsqueda google imágenes “configuración ackerman” http://i.blogs.es/6ced87/ackerman/450_1000.jpg
[39] Robot Operating System (ROS). ROS concepts [en línea] http://wiki.ros.org/custom/images/wiki/ROS_basic_concepts.png
[40] Robot Operating System (ROS). move_base [en línea] http://wiki.ros.org/move_base?action=AttachFile&do=get&target=overview_tf_small.png
[41] Robot Operating System (ROS). costmap_2d [en línea] http://wiki.ros.org/costmap_2d?action=AttachFile&do=get&target=costmap_rviz.png
[42] Robot Operating System (ROS). dwa_local_planner [en línea] http://wiki.ros.org/dwa_local_planner?action=AttachFile&do=get&target=local_plan.png
[43] Búsqueda google imágenes “interpolación” https://www.uam.es/personal_pdi/ciencias/barcelo/cnumerico/recursos/serpentina1.gif

[44] Búsqueda google imágenes “función bezier”
https://upload.wikimedia.org/wikipedia/commons/thumb/d/d0/Bezier_curve.svg/300px-Bezier_curve.svg.png

Anexos

Anexo I: Instalación de ROS.

Como información adicional a este proyecto se incluirá un pequeño manual sobre cómo realizar la instalación de ROS Indigo en nuestro ordenador, suponiendo que ya tiene instalado *Ubuntu*.

En primer lugar necesitamos configurar los repositorios de *Ubuntu*. Hemos de ir a la configuración global de nuestro equipo (el icono con el engranaje y la llave inglesa) y aquí seleccionar en la esquina inferior derecha la opción de *Software y Actualizaciones*. Una vez aquí, hay que asegurarse que tenemos marcadas las opciones correspondientes a "*restricted," "universe,"* y *"multiverse."*

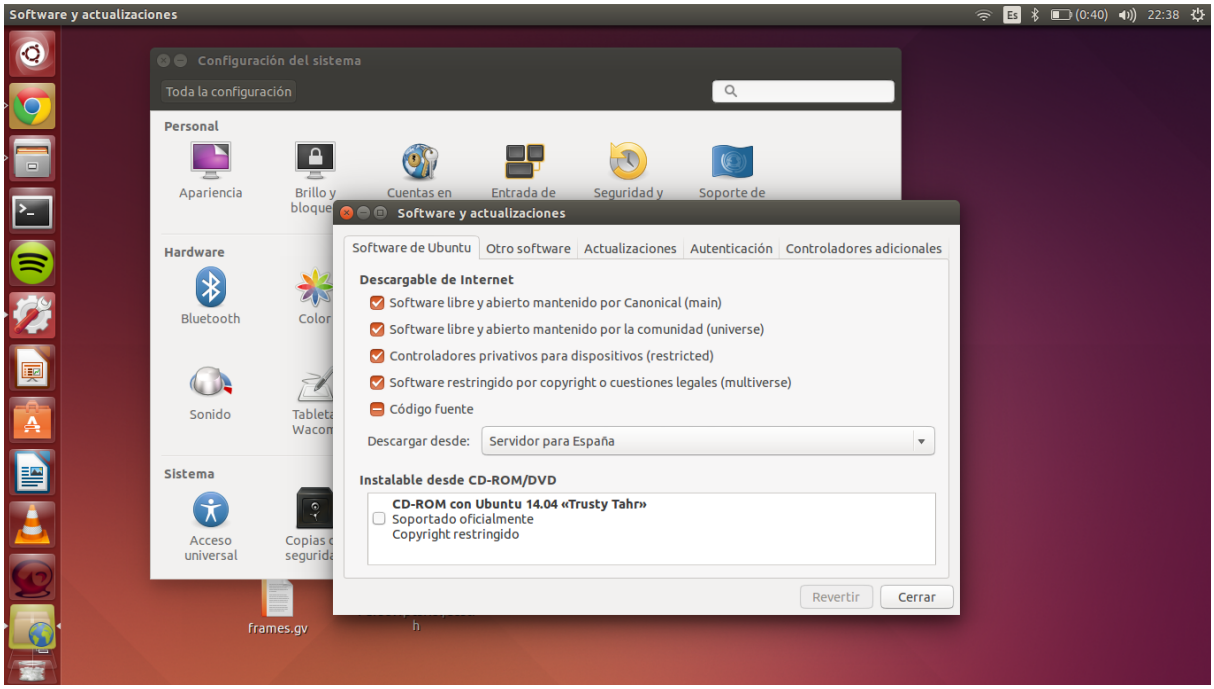


Figura A.1 Configuraciones previas a instalación *ROS*

Ahora, y mediante comandos de la terminal de *Ubuntu* (para abrirla *ctrl+alt+T*), ejecutamos los siguientes comandos:

```
$sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Configuramos nuestro ordenador para poder recibir paquetes de la página *packages.ros.org*.

```
$sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0xB01FA116
```

```
$sudo apt-get update
```

Actualizamos los repositorios

```
$sudo apt-get install ros-indigo-desktop-full
```

Realizamos la instalación de *ROS* completa, con todos sus componentes y funciones.

```
$apt-cache search ros-indigo
```

Con esto actualizamos la lista de paquetes disponibles para nuestro *ROS*.

Ahora, y antes de continuar con la instalación del mismo, debemos inicializar la herramienta *rosdep*, que nos permite instalar dependencias de nuestros paquetes, ya que sin estas nuestros paquetes no compilarán.

```
$sudo rosdep init
$rosdep update
```

Ahora nos interesa realizar la configuración del entorno. Para esto, escribimos el siguiente comando en la consola:

```
$echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$source ~/.bashrc
```

Ya, por último, antes de pasar a configurar nuestro *workspace*, lanzamos un último comando, que nos permite realizar la instalación de los paquetes de forma independiente.

```
$sudo apt-get install python-rosinstall
```

Tras esta instalación, hemos de crear los ficheros *setup.*sh*;

```
$ source /opt/ros/indigo/setup.bash
```

Este fichero, cada vez que realicemos una modificación en *ROS* o abramos una nueva terminal para trabajar deberemos cargarlo con el comando siguiente:

```
$source /catkin_ws/devel/setup.bash
```

2. Creación del *workspace*:

En primer lugar creamos los directorios en los que tendremos nuestro *workspace*:

```
$mkdir -p ~/catkin_ws/src
$cd catkin_ws/src
$catkin_init_workspace
```

Tras esto, y a pesar de no tener nada en el interior del *workspace*, lo “construimos” o compilamos:

```
$cd catkin_ws
$catkin_make
```

(Tras esto, para seguir trabajando hay que volver a hacer el `$source /catkin_ws/devel/setup.bash`)

Anexo II. Código del nodo del paquete *rbcar_path* al completo:

```
#include <vector>
#include <math.h>
#include <stdio.h>
#include "rbcar_path/ParseWpts.h"
#include <tf/transform_listener.h>
#include "ros/ros.h"
#define long 2.6
#define PI 3.1416
#define ANGMAX 0.52


using namespace std;
std::vector<float> pto(2),pti(2),p0(2),p1(2),p2(2), a(4), b(4),c(4),d(4),
D(4),beta(4),alfa(4), ay(4), by(4),cy(4),dy(4), Dy(4),betay(4),xo(4),yo(4),
pf(2);
std::vector<float> y, x, ang;
float t, dt, k;
int i,j,h, w;

float xcoordenate;
float ycoordenate;
float useSpline;

bool bezier(rbcar_path::ParseWpts::Request  &req,
rbcar_path::ParseWpts::Response &res){
        ROS_INFO("Bezier paralelo");
        if(req.get_wpts){

                tf::TransformListener listener;
                tf::StampedTransform transform;

                try{
                        ros::Time now = ros::Time::now();
                        listener.waitForTransform("/odom", "/base_footprint",
now, ros::Duration(3.0));
                        listener.lookupTransform("/odom", "/base_footprint",
ros::Time(0), transform);
                }
                catch (tf::TransformException ex){
                        ROS_ERROR("%s",ex.what());
                        ros::Duration(1.0).sleep();
                }
```

```
            pti[0]=transform.getOrigin().x();
            pti[1]=transform.getOrigin().y();
            x.push_back(pti[0]);
            y.push_back(pti[1]);

            if (long<3.5){
                  pto[0]=pti[0];
                  pto[1]=pti[1]+3.5-long;
                  i=1;
            }
            else {
                  pto[0]=pti[0];
                  pto[1]=pti[1];
                  x.push_back(pti[0]);
                  x.push_back(pti[1]);
                  i=0;
            }

            p0[0]=pto[0];
            p0[1]=pto[1];
            p1[0]=p0[0];
            p1[1]=p0[1]+xcoordenate;
            p2[0]=p1[0]+ycoordenate;
            p2[1]=p1[1];
            t=0;
            dt=0.5;
            while (t<=1){
                  x.push_back((1-t)*(1-t)*p0[0]+2*t*(1-
t)*p1[0]+t*t*p2[0]);
                  y.push_back((1-t)*(1-t)*p0[1]+2*t*(1-
t)*p1[1]+t*t*p2[1]);
                  ROS_INFO("%f, %f", x[i], y[i]);
                  t=dt+t;
                  i++;
            }
            h=0;
            while (h<i-1){
                  res.longitude.push_back(x[h+1]-x[h]);
                  res.latitude.push_back(y[h+1]-y[h]);

                  if (h==0){
```

```
                        res.orientation.push_back(PI/2); //la orientación
respecto a qué eje se hace?
                        //res.orientation.push_back(0);
                }
                else if(h==i-1){
                        res.orientation.push_back(0);
                        //res.orientation.push_back(PI/2);
                }
                else{
                        res.orientation.push_back(atan2((x[h+1]-x[h]),
(y[h+1]-y[h])));
                        //res.orientation.push_back(atan2((y[j+1]-y[j]),
(x[j+1]-x[j])));
                }
                h++;
        }
        for (k=0; k<h;k++){
                if ((ang[k+1]-ang[k])>ANGMAX){
                        ROS_INFO("NO SE PUEDE ALCANZAR EL PUNTO");
                        res.success=false;
                        }
        }
        res.num_wpts=h;
        ROS_INFO("OK!");
        res.success = true;
    }
    else
    {
        res.success = false;
    }

    return true;
}

//SPLINE

bool spline(rbcar_path::ParseWpts::Request  &req,
rbcar_path::ParseWpts::Response &res){
    ROS_INFO("spline línea");

    if(req.get_wpts){

        tf::TransformListener listener;
```

```
		tf::StampedTransform transform;

		try{
			ros::Time now = ros::Time::now();
			listener.waitForTransform("/odom", "/base_footprint",
				now, ros::Duration(3.0));
			listener.lookupTransform("/odom", "/base_footprint",
ros::Time(0), transform);
		}
		catch (tf::TransformException ex){
			ROS_ERROR("%s",ex.what());
			ros::Duration(1.0).sleep();
		}
		pti[0]=transform.getOrigin().x();
		pti[1]=transform.getOrigin().y();
		x.push_back(pti[0]);
		y.push_back(pti[1]);

		i=4; //el número de puntos que utilizamos
		pf[0]=xcoordenate;
		pf[1]=ycoordenate;
		//ptos para generar spline
		xo[0]=pti[0];
		xo[1]=pti[0]+pf[0]/4;
		xo[2]=pti[0]+3*pf[0]/4;
		xo[3]=pti[0]+pf[0];

		yo[0]=pti[1];
		yo[1]=pti[1]+pf[1]/4;
		yo[2]=pti[1]+3*pf[1]/4;
		yo[3]=pti[1]+pf[1];


		//cálculo de la matriz a
		alfa[0]=.5;
		for (j=1; j<=(i-2); j++){
			alfa[j]=1/(4-(alfa[j-1]));
		}
		alfa[i-1]=1/(2-alfa[i-2]);
		//cálculo del vector b
		beta[0]=3*(xo[1]-xo[0])*alfa[0];
		betay[0]=3*(yo[1]-yo[0])*alfa[0];
		for (j=1; j<=(i-2); j++){
```

```
        beta[j]=(3*(xo[j+1]-xo[j-1])-beta[j-1])*alfa[j];
        betay[j]=(3*(yo[j+1]-yo[j-1])-beta[j-1])*alfa[j];
}
beta[i-1]=(3*(xo[i-1]-xo[i-2])-beta[i-2])*alfa[i-1];
betay[i-1]=(3*(yo[i-1]-yo[i-2])-beta[i-2])*alfa[i-1];
//cálculo del vector D
D[i-1]=beta[i-1];
Dy[i-1]=betay[i-1];
for (j=(i-2); j=0; j=j-1){
        D[j]=beta[j]-D[j+1]*alfa[j];
        Dy[j]=betay[j]-Dy[j+1]*alfa[j];
}

for (j=0; j<=(i-2); j++){
        a[j]=xo[j];
        b[j]=D[j];
        c[j]=3*(xo[j+1]-xo[j])-2*D[j]-D[j+1];
        d[j]=2*(xo[j]-xo[j+1])+D[j+1]+D[j];
        ay[j]=yo[j];
        by[j]=Dy[j];
        cy[j]=3*(yo[j+1]-yo[j])-2*Dy[j]-Dy[j+1];
        dy[j]=2*(yo[j]-yo[j+1])+Dy[j+1]+Dy[j];
}
w=0;
for (j=0; j<=(i-2); j++){
        for (t=0; t<=1; t=t+0.33){
                y.push_back(ay[j]+by[j]*t+cy[j]*t*t+dy[j]*t*t*t);
                x.push_back(a[j]+b[j]*t+c[j]*t*t+d[j]*t*t*t);
                w=w+1;
        }
        if (j==(i-2)){
                x.push_back(a[j]+b[j]+c[j]+d[j]);
                y.push_back(ay[j]+by[j]+cy[j]+dy[j]);
        }
}
j=0;
for (k=0;k<=w-1;k++){
        if ((x[k+1]-x[k])>0.1 || (y[k+1]-y[k])>0.1){
                res.latitude.push_back(x[k+1]-x[k]);
                res.longitude.push_back(y[k+1]-y[k]);
                if (k==w-1){
                        res.orientation.push_back(PI/2);
                        //res.orientation.push_back(0);
```

```
                        }
                        else{

                                res.orientation.push_back(atan2((y[k+1]-
y[k]),(x[k+1]-x[k])));
                                ang.push_back(atan2((y[k+1]-y[k]),(x[k+1]-
x[k])));
                                //res.orientation.push_back(atan2((x[k+1]-
x[k]),(y[k+1]-y[k])));
                        }
                        j++;
                }
        }
        for (k=0; k<j;k++){
                if ((ang[k+1]-ang[k])>ANGMAX){
                        ROS_INFO("NO SE PUEDE ALCANZAR EL PUNTO");
                        res.success=false;
                        }
        }
    res.num_wpts=j;
    ROS_INFO("OK!");
    res.success=true;
    }
    else
    {
        res.success = false;
    }

    return true;
}



int main(int argc, char **argv){

    ros::init(argc,argv, "rbcar_path");
    ros::NodeHandle nh;
    ros::NodeHandle n;
    ros::ServiceServer service;

    if (!n.getParam("/rbcar_path/spline", useSpline))
    {
        useSpline = true;
```

```
	}
	if (!n.getParam("/rbcar_path/coordenada_x", xcoordenate))
	{
		xcoordenate=0;
	}
	if (!n.getParam("/rbcar_path/coordenada_y", ycoordenate))
	{
		ycoordenate=0;
	}
	if (useSpline)
			service = nh.advertiseService("get_path", spline);
	else
			service = nh.advertiseService("get_path", bezier);

	ROS_INFO("Waiting for rosservice call");
	ros::spin();


	return 0;
}
```

Anexo III: Guía de inicio del *rbcar*.

1. Encendido y puesta en marcha.

En primer lugar, tomamos como referencia la Figura 25 del documento de la memoria del proyecto, lo que tenemos que hacer es asegurarnos que la seta de emergencia está liberada. Para ello la giramos en el sentido de las agujas del reloj hasta que salta. Una vez hecho esto, introducimos la llave en el contacto y la giramos, como en un coche normal. Posteriormente, vamos al ordenador del coche que se encuentra en el lateral del mismo en el interior de una gran caja negra. Aquí hemos de accionar el contactor de color verde girándolo hacia la derecha y, después, hemos de accionar el botón azul, esto nos permite conectarnos al coche y acceder a los programas que tienen dentro, que son los que nos permiten la navegación del mismo. Por último, hemos de pulsar el botón azul que se ha referenciado en la Figura 25 como número 1 para liberar las ruedas delanteras. Tras esto el coche está listo para circular.

2. Conexión al *rbcar* vía ordenador.

El *rbcar* cuenta con un punto de acceso a su ordenador vía *wi-fi*. Es necesario conectarse a ella antes de establecer el protocolo *ssh* que nos permitirá comunicarnos con el ordenador del coche. Una vez estamos conectados a esta red, introducimos el siguiente comando en la consola:

```
$ssh robotnik@192.168.0.200
```

Acto seguido nos pedirá la contraseña, y una vez introducida estamos conectados con el ordenador del vehículo.

Las contraseñas, tanto de la red *wi-fi*, como para conectarse al coche son: *R0b0tn1K*.

3. Paquetes e interruptores para establecer la navegación autónoma.

Para activar el modo automático en el coche, es necesario bajar el interruptor 6 de la Figura 25, y posteriormente girar a la derecha el interruptor verde que está a la derecha del mismo (7).

Ahora entraremos en los paquetes que hay que lanzar y desde donde hay que hacerlo para lanzar la navegación.

En primer lugar, en la consola en la que hemos establecido el protocolo ssh, lanzaremos lo siguiente:

```
$export ROS_IP=192.1168.0.200
$roslaunch rbcar_bringup all_components.launch&
```

Aquí hemos de esperar a que se inicialice todo correctamente. El & nos permite continuar utilizando la consola aunque tengamos un proceso activo en ella.

```
$roslaunch rbcar_2dnav rbcar_2dnav_real_teb.launch
```

Esto nos lanza el paquete de navegación.

Ahora, y en otra terminal hemos de lanzar lo siguiente:

```
$export ROS_MASTER_URI=192.168.0.50:11311
```

En este comando, el 50 de la dirección *IP* puede variar, para comprobarlo, antes de lanzar este comando, lanzamos *ifconfig* y comprobamos nuestro nombre de *IP*.

```
$export ROS_IP=192.168.0.200
```

Esto ha de lanzarse en cada nueva terminal que abrimos que queramos que se comunique con el coche.

Posteriormente, lanzamos los paquetes que nos generan los puntos y que envían estos puntos al controlador del coche.

```
$roslaunch rbcar_path rbcar_path.launch
```

Y en una nueva terminal:

```
$roslaunch wpts_map wpts_map.launch
```

Se recomienda, sin embargo, lanzar todos los programas antes de activar el modo automático para mayor seguridad.

PRESUPUESTO

Presupuesto

1. Necesidad del presupuesto

Antes de entrar en el presupuesto propiamente dicho, nos vemos en la necesidad de justificar la existencia del mismo. Es necesario cuantificar, tanto la mano de obra que se ha invertido en este proyecto, como realizar un inventario de los materiales utilizados para el mismo. A pesar del carácter educativo del documento, todo documento de un proyecto de ingeniería ha de concluirse con un presupuesto, ya que si en el futuro se buscara la reproducción del mismo, ya sea en parte, o en su totalidad, se tiene que conocer la dispensa económica que esto supondría.

Este presupuesto ha sido elaborado a partir del documento "RECOMENDACIONES EN LA ELABORACIÓN DE PRESUPUESTOS EN ACTIVIDADES DE I+D+I" del Centro de Apoyo a la Innovación, la Investigación y la Transferencia de Tecnología (CTT), en su revisión para el año 2015. La elección de este documento como forma de este presupuesto en lugar del modelo habitual de capítulos y unidades de obra es debido a que al tratarse de un proyecto de investigación, se consideraba más adecuado.

2. Estudio económico

2.1 Coste de personal

Para el cálculo de los costes de personal se ha considerado la siguiente fórmula:

$$Coste\,de\,personal\,(€) = Coste\,por\,hora\,(€/h)\;x\;Número\,de\,horas\,dedicadas\,(h)$$

En este coste por hora ya se considera los gastos correspondientes a Seguridad Social, Indemnizaciones y costes indirectos.

Para el cálculo del coste por hora, se ha considerado del documento citado anteriormente, en la Tabla 2 que nos ofrece información respecto al personal eventual, tomamos como referencia el salario correspondiente a titulado medio. Esta tabla, sin embargo, nos da dos valores, de salario máximo (36,2 €/h) y mínimo (26,8€/h), por lo que para este presupuesto se ha optado por calcular el valor medio entre ambos, 31,5€/h.

Acción	Tiempo (h)	Coste	Total
Instalación partición Ubuntu 14.04	5	31,5	157,5
Instalación ROS	20	31,5	630
Generación de trayectorias en Matlab	40	31,5	1260
Simulación de trayectorias	25	31,5	787,5
Creación del paquete	100	31,5	3150
Simulación en Rbcar	30	31,5	945
Redacción y montaje de documentos	80	31,5	2520
Total	300		9450

Desglosando esto con los porcentajes y costes correspondientes a Seguridad Social (32,1%), Indemnizaciones (3,04%) y costes indirectos (16,5 €/h), nos queda lo siguiente:

Concepto	Precio (€)
Honorarios	1179,27
Seguridad social (32,1%)	3033,45
Indemnización (3,04%)	287,28
Costes indirectos (16,5 €/h)	4950
Total	9450

2.2 Material inventariable.

La expresión que hemos utilizado para el cálculo de esta parte del presupuesto también viene dada por el documento del CTT y es la siguiente:

$$Coste\,material\,inventariable = \frac{Número\,de\,meses\,de\,uso}{12\,meses/año * Periodo\,amortización} * Coste\,equipo * Porcentaje\,uso\,equipo$$

El periodo de amortización se ha tomado como referencia el mismo documento del CTT, que define para equipos de didácticos y de investigación un periodo de 10 años, mientras que para equipo informático y software nos da como referencia un periodo de 6 años.

El porcentaje de uso se ha tomado como horas de tarea en las que dicho equipo o programa informático ha sido utilizado entre el número de horas totales del proyecto.

Considerando esto, el presupuesto del material inventariable es el siguiente:

Equipo	Tiempo de uso (meses)	Años amortización	Coste (€)	Porcentaje uso (%)	TOTAL (€)
Ordenador Portátil Sony Vaio	4	6	769	100	42,72
Ubuntu 14.04	4	6	0	51,67	0,00
ROS Indigo	4	6	0	51,67	0,00
Simulador Rviz	4	6	0	8,33	0,00
Simulador Gazebo	4	6	0	8,33	0,00
Rbcar	4	10	85750	10,00	285,83
Sensor Sick LMS 291-S05	4	10	6500	10,00	21,67
Punto acceso Wi-fi	4	6	25	100	1,39
Software Matlab Education	4	6	500	13,33	3,70
TOTAL					355,31

2.3. Material fungible

Aquí consideramos todo material de corta vida útil que se ha utilizado para el proyecto.

Concepto	Precio (€)
Impresión	2,5
Encuadernación	5
Folios (500 uds)	4,64
Total	12,14

3. Resumen del presupuesto

Concepto	Total (€)
Mano de obra (300h)	9450
Material Inventariable	355,31
Material Fungible	12,14
Subtotal	9817,45
IVA (21%)	2061,67
TOTAL	11879,12