UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

Tesis Final de Master

Departamento de Ingeniería Electrónica

Memoria

Navegación automática en un vehículo con distribución Ackermann

Autor: D. Francisco de Borja Ponz Camps

Directores: Dr. D. Ángel Valera Fernández

Dr. D. Salvador Coll Arnau

Quiero dar las gracias a las personas que me han ayudado en el desarrollo del siguiente proyecto, a Salvador Coll y Ángel Valera por la confianza depositada en mí, y de manera especial a Marc Bosch por su amistad, dedicación, compresión y contribución durante el desarrollo de proyecto.

Dedicado a mis padres, hermana y cuñado, por el apoyo incondicional recibido, aunque han sido unos años donde los problemas se han sucedido sin parecer tener fin, hay que agradecerles, para mi ellos representan más del 90% del merito de este proyecto. Además también por sus ánimos y por hacerme creer que soy capaz de hacer un proyecto como el siguiente, Simplemente Gracias.

Índice

Tabla Imágenes.

Tabla de tablas y ecuaciones.

1) Introducción

1.1) Planteamiento del problema.

En la actualidad, una de las áreas donde se da un crecimiento muy grande de la implantación de sistemas electrónicos y automáticos es la industria de la automoción. Esta implantación está permitiendo que los vehículos no sólo sean más seguros sino que permiten que sean más eficientes y limpios.

Esta tendencia quedó clara desde hace bastante tiempo. De hecho en la conferencia plenaria que W.E Powers, ingeniero de Ford Motors Company, impartió en el 1999 World Congress of IFAC, demostró que los automóviles de la década de los años 90 eran al menos 10 veces más limpios y consumían la mitad de combustible que los vehículos de los años 70, y que estas ventajas se debían en gran parte a los sistemas electrónicos de control basados en microprocesador que se están incorporado, añadiendo seguridad, confortabilidad y facilidad de conducción.

De esta forma podemos encontrar vehículos como el Volvo S80, que cuenta con más de 50 computadores y varias redes de datos, los automóviles de BMW, con más de 85 computadores (45 directamente para control), o algunos modelos de Mercedes Benz en los que se pueden encontrar ente 40 y 75 computadores, y entre 30 y 150 motores eléctricos. Esos sistemas electrónicos utilizan unas 2.500 señales que se agrupan en 4 buses de sistemas, lo que suponen entre 2 y 4 km de cables con un peso que puede llegar hasta los 80 kg.

Así podemos encontrar sistemas electrónicos en cualquier parte del automóvil, como por ejemplo:

- Motores de combustión: acelerador eléctrico, inyección automática de fuel, tren de válvulas mecatrónicos, turbocompresor de geometría variable (VGT), control emisión, bombas y ventiladores eléctricos...
- Sistemas de actuación: transmisiones hidrodinámicas automáticas, cajas de cambio automático, transmisiones variables continuas (CVT), control de tracción automático (ATC), control automático de velocidad y distancia (ACC)...
- Suspensiones: amortiguadores semiactivos, suspensión hidráulica activa (ABC), suspensión neumática activa…
- Freno: frenos anti-bloqueo (ABS), programa electrónico estabilidad (ESP), frenos electro-hidráulicos (EHB), frenos electro-mecánicos (EMB), frenos eléctricos de parking...
- Direcciones: dirección asistida parametrizable, dirección asistida electromecánica (EPS), dirección delantera asistida (AFS)…

El Trabajo Fin de Máster que aquí se presenta se enmarca en esta línea de trabajo: el control y la navegación automática de vehículos. Se trata de un trabajo que está completamente justificado puesto que se puede observar una gran cantidad de proyectos de investigación de las principales marcas del sector de la automoción donde se aborda el control automático de diferentes parámetros y funciones de los vehículos.

1.2) Objetivos.

El objetivo principal que se persigue con este TFM es la realización del estudio sobre las necesidades técnicas para la navegación autónoma de un vehículo ligero con distribución Ackermann, así como su desarrollo final sobre un vehículo real.

El vehículo real es un vehículo ligero disponible en el Instituto Universitario de Automática e Informática Industrial (ai2) de la Universitat Politècnica de València.

Debido a las limitaciones que tiene este tipo de vehículos, uno de los grandes retos en este TFM será poder llevar a cabo la planificación de las rutas a seguir, y conseguir posteriormente que el vehículo siga de forma automática la trayectoria especificada. Para ello será necesario primero realizar una búsqueda bibliográfica que permita analizar el estado del arte actual, y tratar de escoger una solución que se adapte lo mejor posible a las características del sistema disponible en el ai2.

Para tratar de alcanzar este objetivo principal, se proponen los siguientes objetivos específicos:

- Estudiar los diferentes modelos cinemáticos que se pueden encontrar en los vehículos ligeros y robots móviles.
- Analizar las ventajas e inconvenientes y limitaciones del modelo cinemático Ackermann.
- Seleccionar un entorno de desarrollo y el lenguaje de programación para la realización del trabajo.
- Estudiar y adaptar los sensores disponibles en el vehículo eléctrico que puedan ser utilizados en el entorno de programación seleccionado.
- Realizar la fusión de varios sensores que permita aumentar la precisión de las mediciones obtenidas.
- Estudio y desarrollo de un planificar de rutas que permitan al vehículo realizar la navegación autónoma.
- Cálculo, en un tiempo razonable, de la ruta a seguir.
- Analizar y comprobar el correcto funcionamiento del sistema.

1.3) Antecedentes.

El trabajo desarrollado en esta tesina se enmarca en la línea de investigación de control de robots móviles y vehículos ligeros del Grupo de Robótica del Instituto Universitario de Automática e Informática Industrial (ai2) de la Universitat Politècnica de València. El Grupo de Robótica, a partir de diversos proyectos de investigación, dispone de una variedad amplia de plataformas experimentales.

Una de estas plataformas es el vehículo eléctrico que se muestra en la Fig. 1.

Figura 1. Vehículo eléctrico utilizado en el trabajo

Las características físicas del vehículo son las siguientes:

- Dimensiones:
  Largo: 2660 mm.
  Ancho: 1230 mm.
  Alto: 1720 mm (incluyendo barra antivuelco).

- Dimensiones remolque:
  Largo: 790 mm.
  Ancho: 1100 mm.

- Peso: 690kg.

- Velocidad máxima: 32 km/h.

- Motor: 3,3kW AC 48V

- Autonomía: 70 km.

- Angulo máximo de pendiente: 25%

El vehículo está equipado con un PC. El sistema operativo es la distribución Ubuntu 14.04 de Linux sobre la cual se ha instalado la versión *Indigo* del middleware de control de robots ROS (Robot Operating System).

Antes de empezar con este trabajo fin de máster se habían programada algunas funciones básicas para el control del vehículo, como el control de la orientación de las ruedas de dirección, el control de la velocidad y del freno. A partir de estos controles básicos se había desarrollado una aplicación para la teleoperación del vehículo.

Sin embargo, no se disponía de un planificador de trayectorias, ni se podía establecer el control de movimiento del vehículo. Éste fue el motivo que justificó la propuesta de trabajo y lo que finalmente se ha desarrollado.

2) Desarrollo teórico

2.1) Navegación autónoma

2.1.1) Introducción a la navegación autónoma.

En los últimos tiempos se ha podido evidenciar cada vez más la aparición de una serie de vehículos que incorporan diferentes controles automáticos, despertando un gran interés no sólo en las grandes empresas del automóvil, sino también en la industria auxiliar.

Una de las tareas más complejas con la se puede encontrar los vehículos automáticos es la navegación autónoma ya que, aunque puede parecer que es una tarea trivial y/o para los humanos, en realidad se deben considerar un variedad amplia de subtareas y cuestiones que dificultan enormemente obtener una solución adecuada en tiempo real.

De esta forma, a la hora de realizar la conducción, se deben considerar y resolver las acciones siguientes:

- Detectar el entorno: haciendo uso de la visión, las personas debemos analizar qué está ocurriendo en nuestro entorno y las condiciones que nos impone él mismo. Así obtenemos información de las señales y semáforos de tráfico, cómo estamos situado respecto a los carriles de la calle o carretera, obtenemos información que tráfico que nos rodea analizando el comportamiento de los vehículos que circulan junto a nosotros, condiciones de la calzada que pueda provocar un cambio de adherencia de la misma, etc.

- Ajustar los parámetros de la conducción del vehículo en función de las condiciones que se obtengan del entorno, como puede ser aumentar o disminuir la velocidad, cambiar el ángulo de giro del volante, o simplemente modificar el modo de la conducción, de forma que se puede pasar de una conducción más relajada y confortable a una conducción más deportiva y agresiva.

- Selección de la ruta a seguir. Para generar la ruta deseada podemos indicar, desde el punto de partida y el punto de destino, un conjunto de puntos de paso. De esta forma se puede generar la ruta a partir de todo este conjunto de referencias.

La complejidad de cada una de estas tres áreas ha hecho que las soluciones adoptadas por la industria para la resolución de estos problemas sean muy variadas.

Para analizar cómo ha ido evolucionando la navegación autónoma se debe tener en cuenta la evolución de los vehículos autónomos. De esta forma, el primer proyecto se desarrolló en Holanda en 1990. Se trató del proyecto de investigación llamado "*the PATH program*" y planteaba poder conducir los coches en modo de un pelotón o convoy de vehículos. Los sensores utilizados para este propósito fueron principalmente radares, utilizándose además sistemas de comunicación entre los diferentes vehículos y el control central.

Figura 2. Coches autónomos del "*Path program*" en Holanda en 1990.

El siguiente hito destacado en la navegación autónoma fue la puesta en marcha del DARPA (*Defense Advance Research Projects Agency*), organizado por el Departamento de Defensa de los Estados Unidos de América. Esta agencia fue la encargada de organizar por primera vez una competición de vehículos autónomos, con el objetivo de poder comparar diferentes soluciones de vehículos autónomos y tratar de encontrar cuales eran sus puntos débiles y fuertes.

La primera competición realizada fue en el desierto de Mojave en 2004, llamándose *DARPA grand challange*, y surgió de la necesidad de poder mover los vehículos por zona hostiles sin necesidad de tener un conductor al mando. En esta primera edición de la competición se inscribieron más de 100 equipos, aunque ninguno de ellos consiguió terminar.

En la segunda edición celebrada en el año siguiente se llegó hasta los 200 equipos inscritos. En esta ocasión fueron varios los vehículos que sí que lograron terminar el circuito, siendo el equipo de Stanford el ganador de la prueba.

La siguiente edición, denominada 2007 DARPA *urban challange*, introdujo un circuito urbano, de forma que los participantes deberán obedecer las normas de tráfico y comportarse responsablemente en presencia de tráfico.

Algunos de los vehículos que consiguieron completar el circuito propuesto en el tiempo determinado fueron los siguientes:

- **Tartan Racing:** este vehículo (de nombre *Boss*), fue desarrollado sobre un coche de producción por investigadores de la *Carnegie Mellon University*. Se trata de un vehículo que es capaz de seguir las normas de la carretera, de detectar y seguir la trayectoria de otros vehículos a largas distancias, de respetar y resolver las reglas de las intersecciones de calles y carreteras (una de las maniobras más complejas para estos vehículos autónomos), etc.

  El vehículo está equipado con docena de sensores láseres de rango, cámaras y radares para ver el entorno, de forma que es capaz de percibir, planificar y comportarse en situaciones de tráfico.

La gran cantidad de sensores que necesita provoca que no sea solución comercial, no sólo por el precio que suponen todos estos sensores, sino también por el mantenimiento asociado al vehículo.

Figura 3. Foto del Vehículo boss.

- **Stanford Racing:** El segundo vehículo clasificado, conocido como "Junior", fue desarrollado íntegramente por la universidad de Stanford sobre un modelo de Volkswagen, y se puede considerar que es vehículo más cercano a lo que podría ser un prototipo de un vehículo comercial. La principal característica de este vehículo que es capaz de detectar los objetos que se mueven a su alrededor en un rango de 360º gracias a la combinación de los sensores laser y las 6 cámaras. El vehículo cuenta además de una IMU y un GPS.

Figura 4. Foto del vehículo Junior.

- **Victor Tango:** Este vehículo fue desarrollado por la universidad Virginia Tech, y está equipado con 6 sensores laser y dos cámaras de video.

Figura 5. Foto del vehículo Odín

2.2) Configuración cinemática de vehículos.

Una de las cuestiones más importantes que aparece cuando se aborda el modelado y control de robots es el problema o modelo cinemático. Éste calcula la posición, velocidad y aceleración de cada uno de los elementos del robot sin tener en cuentas las fuerzas que causas dicho movimiento.

En el caso de los robots móviles y los vehículos se necesita saber cuántas ruedas motrices y directrices tenemos en sistema, porque esto determinará el tipo y las limitaciones de movimiento que se podrá realizar con el vehículo. De esta forma, para esta clase de robots podemos encontrar dos configuraciones cinemáticas distintas: la configuración diferencial y la configuración de Ackerman.

En la configuración diferencial (Fig. 5) se tienen dos ruedas motrices, de forma que la posición y orientación del robot móvil se obtiene a partir de la diferencia entre la velocidad de la rueda derecha (v_R) y la velocidad de la rueda izquierda (v_L).

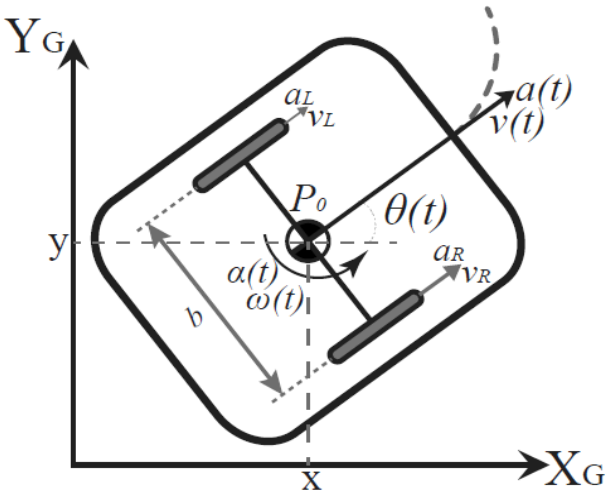

Figura 6. Vehículo con configuración cinemática diferencial

En el modelo cinemático de Ackerman se tienen las ruedas motrices y las ruedas de dirección (Fig. 6).

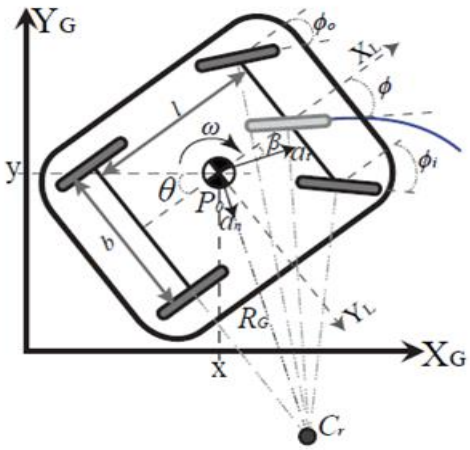

Figura 7. Vehículo con configuración cinemática Ackerman

El vehículo que se ha utilizado en este proyecto tiene esta segunda configuración. Las ruedas traseras son las motrices, mientras que la dirección del vehículo se controla mediante la orientación de las ruedas delanteras. Esta configuración limita el rango de giros que puede realizar el vehículo (Fig. 7), tanto en los movimientos hacia delante, como los que se realizan hacia detrás.

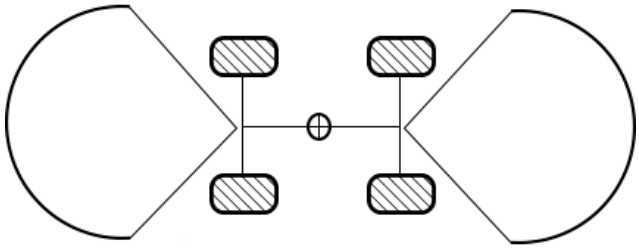

Figura 8. Movimientos vehículo.

Como podemos observar en la figura anterior, se tienen dos zonas con las posibles posiciones a las que se puede mover un vehículo Ackerman. Estas zonas vienen marcadas sobre todo por el máximo ángulo de giro de las ruedas directrices, y suponen una limitación de los movimientos permitidos para el planificador de las rutas que podrá seguir el vehículo.

2.2.1) Modelo cinemático de Ackermann.

El modelo cinemático de los vehículos que circulan por nuestras carreteras está basado en el efecto llamado Ackerman. El hecho que se cumpla este efecto evita un desgaste no homogéneo de las ruedas y, sobre todo, poder evitar deslizamientos que suponen un riesgo muy alto de accidentes.

El efecto Ackerman indica que es muy importante que todas las ruedas del vehículo estén girando respecto a un mismo punto, definido como centro instantáneo de rotación. De esta forma se denomina centro instantáneo de rotación (CIR) al punto donde la velocidad será cero respeto al solido que está girando en ese instante.

Dado que las ruedas traseras son fijas y no cambian de orientación, el CIR estará sobre una línea perpendicular al movimiento del vehículo. Además, en caso de que el vehículo circule en línea recta, el CIR aparecerá en el infinito, por lo que todas las ruedas del vehículo estarán girando a la misma velocidad.

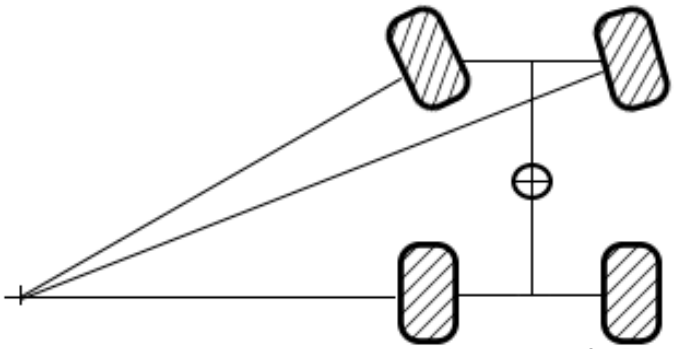

Figura 9. Vehículo con un sólo CIR.

La Fig. 8 muestra un vehículo con un solo CIR. Se puede observar cómo la rueda más cercana al interior de la curva tendrá un ángulo de giro mayor.

En los vehículos en los que no se tienen en cuenta el principio de Ackermann se puede comprobar que aparecen dos CIR en la parte trasera del vehículo (Fig. 10, lo que provoca que el vehículo no están girando respecto al mismo punto, induciendo a que aparezcan inestabilidades cada vez que se toma una curva.

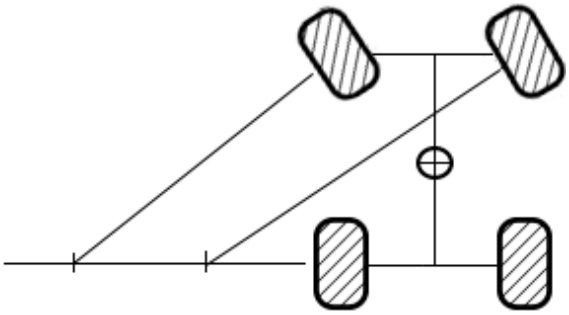

Figura 10. Vehículo con dos CIR.

2.3) Sensores en los vehículos autónomos.

Cuando se trata de abordar la navegación autónoma de vehículos, una de las cuestiones más importantes que aparecen es la detección correcta del entorno, por lo que será necesario disponer del conjunto adecuado de sensores.

Para obtener el posicionamiento y movimiento del vehículo se suelen utilizar los sistema de posicionado global (GPS) basados en satélites o las unidades inerciales (IMU). Además, también se pueden encontrar sensores de distancia basados en láser y/o cámaras para obtener una visión estéreo.

La Fig. 11 muestra es esquema de los sensores del vehículo de la Carnegie Mellon University. La información detallada de cada uno es estos sensores viene resumida en la Tabla 1.

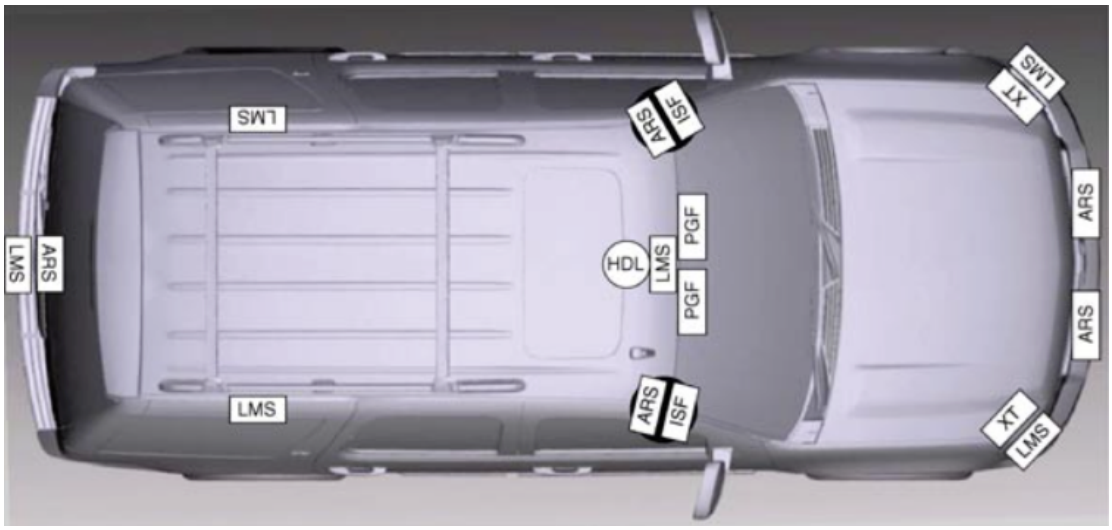

Figura 11. Distribución de sensores en el vehículo "boss".

Sensor	Características
Applanix POS-LV 220/420 GPS/IMU (APLX)	Fuertemente inercial acoplada.
Sick LMS 291-S05/S14 LIDAR(LMS)	180º Verticales con resolución angular de 0.5º. 80 metros de rango máximo.
Velodyne HDL-64 LIDAR (HDL)	360º horizontales x 26º Verticales con resolución de 0.1 grados. 70 metros de rango máximo.
Continental ISF 172 LIDAR (ISF)	12º horizontales x 3.2º. Verticales 150 metros de rango máximo.
Continental ARS 300 Radar (ARS)	60º/17º horizontales x 3.2º Verticales
Point Grey Firefly (PGF)	Cámara con gran rango dinámico 45º FOV
IBEO Alaska XT LIDAR (XT)	240º Horizontales x 3.2º Verticales 300 metros de rango máximo.

Tabla 1. Datos sensores coche "boss".

Como se puede observar, aunque la gran mayoría de los sensores están colocados en la parte la parte delantera del vehículo, la configuración no sólo permite capturar prácticamente todos los ángulos del vehículo, sino que además hay ciertas zonas con una superposición de la información captada por los sensores.

Así, por ejemplo, la combinación de la información obtenida con los radares con la obtenida a partir del conjunto de las cámaras en estéreo hace que posible tratar las imágenes para detectar vehículos y/o transeúntes. Además, para poder obtener una distancia más certera de puede disponer de una calibración entre los diferentes sensores laser y las cámaras.

Un segundo ejemplo de sistema autónomo es el vehículo Odín desarrollado por Victor Tango en la universidad Virginia Tech. En este caso, a pesar de tener un presupuesto más ajustado (cuenta con un total de 9 sensores), la distribución de los sensores es capaz de captar los 360º grados del vehículo, incluyendo la superposición de éstos en algunas zonas.

Sick LMS 291-S05/S14 LIDAR(Amarillo)	180º Verticales con resolución angular de 0.5º. 80 metros de rango máximo.
Point Grey Firefly (Rojo)	Cámara con gran rango dinámico 45º FOV
IBEO Alasca XT LIDAR (Azul)	240º Horizontales x 3.2º Verticales 300 metros de rango máximo.
IBEO Alasca A0 LIDAR (Morado)	160º Horizontales x 2.0º Verticales 80 metros de rango máximo.

Tabla 2: Datos sensores vehículo Odín.

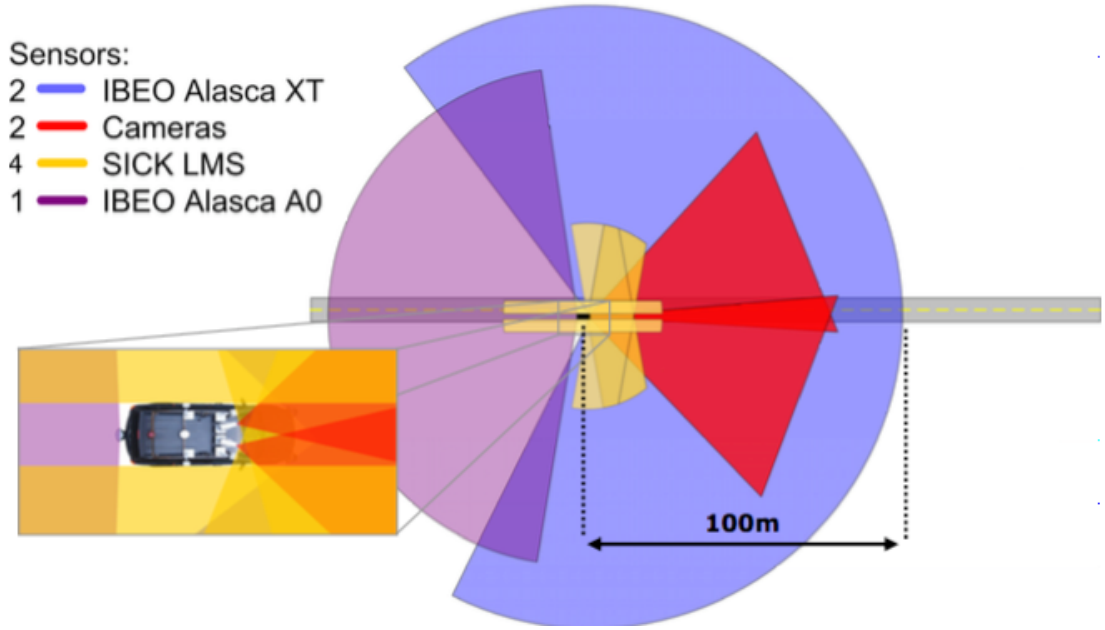

Figura 12. Rango de detección del vehículo Odín.

2.4) Componentes de la navegación autónoma.

Analizando los diferentes vehículos participantes en la *DARPA Urban Challenge* se ha podido comprobar que cada grupo de investigación ha desarrollado una solución particular a la hora de abordar la navegación autónoma de vehículos. Por ello es interesante analizar las distintas soluciones de navegación propuestas para tratar de determinar si existen algunos factores comunes.

Por ejemplo, en el caso del vehículo Junior desarrollado por Stanford, el diagrama funcional es el siguiente:

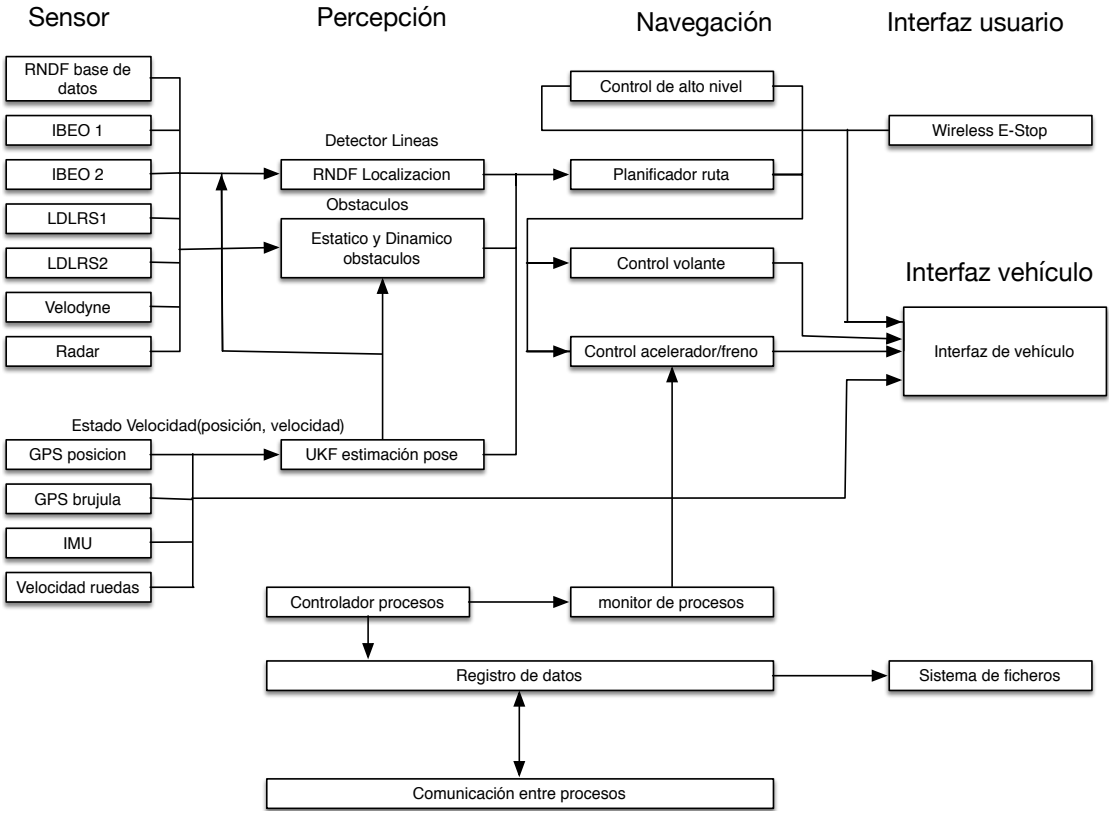

Figura 13. Arquitectura funcional del vehículo Junior.

En esta solución, el primero de los bloques que nos encontramos es la parte de interfaz, donde se engloban todos los sensores y se adecuan las salidas.

A continuación está el bloque de percepción *(Perception)*, encargado de detectar las líneas de la carretera y de corregir el mapa (RNDF Localization). La etapa *Estatico y Dinamico obstaculo*, encargada de determinar los obstáculos, tanto estáticos como dinámicos que se encuentran en el entorno. Para esta detección se utilizan los sensores láser y las diferentes cámaras que dispone el vehículo. Por último se tiene uno de los bloques más importantes: UKF Pose Estimation. Se trata de un filtro de Kalman unscented que a partir de los datos del GPS, la IMU y los encoders de las ruedas realiza una estimación de la posición del vehículo respecto del eje de coordenadas.

Por último se encuentra el bloque de navegación, que se encuentra dividido por 4 etapas: *Top Level Control, Path Planner, Steering Control y Throttle/Brake Control.* La etapa más importante es la planificación de caminos, encargada de generar la

trayectoria. Esta etapa está dividida a su vez en dos subetapas: un planificador local encargado de generar los posibles movimientos para seguir la trayectoria global, y la parte encargada de generar las acciones de control para el sistema de la dirección (*steering control*) y el acelerador y freno (*throttle/brake control*).

Una segunda arquitectura de control analizada es la desarrollada por el MIT en el vehículo *Talos*. Esta configuración tuvo como premisa gastar un gran número de sensores de bajo coste, en vez de realizar el proyecto utilizando sensores caros.

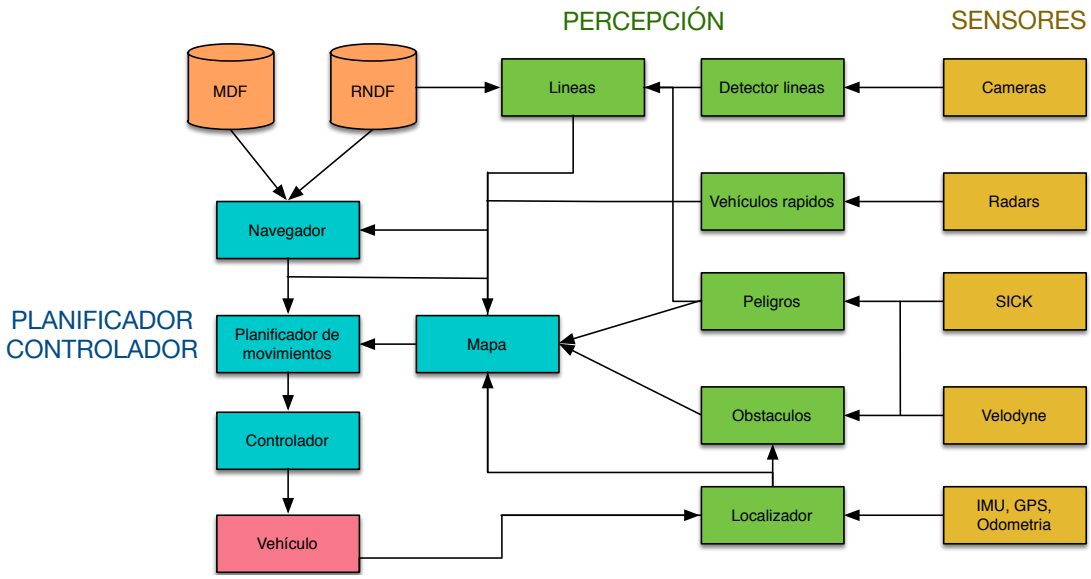

Figura 14. Arquitectura del vehículo Talos.

Los bloques que componen esta arquitectura son los siguientes:

- *Road Paint Detector:* en este bloque se utiliza los datos recibidos por las cámaras y mediante el uso de Splines, realiza un ajuste de las líneas de la carretera
- *Lanes:* encargado de localizar el vehículo en la carretera. Para ello utiliza los datos disponibles en la base de datos y se comparan con las detectadas por el bloque anterior.
- *Obstacles:* tiene la misma funcionalidad que la de Junior, encargándose de la detección de los obstáculos dinámicos y estáticos.
- *Hazards:* se encarga de la detección de los bordillos y salientes que el vehículo puede encontrar en la navegación autónoma. Este nodo utiliza los un conjunto sensores de distancia láser inclinados hacia debajo.
- *Positioning:* este bloque calcula la posición en dos coordenadas: una local y una global basada en el GPS. Para calcular el posicionamiento se fusionan los datos obtenidos por la IMU, el GPS y la odometría.
- *Navigator:* basándose en los datos que hay en RNDF y en MDF, este bloque se encarga de seguir el estado de la misión y desarrolla un plan a alto nivel. La salida de este bloque proporciona al bloque *Motion Planner* un objetivo a corto plazo.
- *Drivability Map:* provee una interfaz para manejar eficientemente los datos tratados en la etapa de percepción.
- *Motion Planner:* identifica y optimiza las posibles trayectorias que se necesitan para moverse en dirección al objetivo destino.

- *Controller:* se encarga de controlar que las acciones que se realizan son las necesarias para conseguir los objetivos en la ruta.

Por último, se va a presentar la arquitectura propuesta para el desarrollo del vehículo Odín. Tal como se puede verificar en la figura siguiente, el contenido de los bloques de la arquitectura propuesta es idéntico a las propuestas anteriores.

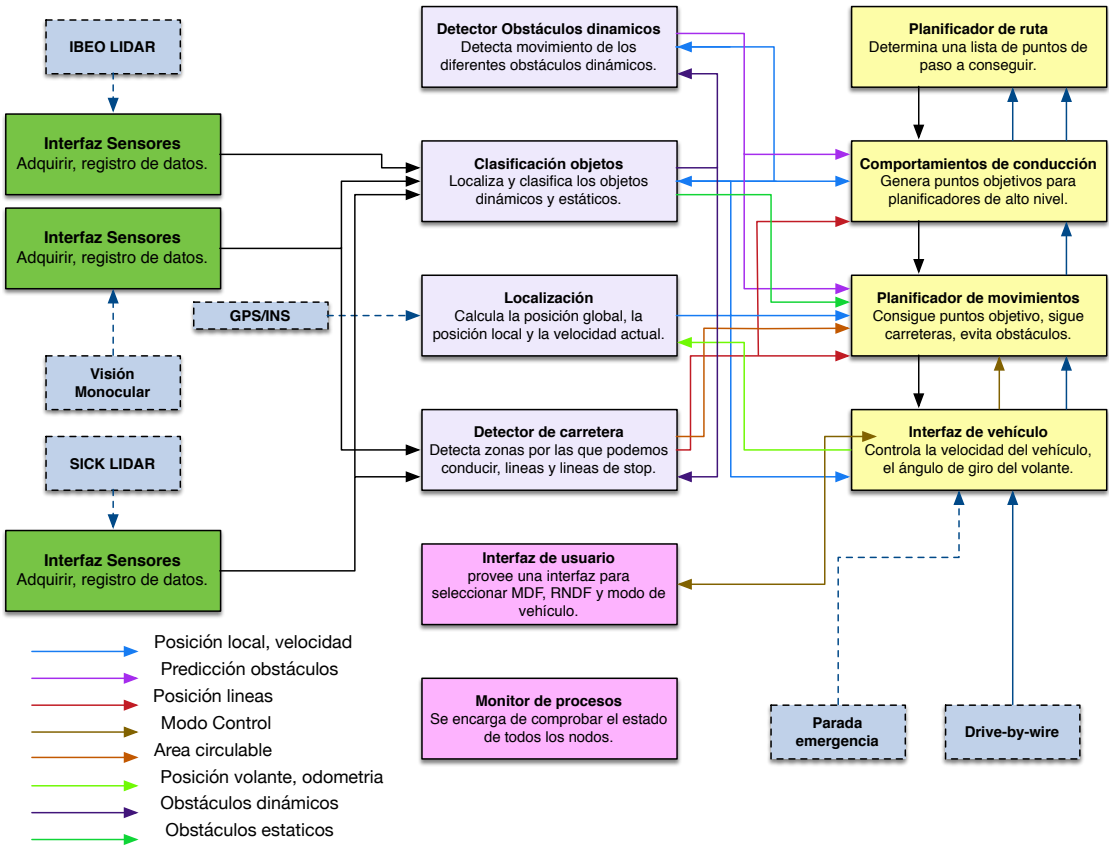

Figura 15. Arquitectura del vehículo Odín.

Por lo tanto, después de analizar las propuestas de las tres arquitecturas, se puede determinar que el vehículo autónomo que se plantea en este trabajo debería disponer de un localizador, un paquete de navegación y un paquete para controlar obstáculos, tanto dinámicos como estáticos.

Estos contenidos se desarrollarán con mayor detalle en los apartados siguientes.

2.4.1)Algoritmos de navegación.

Si bien en la actualidad se pueden encontrar una gran variedad de algoritmos de optimización de ruta, la mayoría de éstos algoritmos están basados en dos algoritmos: el algoritmo de Dijkstra y el algoritmo conocido como A*(A-estrella).

Ambos algoritmos están basados en la utilización de grafos, de forma que la ruta se calcula mediante la utilización de nodos y aristas. Los nodos, que tienen una puntuación, están interconectados mediante el conjunto de aristas, las cuales también poseen una puntuación.

El **algoritmo Dijkstra** fue formulado por Edsger Dijkstra en 1959, siendo uno de los más conocidos y utilizados a día de hoy. La idea principal de este algoritmo es, mediante una búsqueda uniforme, ir explorando todos los caminos desde el nodo origen hasta el nodo destino de manera que se escogerá aquel que tenga una menor puntuación, es decir, que sea el más corto.

```
DIJKSTRA (Grafo G, nodo_fuente s)
        para u ∈ V[G] hacer
          distancia[u] = INFINITO
          padre[u] = NULL
        distancia[s] = 0
        añadir (cola, (s,distancia[s]))
        mientras que cola no esta vacía hacer
          u = extraer_mínimo(cola)
          para todos v ∈ adyacencia[u] hacer
            si distancia[v] > distancia[u] + peso (u, v) hacer
              distancia[v] = distancia[u] + peso (u, v)
              padre[v] = u
              añadir(cola,(v,distancia[v]))
```

Algoritmo 1. Algoritmo de Dijkstra.

Es necesario tener en cuenta que uno de los inconvenientes de este algoritmo es que no acepta valores con coste negativo en las aristas. En caso de necesitar esta funcionalidad se debería, por ejemplo, utilizar la modificación llamada Bellman-Ford.

Como se ha comentado, el segundo algoritmo utilizado habitualmente para la generación de caminos es el **algoritmo A***. Este algoritmo fue formulado por Peter E. Hart, Nils J Nilsson y Bertram Raphael en 1968. Se trata de un algoritmo de búsqueda que, a diferencia de Dijkstra, asegura encontrar el camino con menor coste desde el origen hasta el destino. Para asegurar que encuentra la mejor solución el algoritmo comprueba todas las posibles soluciones, requiriendo para ello un mayor coste computacional.

```
Algoritmo A*(Grafo g, Sinicio, Sdestino)
        g(Sinicio)=0
        Abierto = 0
        Insertar Sinicio en Open con valor f(Sinicio)= E*h(Sinicio)
        Mientras Sdestino no esta expandido
                Quitar s con el menor valor desde Abierto
                Si s' no ha sido visto antes hacer
                        F(s') = G(s')=Infinito
                Si G(s') es mayor que G(s) + C(s, s') hacer
                        G(s') = G(s)+C(s,s')
                        F(s')=G(s')+E+H(s')
                        Insertar s' en Abierto con F(s')
```

A la hora de programar los algoritmos de búsqueda se suelen considerar unos ciertos límites en los planificadores que nos asegure una salida en tiempo real. Por ello se ha desarrollado una modificación del algoritmo A*, el algoritmo ARA*. Es un algoritmo basado en primitivas en el que la búsqueda se realiza desde el destino hasta el punto de inicio que permite recalcular la ruta para mejorar la parte que queda por recorrer

```
Key(s)
        Devolver g(s)+E·h(Sinicio, S)


ImprovePath()
        Mientras(min_S·∈ Abierto(key(s))menor que key(Sinicio))
                Quitar s con la minima key desde Abierto
                CERRADO = CERRADO ∪ {s}
                Para todos s' ∈ Pred(s)
                        Si s' no ha sido visitado antes hacer
                                G(s') = infinito
                        Si G(s') menor que C(s', s) +G(s) hacer
                                G(s') = C(s', s) + G(s)
                        Si G(s') no esta Cerrado. hacer
                                Añadir s' en Abierto con key(s')
                        Si no hacer:
                                Añadir s' en INCONS
```

```
Main()
        G(Sinicio) = Infinito
        G(Sdestino) = 0
        Abierto = Cerrado = INCONS = Nulo
        Insertar Sdestino en Abierto con key(Sdestino)
        ImprovePath()
        Publicar la actual la solución sub-optima
        Mientras A > 1
                Decrementar |A|
                Mover Estados desde INCONS hasta Abierto
                Actualizar prioridades para todos las S ∈ Abierto de acuerdo con Key(s)
                Cerrado = Nulo
                ImprovePath()
                Publicar la actual solución sub-optima.
```

Con este algoritmo se evita calcular la ruta y seguirla hasta el final, sino que se obtiene una ruta y después se va recalculando en función de los diferentes cambios.

A la hora de cálculo de rutas, hay un algoritmo que no se calcula mediante la colocación de nodos y aristas sobre un espacio imaginario: el a**lgoritmo banda elástica.** Este algoritmo se basa en principio de modelos físicos. Su funcionamiento se puede simplificar como una banda que va desde el Inicio hasta el destino modelado como un conjunto de muelles. En cada extremo se considera que se aplica una fuerza.

$$\overrightarrow{ftotal} = \overrightarrow{finterior} + \overrightarrow{fexterior} + \overrightarrow{fconst} \qquad (1)$$

En función de los obstáculos que van apareciendo las fuerzas van cambiando y se obtiene una trayectoria nueva

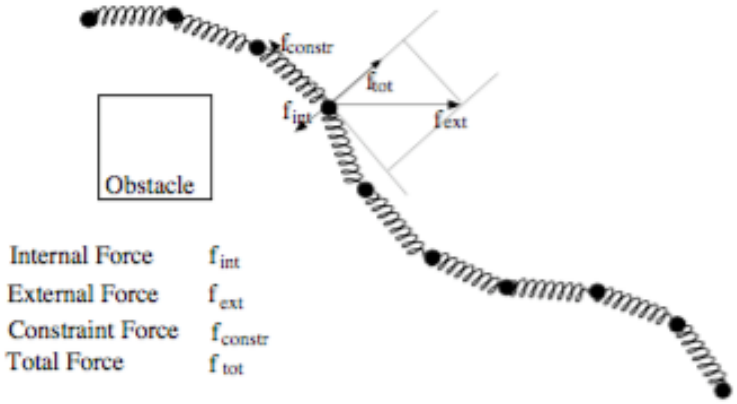

Figura 16. Modelo de banda elástica.

2.5) Arquitectura orientada a servicios.

2.5.1) Introducción a las arquitecturas SOA.

En los últimos años, la tendencia predominante de las arquitecturas software para el desarrollo de aplicaciones se basan en lo que se conoce como arquitecturas basadas en servicios (arquitecturas SOA, *Service Oriented Arquitecture*). Es una metodología utilizada para estructurar el software de proyectos y aplicaciones en el que el objetivo principal es la utilización de un conjunto de servicios para dar soporte a los requisitos del producto final de forma que es una tecnología neutral e independiente del lenguaje gastado para su desarrollo.

Esta arquitectura software tiene sus orígenes en los años 80, cuando se empezó a desarrollar y a aplicar la programación orientada a objetos. Después en los años 90 los modelos basados en componentes fue la antesala a los actuales modelos orientados a servicios. En 1996 la consultora Gartner presento por primera vez las arquitecturas orientada a servicios, pero su uso no se expandió hasta la llegada de la revolución de las empresas llamadas *punto.com*.

Dentro de la arquitectura SOA los conceptos más importantes son las siguientes:

- **Servicios:** es una función sin estado y autocontenida, que acepta peticiones y suele devolver una respuesta dentro de una interfaz muy bien definida. Los servicios pueden ser síncronos o asíncronos: los síncronos suelen devolver la respuesta cuando se realiza la llamada, mientras que los asíncronos lo pueden hacer en otro momento.

- **Orquestación:** Es la parte encargada de secuenciar los servicios y en caso necesario proveer la lógica adicional para procesar los datos.

- **Proveedor:** La función que ofrece un servicio como respuesta a una llamada desde un consumidor.

- **Consumidor:** La función que realiza la llamada a un proveedor, y espera hasta conseguir la respuesta.

Frente a otros tipos de arquitecturas, las principales ventajas de la utilización de arquitecturas SOA son: una clara mejora en el tiempo de realización de cambios sobre procesos ya existentes y una gran facilidad para la integración de nuevas tecnologías puesto que permite la generación de aplicaciones reutilizables y adaptables a nuevos desarrollo. Esto supone también una clara mejora en lo referente a los costes de posibles ampliaciones del proyecto.

Existen numerosos desarrollos de arquitecturas SOA orientadas al control y la programación de control de robots. Cada una de ellas tienen diferentes plataformas, posibilidades de tiempo real y simulaciones, etc. La tabla siguiente muestra las arquitecturas SOA más utilizadas.

Nombre	Open Source y Free Software	Arquitectura distribución avanzada	Drivers Hardware	Algoritmos robótica	Simulaciones	Orientado a control y tiempo real
JDE+	Si	No	Si	Si	Si	No
OpenRave	Si	No	No	Si	Si	No
OpenRDK	Si	Si	Si	No	No	No
OpenRTM	Si	Si	Si	Si	No	No
OpROS	No	Si	Si	Si	Si	No
ORCA	Si	Si	Si	No	No	No
OROCOS	**Si**	**Si**	**Si**	**Si**	**No**	**Si**
ROS	**Si**	**Si**	**Si**	**Si**	**Si**	**No**
Webots	No	No	Si	No	Si	No
Yarp	Si	Si	Si	No	Si	No

Tabla 3: Comparativa de las diferentes SOA.

En la tabla anterior se han resaltado dos arquitecturas orientadas a servicio que tienen una gran implementación en el campo de la robótica y que se están quedando como un estándar en el área de la robótica. La primera de ellas es OROCOS, cuya principal característica es la de disponer de un planificador de tareas basado en tiempo real. La segunda plataforma es ROS, que dispone un simulador muy potente y un conjunto de herramientas y funcionalidades de alto nivel de control de robot. Por ello, ésta ha sido la plataforma escogida para el desarrollo del presente trabajo.

2.5.2) Plataforma Robot Operating System (ROS).

Los orígenes de la plataforma ROS datan de 2007, cuando el *Laboratorio de Robótica e Inteligencia Artificial* de la *Universidad de Stanford* empezó un proyecto bajo el nombre de *Switchyard*. Inicialmente este proyecto nació para intentar desarrollar el software de soporte para uno de los robots que estaban desarrollando en aquel tiempo llamado *Robot Stair*. A partir de 2008 el proyecto fue desarrollado por la incubadora de empresas *Willow Garage*, que fue el mantenedor del proyecto hasta 2013, cuando *Willow Garage* cedió los derechos a la *Open Source Robotics Foundation*.

La idea principal de ROS fue la creación de una herramienta base para que laboratorios y empresas del mundo dispusieran de un estándar y un software para utilizarlo. Con la filosofía de disponer de un software libre se podría optar a realizar diseños muy complejos sin la necesidad de disponer de muchos medios o especialistas en todas las áreas de la robótica.

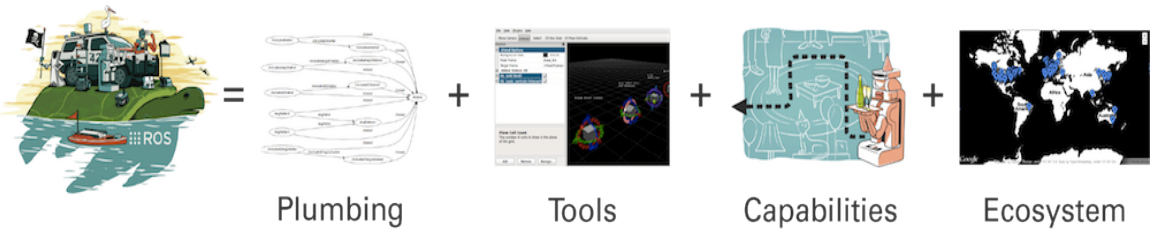

Figura 17. Ilustrativa de la idea general de ROS.

En la actualidad ROS se encuentra distribuido bajo una licencia *Creative Commons Attribution 3.0*, Esta licencia permite compartir, copiar, modificar y redistribuir mediante cualquier medio o formato. Además, se facilita totalmente distribuir las necesidades de computación, permitiendo equilibrar el sistema de una forma simple.

Existen disponibles varias versiones de ROS. Las versiones de desarrollo se publican cada 6 meses y es donde se van implementado las funcionalidades que se aplicarán de las versiones LTS (*Long Term Service*). Estas versiones se publican alrededor de cada año. La última versión disponible es *Indigo Igloo*, y fue liberada a la comunidad en mayo de 2014. Se espera que la nueva versión, conocida como kinetic kame, saldrá a lo largo del año 2016.

Figura 18. Logo de ROS Indigo Igloo.

2.5.2.1) Conceptos básicos de ROS.

Dentro de ROS se pueden encontrar con una serie de conceptos básicos:

Roscore: Es bloque más importante de toda la estructura de ROS puesto que para tener a éste funcionando se debe tener funcionando este nodo. La definición que ofrece la comunidad es la siguiente: es una colección de ***nodos*** y programas que son los pre-requisitos para tener un sistema basado en ROS. De esta forma, es el programa que se va a encargar de controlar la comunicación entre las diferentes partes de la aplicación.

Cuando se lanza el ***roscore***, en realidad se lanzan tres procesos diferentes:

- **Master**: es el encargado de proveer nombres y gestionar servicios, y de servir publicadores y subscriptores para el resto de nodos del sistema ROS. Además, mediante conexiones *peer-to-peer,* es el nodo que gestiona cómo se interconectan los diferentes nodos y sus comunicaciones.
- **Parameter Server**: este programa se utiliza para almacenar datos de los diferentes parámetros y tiene la funcionalidad de responder a demandas de los diferentes estados de los parámetros.
- **Rosout**: Este nodo es el que se encarga de republicar toda la información a los diferentes nodos interesados. Además este nodo se encarga también de generar el *log* que se puede obtener con la ejecución de ROS.

Nodos: Son los recipiente donde se desarrollan los diferentes procesos. Para la comunicación entre diferentes procesos se disponen de diferentes herramientas como *topics, servicies* y *parameters*. El hecho de poder disponer de nodos permite

aplicar técnicas de escalabilidad, lo que permite que la realización e implementación de programas sea más simples.

Topics: todo nodo que desee comunicar periódicamente alguna información con los demás nodos debe utilizar *topics*. Se disponen de dos funcionalidades: los publicadores, que son los que escriben en el topic, y los subscriptores, que son los que reciben la información. Para poder establecerse la comunicación sin ningún problema es necesario que tanto el publicador como el subscriptor utilicen el mismo tipo de mensaje y que el nombre del topic sea también el mismo.

Mensajes: ROS permite disponer de una serie de mensajes estándar, aunque se pueden utilizar de la misma forma mensajes creados por el propio programador. Estas estructuras se utilizan en los subscriptores o publicadores. Por lo general los mensajes son bastante simples y suelen estar compuestos por diferentes campos.

float64 x
float64 y
float64 z

Estructura del mensaje geometry_msgs/Vector3.msg

Services: Los servicios son los encargados de realizar un intercambio de información entre nodos, pero a diferencia de los *topics,* éstos funcionan bajo demanda. Para poder utilizar los servicios se deben tener dos nodos, uno configurado como un *Service Server* y otro como *Service Client*. Como se puede intuir, el nodo que realiza la llamada es el cliente, y el servidor es el encargado de procesar los datos recibidos y generar la respuesta.

En la siguiente imagen se muestra con más detalle las diferencias entre la comunicación basada en topic, en la cual la publicación se hace a una frecuencia fija, y la comunicación basada en servicios, en la cual ésta se realiza bajo demanda.

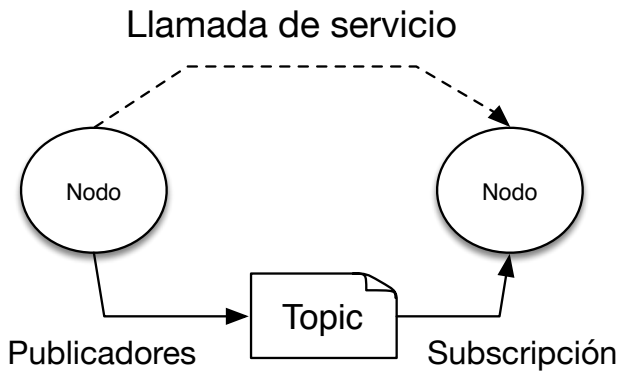

Figura 19. Comunicación vía topic y servicio.

Bags: los bags es un formato con el que se pueden guardar todo el tráfico de datos entre el *roscore* y los nodos de la aplicación. Como se guarda también la información del tiempo, se puede reproducir posteriormente y tantas veces como se quiera los datos y la respuesta del sistema, lo que permite verificar el funcionamiento de la aplicación.

Transformada: proporciona al usuario la posibilidad de mantener un seguimiento de múltiples coordenadas del robot. Las transformadas se introducen en un buffer, y se puede rescatar el estado en el que estaban las transformadas en un instante de tiempo determinado. Las transformadas se pueden realizar por medio de un fichero de configuración o bien estáticamente.

En la siguiente figura se puede observar diferentes transformadas, una por cada eje y por cada sensor que dispone el robot PR2.

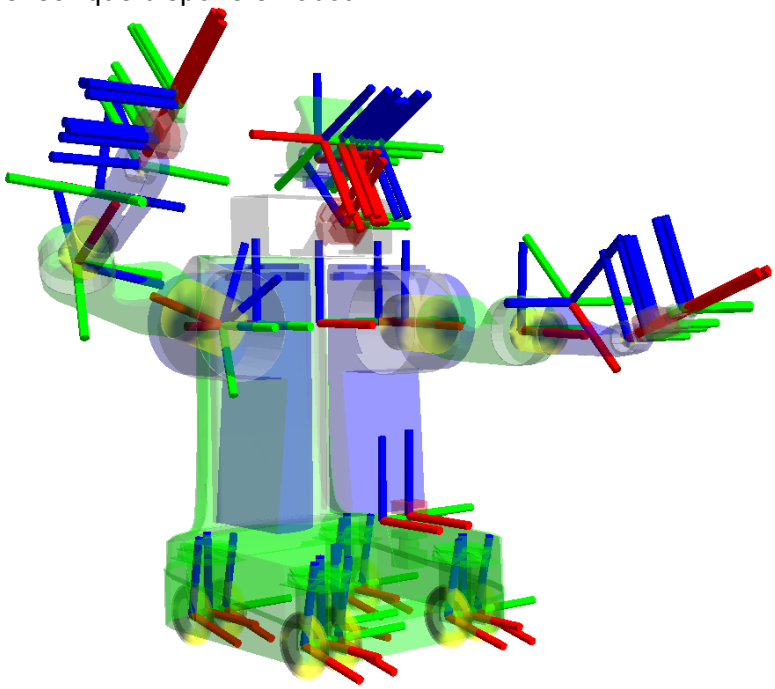

Figura 20. Conjunto de transformadas de un robot.

3) Desarrollo Práctico.

En los siguientes apartados se van tratar a las diferentes metodologías utilizadas para la el desarrollo práctico del trabajo realizado en el presente trabajo fin de máster. De esta forma se van a ir tratando las diferentes etapas realizadas desde el principio del proyecto.

Los pasos que se han seguido durante el desarrollo del vehículo han sido los siguientes: primero se tuvieron que adaptar los sensores del vehículo para cumplir con los estándares definidos por ROS. Una vez conseguido este objetivo se pasó a comprobar las limitaciones cinemáticas del vehículo. Posteriormente se desarrolló del paquete de navegación y por último se desarrolló una aplicación que interactuaba con la navegación del vehículo.

3.1) Adaptación de los sensores del vehículo.

Para poder realizar la navegación autónoma del vehículo se debe poder disponer de varios sensores. En la parte frontal del vehículo se ha ubicado un sensor de rango láser para la detección de las distancias en un plano 2D. También se dispone de un sensor que incluye una unidad inercial y de un GPS, ambos colocados lo más cerca del centro del vehículo pero a diferentes alturas.

Para poder obtener la información del desplazamiento de cada rueda del vehículo, cada una de éstas está equipada con un encoder. Por último, para poder medir el ángulo de giro de las ruedas de dirección, el vehículo viene equipado con otro encoder.

Cualquiera de los sensores que vayan colocados en el vehículo debe cumplir con los *ROS Enhancement Proposals* (REP). Éstos son las especificaciones que se deben utilizar cuando se está desarrollando un proyecto de ROS. Además, también nos indica cómo se deberían configurar los distintos sensores. En concreto el REP103 define las unidades de medidas estándar y las coordenadas. Como se puede comprobar, las 4 primeras son las unidades base del sistema internacional y las siete siguientes las unidades derivadas del sistema internacional:

Cantidad	Unidad
Longitud	Metros
Masa	Kilogramos
Tiempo	Segundos
Corriente	Amperios
Angulo	Radianes
Frecuencia	hercios
Fuerza	Newton
Potencia	Vatios
Voltaje	Voltios
Temperatura	Celsius
Magnetismo	Tesla

Tabla 4. Unidades Sistema Internacional

El primer sensor con el que se empezó a trabajar fue la unidad *IMU* y el GPS. En este caso se trata del dispositivo IG500N, fabricado por la empresa SBG.

Figura 21. Dispositivo IG500-N.

La primera prueba que se realizó con el sensor fue observar cual era la configuración de los ejes de coordenadas. Según indica la REP105 es necesario que la configuración del sensor sea ENU, lo que significa que la coordenada X apunte hacia el Este, la coordenada Y hacia el Norte y la coordenada Z hacia arriba (*Up*). Sin embargo se comprobó que el dispositivo IG500N tenía una configuración NED, de forma que las coordenadas X apuntaban al Norte, las coordenadas Y apuntaban al Este y las coordenadas Z, apuntaban hacia abajo. Por lo tanto se tuvo que utilizar en la configuración de la unidad inercial una matriz de rotación para convertir de NED a ENU.

$$\begin{pmatrix} E \\ N \\ U \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{pmatrix} N \\ E \\ D \end{pmatrix} \quad (1)$$

A continuación se tuvo que comprobar si las rotaciones sobre RPY (Roll, Pitch, Yaw) eran rotaciones sobre los ejes X-Y-Z respectivamente. Para ello se comprobó que al hacer un movimiento de 90º grados sobre el eje X (es decir, una rotación Roll de 90º) se obtenía una aceleración de +9,81 m/s^2. Para el eje Y se realizó lo mismo, comprobándose que al hacer una rotación sobre el Pitch de +90º se obtenía una aceleración de +9,81 m/s^2. Por último se pudo comprobar que en estado de reposo se tenía una aceleración de +9,81m/s^2 en el eje Z.

Una vez verificada la correcta configuración del dispositivo, éste se fijó al coche para que la coordenada X apuntara hacia la parte delantera del vehículo, la coordenada Y apuntara hacia la parte izquierda y la coordenada Z apuntara hacia arriba.

Después de verificar la configuración las salidas de la IMU y de la antena GPS se tuvo que programar mediante una librería C++ proporcionada por el fabricante un nodo de ROS que fuera capaz de integrar el sensor en la plataforma de ROS.

Los parámetros que acepta el nodo de IMU+GPS son los siguientes:

- *imu_frame_id:* indica el nombre de la transformada de la IMU.
- *gps_frame_id:* indica el nombre de la transformada del GPS.
- *dev:* indica la dirección del puerto que está conectado.

La segunda clase de sensores que se tuvieron que configurar fueron los sensores láser. En concreto el vehículo está equipado con dos láseres: el *SICK*

LMS291-s05 y el *Hokuyo urg-04lx*. En el caso de estos sensores, ambas marcas proporcionan un nodo de ROS que se encarga de conectar el dispositivo y publicar a través de un topic de ROS la información recibida por el láser.

El láser SICK LMS291-s05 tiene las especificaciones siguientes:

- Alimentación: 24Voltios DC.
- Potencia: 30 Watts.
- Alcance detección laser: 80 metros.
- Resolución angular: 0.5º
- Peso: 1.4Kilogramos.
- Área de detección: 180 º.

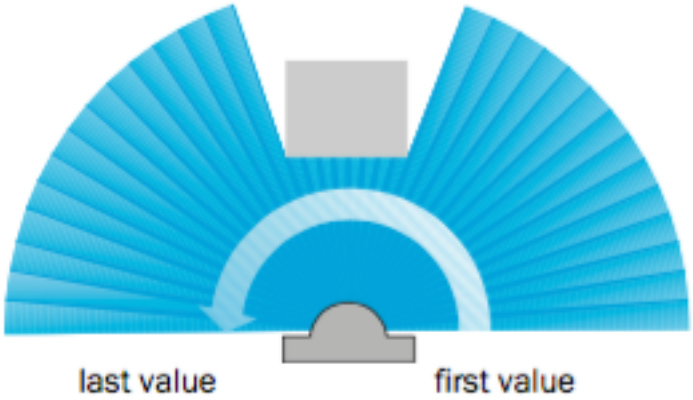

Figura 22. Área detección Sick lms291-s05.

El láser Hokuyo urg-04lx cuenta con estas especificaciones

- Alimentación: 5 Voltios DC.
- Potencia: 2.5 Watts.
- Alcance detección laser: 4 metros.
- Resolución angular: 0.36º
- Peso: 160 gramos.
- Área de detección: 240 º.

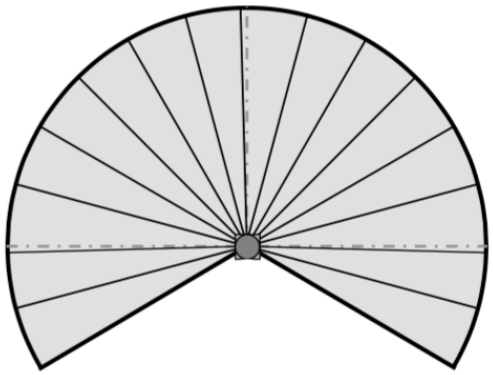

Figura 23. Área de detección Hokuyo 04urg-lx.

Como podemos ver entre ambos láseres hay una gran diferencia, tanto en área de detección como en longitud máxima de detección.

Para poder empezar a utilizar los láseres solo es necesario instalar los nodos de los sensores en el ordenador que van a estar conectados. Para ello se tuvieron que seguir los pasos siguientes:

1. Instalar paquete de **Hokuyo urg-04lx**:
   $ sudo apt-get install ros-indigo-hokuyo-node

2. Configurar los parámetros que acepta el nodo:
   frame_id: donde se debe indicar la transformada del sensor.

3. Comando instalar paquete de **Sick Lms 291-s05**:
   $ sudo apt-get install ros-indigo-sicktoolbox
   $ sudo apt-get install ros-indigo-sicktoolbox-wrapper

4. Configurar los parámetros que acepta el nodo:
   frame_id: se indica la transformada del sensor.
   port: indica el puerto USB (p.e. /dev/ttyUSB0).
   Bauds: indica los baudios que debe tener la conexión.
   connect_delay: indica un pequeño retraso al conectarse.

Para acabar la configuración es necesario especificar las transformadas entre los diversos sensores y ciertos puntos del vehículo. En ROS es aconsejable que todos los sensores se referencien respecto de *base_link*.

En ROS, el fichero que se encarga de publicar las transformadas respecto de algún marco de trabajo es el fichero '*xacro'*. Por ello, para cada sensor que se utilice en el vehículo, hay que añadir estas líneas:

```
<xacro:sensor_imu name="name_id" parent="base_link">
        <origin xyz="-0.6 0.0 0.75" rpy="0 0 0"/>
</xacro:sensor_imu>
```

Es éste se pueden ver varios parámetros. En la segunda línea se indica donde está instalado el sensor y con qué orientación. Por tanto *xyz* indica la diferencia desde el centro del vehículo hasta la posición del sensores utilizando como unidades metros. A continuación está *rpy*, que hace referencia a los ángulos de *Roll Pitch* y *Yaw*. En este caso, al ser una rotación de acuerdo con los REP previamente comentados, están en unidades de radianes.

Además de editar el fichero '***xacro'*** con la configuración de cada sensor es necesario también añadir el modelo ***urdf***. Este fichero contiene un modelo 3D del láser puesto que pueden aparecer problemas en caso de no disponer de los ficheros con los modelos 3d en las simulaciones cuando se utiliza un simulador como Gazebo.

En cuanto a los encoders de cada una de las ruedas y del volante, la empresa que realizó la instrumentación de esta parte del vehículo proporcionó los nodos encargados de la lectura de estos sensores. Sin embargo, fue necesaria realizar una calibración de dichos sensores

3.1.1) Calibración del volante.

Para poder hacer una navegación autónoma, la dirección del vehículo está equipada con un motor. Por ello, para poder hacer la calibración del volante se tuvieron que resolver dos problemas. En el primero se tuvo que obtener el ángulo de giro que realmente tenía el volante. En el segundo problema viene dado porque el volante dispone un motor para poder controlar la dirección del vehículo, en caso de ser necesario hay que conseguir calibrar este motor para obtener el ángulo necesario por sistema de navegación autónoma.

Para ello se tuvieron que obtener primero los valores de los encoders en los límites del volante. Se pudo comprobar que cuando el volante estaba girado completamente a la izquierda se obtenía en el encoder un valor de 2.674. Cuando el volante estaba en la posición central y las ruedas rectas, se obtenía un valor de 3.100 (éste es un valor aproximado puesto que si se giran el volante un poco, aunque las ruedas no se mueven, hay una variación de unas 20 cuentas). Por último, cuando el volante se giraba por completo a la derecha se obtenía un valor de encoder de 3.603 cuentas.

Fue necesario realizar la calibración precisa del volante puesto que, tal como se puede comprobar, con el giro a izquierdas se tiene una variación en las cuentas de encoder más pequeño que con los giro a derechas. De esta forma, si se utilizaba la misma recta de calibración se obtenían unos límites de giro diferentes a cada lado, aunque se girara realmente lo mismo.

Primera metodología:

Con la primera metodología que se utilizó para obtener tratar de obtener la calibración del volante se buscó obtener una relación directa entre un determinado valor del encoder y el ángulo de giro.

El método consideraba que el vehículo se encontraba en una posición determinada con un ángulo de giro del volante constante. A continuación se movía el vehículo hasta que éste realizaba un giro de 180º, midiéndose por último la distancia entre la posición inicial y final del vehículo.

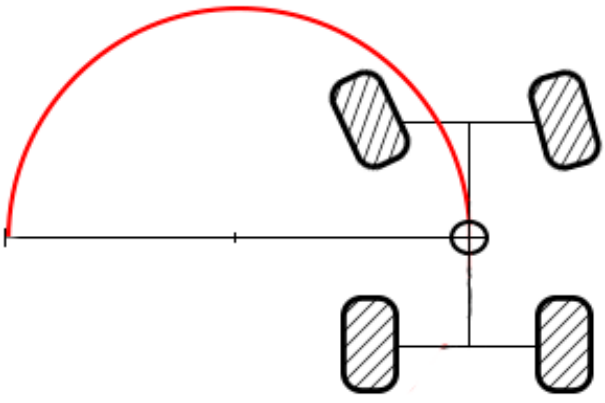

Figura 24. Primera metodología de calibración.

A partir de las ecuaciones del vehículo:

$$Angulo = \frac{Distancia\ entre\ ejes}{\sqrt{\left(\frac{Rcentro+Distancia\ ancho\_coche}{2}\right)^2 + Distancia\ al\ centro}} \quad (2)$$

y sustituyendo los valores de las constantes del vehículo:

$$Angulo = \frac{1{,}65}{\sqrt{\left(\frac{Rcentro+0{,}55}{2}\right)^2 + 0{,}7225}} \quad (3)$$

Hay que tener en cuenta que la ecuación (3) no devuelve un resultado con signo. Esto se ha hecho para poder adaptar el signo del resultado en función del convenio que indica los REP de ROS. De esta forma en la plataforma se considera que cualquier giro a izquierdas es positivo y cuando se realiza a derechas es negativo.

En la siguiente tabla podemos observar el resultado que se ha obtenido para la calibración del volante.

encoder	Media-vuelta(metros)	Angulo con signo
2674	3,68	0,650463852
2700	4,30	0,582907875
2725	4,80	0,537456432
2750	5,60	0,477409292
2774	6,50	0,423739098
2874	9,44	0,309098264
2974	16,60	0,185586654
3125	0,00	0
3300	13,66	-0,222108891
3400	9,44	-0,309098264
3500	6,24	-0,437997189
3525	6,00	-0,452012386
3550	5,60	-0,477409292
3575	5,05	-0,517189923
3603	3,63	-0,656556852

Tabla 5. Resultados primera metodología de calibración.

Como podemos observar se obtienen unos ángulos muy similares en ambos extremos a pesar del diferente valor de las cuentas de encoder a un lado y al otro del volante.

Si se grafican los resultados obtenidos con este método se puede comprobar que la zona central es bastante lineal, por lo que se podría fijar la calibración mediante una línea recta. Sin embargo, en los valores extremos del volante hay un cambio de pendiente muy pronunciado.

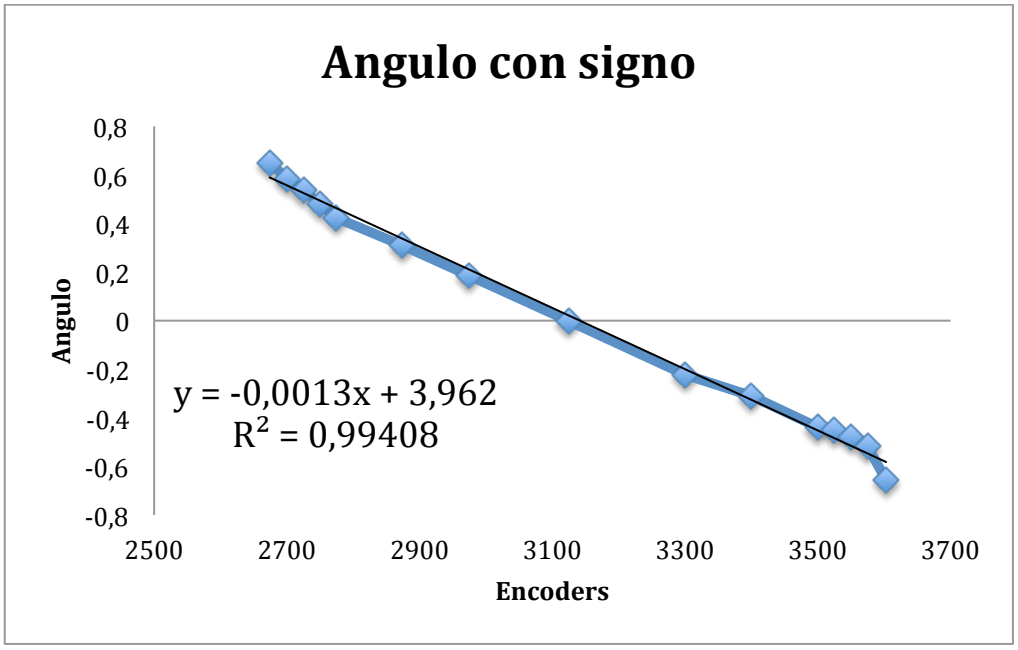

Figura 25. Recta de calibración con el primer método utilizado.

Una posible opción sería utilizar una calibración por tramos. Sin embargo, analizando los valores de los extremos se llegó a la conclusión de que tal vez este problema que aparecía se pudiera deber a que esta metodología no es muy precisa, lo que podía inducir a errores en la lectura de los resultados. Esto se debe a que, al estar basado en el movimiento del vehículo, se añade una gran cantidad de errores. Por todo esto se planteó un método alternativo de calibración.

Segunda metodología:

Con esta metodología se pretendía tener un método de calibración más robusto a errores en la medida. Para ello se evitó tener que mover el vehículo.

En este método, después de ubicar las ruedas en el punto central del volante se situó una escuadra de calibración a medio metro delante del vehículo. A continuación se giraba el volante de vehículo, y con la ayuda de una superficie rígida colocada en paralelo a la rueda, se prolongaba la recta hasta intersectar con la escuadra.

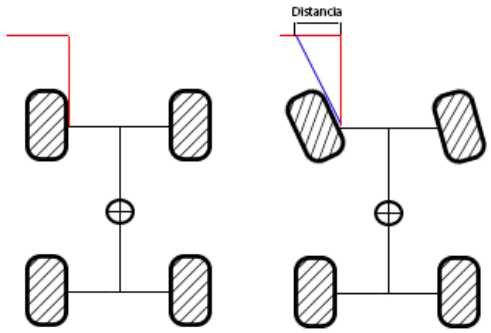

Figura 26. Metodología calibración 2.

El ángulo de giro del vehículo para este método viene dado por la siguiente ecuación

$$Angulo = \frac{1}{\frac{Distancia\_Rueda\_CintaPapel}{Medida\ Distancia} + \frac{Ancho\ coche}{2 \cdot Distancia\ entre\ ruedas}} \quad (4)$$

Considerando las constantes del vehículo, la ecuación (4) se transforma en la expresión siguiente:

$$Angulo = \frac{1}{\frac{0,5}{Medida\ Distancia} + \frac{1,10}{2 \cdot 1,65}} \quad (5)$$

Con este método de calibración se realizaron un conjunto de pruebas. Además, al ser un método más simple y rápido, se pudieron realizar un número mayor de medidas. La Tabla 6 muestra los valores obtenidos para las pruebas de calibración

encoders	medida	Angulo	encoders	medida	Angulo
2665	45	0,692307692	3120	1	-0,01986755
2675	42,4	0,661122661	3130	1,5	-0,02970297
2685	39,4	0,62407603	3140	2	-0,039473684
2695	37,3	0,597437266	3150	2,5	-0,049180328
2705	34,4	0,559652928	3160	3	-0,058823529
2715	32,8	0,538293217	3170	3,4	-0,066492829
2725	31	0,513812155	3180	3,6	-0,0703125
2735	30	0,5	3190	4	-0,077922078
2745	29	0,48603352	3200	4,5	-0,087378641
2755	28,3	0,476163769	3225	5,5	-0,106109325
2765	26,6	0,45186863	3250	7,6	-0,144670051
2775	26,2	0,446083995	3275	10,4	-0,194513716
2800	23,4	0,404844291	3300	13,4	-0,246022032
2825	21,3	0,373029772	3325	16,5	-0,297297297
2850	19,4	0,343565525	3350	18,4	-0,327790974
2875	17,5	0,313432836	3375	20,8	-0,365339578
2900	15,5	0,280966767	3400	23,5	-0,406340058
2925	13,4	0,246022032	3425	25,7	-0,438816164
2950	10,9	0,203231821	3450	28,6	-0,480403135
2975	8,7	0,164461248	3475	31,1	-0,515184981
3000	6,9	0,131931166	3500	34,5	-0,56097561
3010	6,4	0,122762148	3510	35,5	-0,574123989
3020	5,6	0,107969152	3520	36,3	-0,584541063
3030	4,8	0,093023256	3530	37,2	-0,596153846
3040	4,1	0,0798183	3540	40	-0,631578947
3050	3,5	0,068403909	3550	42	-0,65625
3060	2,6	0,051114024	3560	43,9	-0,679216091
3070	2	0,039473684	3570	46,4	-0,708757637
3080	1,5	0,02970297	3580	48,6	-0,734138973
3090	1	0,01986755	3590	49,8	-0,747747748
3100	0,4	0,007978723	3600	52,4	-0,776679842
3110	0	0			

Tabla 6. Tabla Resultados Metodología 2.

Representando gráficamente las medidas obtenidas se puede comprobar que la respuesta obtenida es mucho más lineal que la de la metodología anterior, por lo que mediante una única recta se puede calibrar la respuesta del volante del vehículo.

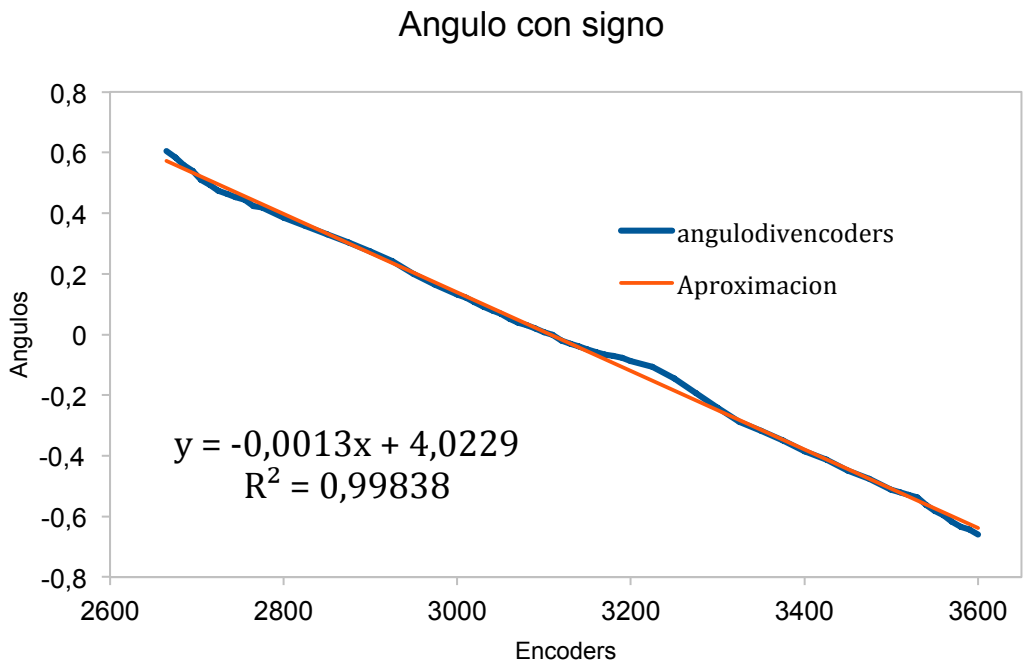

Figura 27. Recta de calibración con el segundo método utilizado.

La implementación la recta de calibración es muy simple puesto que sólo se debe programar la ecuación de la recta obtenida. De esta forma, utilizando como ***x*** el valor del encoder leído, se puede calcular el ángulo resultante (valor ***y***).

$$y = -0.0013x + 4{,}0229 \tag{6}$$

Si se necesita poder calcular la posición de encoders en función del ángulo, solo habría que despejarla directamente a partir de la ecuación (6).

$$x = -\frac{y-4{,}0229}{0.0013} \tag{7}$$

3.2) Configuración del robot.

En el siguiente apartado se va a realizar una visión del trabajo realizado sobre el software necesario para poder disponer de un control correcto sobre el vehículo. Robotnik, la empresa encarga de fabricar el vehículo, aportó una serie de paquetes básicos para el funcionamiento del vehículo:

- Un paquete que se utiliza para cargar toda la configuración referente a los diferentes marcos (frames) y transformadas.
- El nodo rbcar_controller, el cual cuenta con dos nodos más, uno para controlar el volante y otro para controlar el motor.

Por lo tanto, una de las primeras tareas que se tuvo que realizar fue la de verificar el correcto funcionamiento de cada uno de los componentes que fueron suministrados.

3.2.1) Robot config.

Cuando se aborda un proyecto de este tipo, lo primero que es necesario hacer es disponer de un fichero de configuración del sistema. Para ello es necesario recordar que éste debe cumplir con los requisitos especificados por la norma *REP 105* a la hora de definir los diferentes *frames* de trabajo.

En este sentido Robotnik suministró un conjunto básico de ficheros de configuración

- ***rbcar.urdf.xacro***: incluye tres ficheros diferentes *rbcar_base*, *suspensión_wheel*, y *all_sensors*, estableciéndose la siguiente jerarquía:

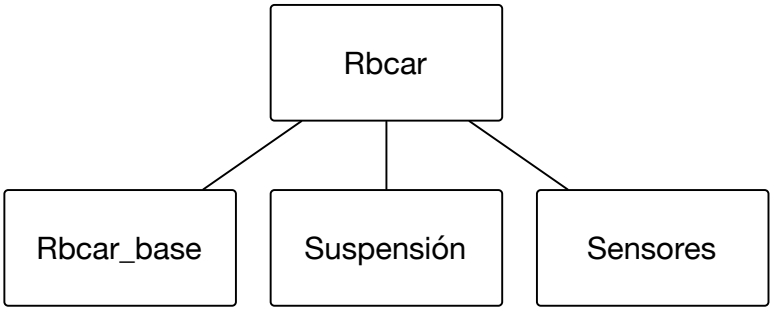

Figura 28. Diagrama de distribución del fichero xacro.

En el fichero se encuentra también diferentes definiciones de los sensores del vehículo. Estas definiciones servirán para poder calcular las diferentes transformadas disponibles en el vehículo. Estas transformadas están referenciadas a *base_link*. Por lo tanto, para realizarlo correctamente se debe tener una definición de dónde se encuentra la definición de *base_link*.

```
<xacro:include filename="$(find
rbcar_description)/urdf/bases/rbcar_base.urdf.xacro" />
<xacro:include filename="$(find
rbcar_description)/urdf/wheels/suspension_wheel.urdf.xacro" />
<xacro:include filename="$(find
robotnik_sensors)/urdf/all_sensors.urdf.xacro" />
```

```
<xacro:sensor_hokuyo_urg04lx name="hokuyo1" parent="base_link">
    <origin xyz="0.9 0.0 0.75" rpy="0 0 0"/>
</xacro:sensor_hokuyo_urg04lx>
```

Se ha adjuntado las líneas que tienen más relevancia en el fichero xacro, puesto que es cuando se incluye los diferentes ficheros que contienen datos de configuración. Además también he añadido las líneas que se gastan para añadir un sensor en la configuración, donde *name* es el nombre de la transformada que nos publicara, *parent* nos sirve para conocer cual es el padre de la transformada, la siguiente línea nos sirve para colocar el sensor en el espacio, así como también indicarle que orientación tiene el sensor.

En la figura siguiente se puede apreciar dónde están situados los diferentes sensores y qué convenio ha sido utilizado (respecto a orientación/posición) de los diferentes ejes de coordenadas.

Figura 29. Imagen RBCAR con transformadas.

Para ver cómo están estructuradas las transformada se puede observar cómo quedaría el nuevo árbol de transformadas del vehículo. En el caso del vehículo, el árbol de transformadas es el siguiente:

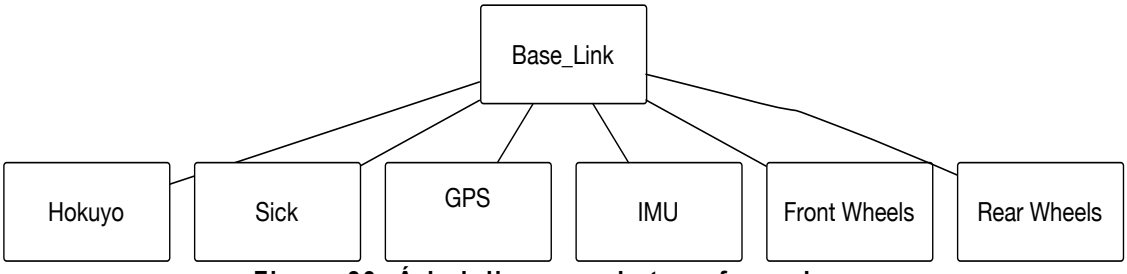

Figura 30: Árbol diagrama de transformadas.

En dicho árbol falta por definir tres trasformadas importantes:

- ***Base_footprint***: Si nos fijamos en la figura 29, podemos observar que *base_link* en encuentra en el centro de masas del vehículo. *base_footprint* es el mismo centro de masas del vehículo en cuanto en posición X e Y, pero representado sobre el suelo del sistema. Por ello se considera que éste es equivalente a *base_link* con un Z igual a cero.
- **Odom**: esta transformada indica la distancia desde el origen de coordenadas del mapa, hasta la posición actual del vehículo.
- **Map**: el frame de coordenadas map establece el origen del sistema de coordenadas. En algunos casos esta transformada puede estar referenciada a un punto fijo, pero también puede estar referenciado a un punto relativo que puede cambiar cada vez que arranca del sistema.

Para poder ver el árbol de transformadas de un sistema se puede utilizar el nodo *view_frames* que forma parte del paquete *tf* de ROS:

Para utilizar el nodo view_frame tenemos que introducir en la terminal el siguiente comando:

$ rosrun tf view_frames.

Una vez se ha lanzado hay que esperar 5 segundos, mientras el nodo view_frame esta escuchando los diferentes topics y transformadas, hasta generar un fichero con extensión pdf con el siguiente árbol de transformadas.

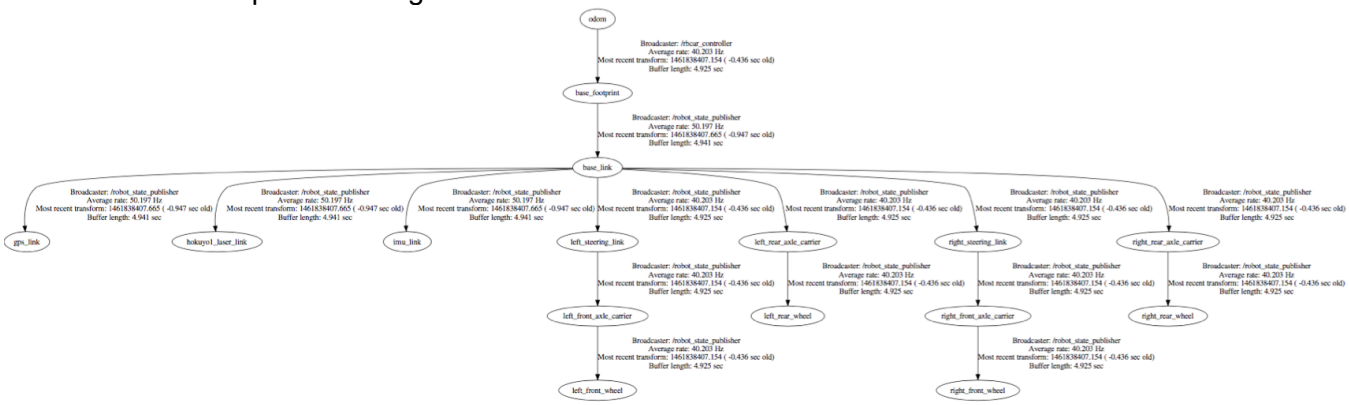

Figura 31. Árbol de transformadas completo.

Para poder apreciar mejor el árbol de transformadas, éste se va a analizar por partes. En la figura siguiente se puede observar las transformaciones entre los frames de *odom* y *base_link*. Entre los *frames* se puede observar qué transformaciones se tienen. En esta información se puede ver qué nodo la publica (el *Boardcaster*), y la frecuencia (*Average rate*), el tiempo que tiene la última transformada (*Most recent transform*) y el *Buffer length*, donde se especifica cuanto tiempo mantiene una transformadas).

Referente a *Most recent transform* se puede ver un número muy largo que representa el tiempo del sistema y entre paréntesis cuánto tiempo tiene la transformada.

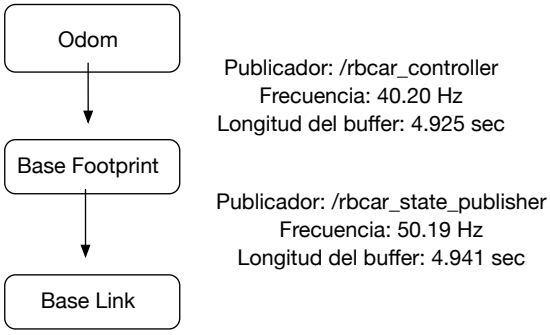

Figura 32. Transformadas de odom a base_link.

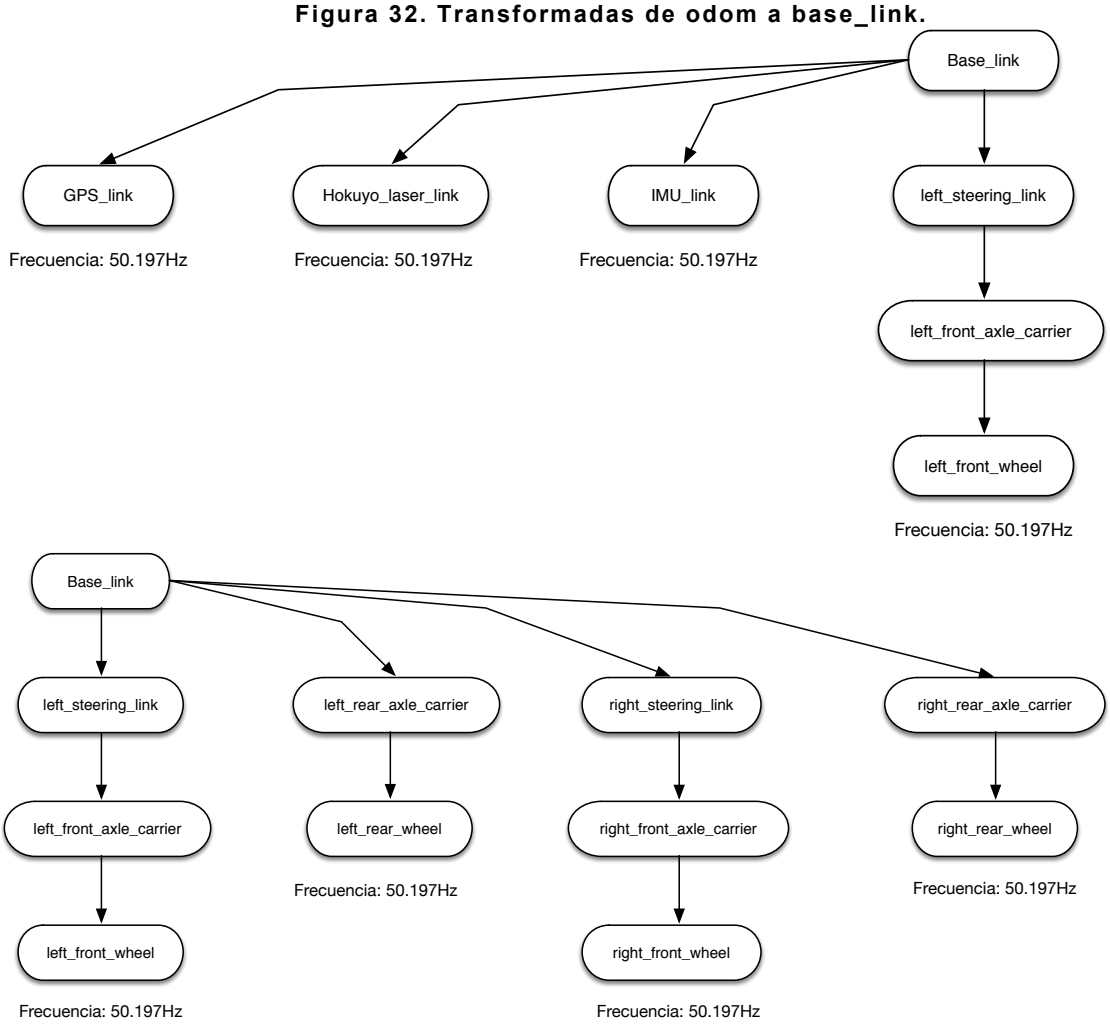

Figura 33. Detalles del árbol de transformadas.

Todas las transformadas se han configurado para que se actualicen a una frecuencia de 40Hz. y para que sean capaces de guardar de una cantidad de 4'92 segundos. Por lo tanto, con la configuración actual, el sistema no podrá superar en ningún caso unas modificaciones superiores a 40Hz. puesto en caso de hacerlo no se puede asegurar que todos los procesos se ejecuten correctamente.

3.2.2) Volante.

Una vez se dispone de la configuración de transformadas del vehículo, el siguiente paso que se tuvo que hacer fue comprobar el funcionamiento correcto del código. Para verificar que el nodo del volante funciona correctamente se deben seguir los siguientes pasos:

1. Asegurar que se lanza la ejecución del nodo. Para ello se debe lanzar un ros master y posteriormente lanzar el nodo del volante: ***rbcar_steering_controller***. Las instrucciones para realizar este paso son las siguientes:
    - $ rosrun roscore &
    - $ rosrun rbcar_steering_controller rbcar_steering_controller

2. A continuación es necesario comprobar que se reciben correctamente los datos de configuración a través de los parámetros de ROS, como por ejemplo el puerto al que esta conectado el volante o cual es el nombre del dipositivo.

3. Se debe comprobar también que el nodo es capaz de publicar y recibir mensajes a través de los topics. Para este caso se tienen tanto subscriptores (*subscribers*) como publicadores (*publishers*).

   Para realizar la comprobación se debe lanzar el nodo y posteriormente ejecutar el comando *rostopic echo* sobre el topic en que se publica. Para probar el subscriptor se recomienda que el nodo publique un mensaje, de forma que en la función *callback* de éste se puede poner un ROS_INFO(). De esta forma devolverá por pantalla el valor de las variables enviadas mediante el mensaje, en este ejemplo se puede observar un caso de subscripción al nodo que publica el ángulo del volante, mediante la utilización del ROS_INFO nos los muestra por pantalla, el valor que ha recibido por el topic.

   ```
   void cmdAngleCallback(const std_msgs::Float64::ConstPtr& cmd)
   {
           ROS_INFO("rbcar_steering_controller::cmdPositionCountsCallback:
   value = %d",cmd->data);
   }
   ```

4. Comprobar que el funcionamiento de las diferentes funciones sea el correcto.

Por otro lado, las funciones que componen el programa del volante son las siguientes:

- ***cmdAngleCallback***: se encarga de recibir el ángulo en el que se encuentra el volante.
- ***cmdPositionCountsCallback***: se encarga de devolver un valor entero que muestra el valor del encoder que nos devuelve el volante.
- ***readAndPublish***: se encarga de coger los valores recibidos del motor y el valor del encoder y lo convierte en un valor de ángulo.
- ***encoderCountsToAngle***: recibe como entrada el valor de la posición del encoder y se encarga de transformar y devolver el valor de ángulo. Para ello

esta función utiliza la recta de calibración que se obtuvo de manera práctica en el apartado *3.1.1) Calibración del volante*.

- ***setSteeringPosition***: se encarga de transformar el valor del ángulo recibido en un valor de cuentas de encoder. Dicho valor se utilizará para poder mover el volante hasta la posición que se desee.

Las tres primeras funciones se basan en el método de calibración del apartado 3.1.1. Para las funciones ***encoderCountsToAngle*** y ***setSteeringPosition*** se utilizó una recta de comprobación como la calculada en el apartado teórico 3.1.1.

3.2.3) Motor.

Para el control del motor del vehículo Robotnik proporcionó un nodo llamado ***curtis_controller***. En este caso no se tuvieron que realizar ninguna modificación en este nodo puesto que en las diferentes pruebas que se hicieron se puedo comprobar que cumplía con las especificaciones requeridas.

Para el funcionamiento de este nodo se disponen de los parámetros siguientes:

- ***Wheel_diameter:*** el diámetro de la rueda, en este caso es 0,5m.
- ***Max_rpm:*** El máximo de revoluciones que puede funcionar el motor: 4.300rpm.
- ***Max_v_auto:*** velocidad máxima en modo autónomo expresado en m/s. El valor por defecto es 2,5 m/s
- ***Gear_ratio:*** Parámetro para convertir velocidad del motor a la velocidad del coche. Para este vehículo el valor es de 16,0.

3.2.4) Controlador del coche.

Otro de los paquetes suministrado por Robotnik es el nodo encargado de controlar el vehículo. La función de este nodo es recibir las órdenes respeto al vehículo y se deberá enviar a los nodos del volante y del motor para ejecutar las acciones.

Otra de las funcionalidades de este nodo es la de calcular la odometría del vehículo, calculándose dicho valor a partir de los valores obtenidos desde los diferentes sensores.

El controlador tiene las siguientes entras:

- ***Command:*** Al subscribirnos en este topic se recibe qué debería hacer el vehículo. Para ello se utiliza un mensaje del tipo *AckermannDriveStamped*. El *header* de este tipo de mensaje tiene la información de qué *frame_id* se ha utilizado para calcular los datos del mensaje y cuándo se ha hecho:

  uint32 seq
  time stamp
  string frame_id

  seq es un identificador único del mensaje. Dicho valor solo puede incrementarse. *stamp* es el tiempo en el que se ha enviado el mensaje (expresado en segundos o nanosegundos). *frame_ID* indica respecto a qué *frame* se ha calculado los siguientes datos:

  float32 steering_angle
  float32 steering_angle_velocity
  float32 speed
  float32 acceleration

  donde steering_angle es la variable de donde vamos a indicar que valor de giro queremos, steering_angle_velocity sirve para indicar con que velocidad queremos cambiar la posición del volante, speed sirve para indicar la velocidad a la que queremos que vaya y acceleration es la aceleración que queremos que tenga el vehículo.

- ***Command differential:*** una forma alternativa para indicar qué movimiento debe hacer el vehículo es mediante un mensaje de tipo de configuración diferencial. De hecho, en robótica móvil es el tipo de mensaje más común. El tipo de este mensaje es *TwistStamped*, y está compuesto por el apartado header. En este caso se tienen dos tipos de velocidades: la velocidad lineal y la angular, indicándose en ambos casos con vectores en X,Y,Z.

  float64 x
  float64 y
  float64 z

  En este tipo de mensajes la variables están relacionadas entre ellas, puesto que un valor en el motor constante puede verse modificado en una componente según se tenga posicionado el volante. Además, las velocidades en Z serán nulas puesto que no se contempla los cambios de altitud.

- ***Curtis data:*** se trata de un topic que recibe la información del nodo del motor. Mediante el topic se obtiene información relativa de los motores, como a qué velocidad se está moviendo o cuál es el estado del motor.

- ***Steering Controller:*** este topic recibe la información desde el nodo del volante y permite obtener el estado del motor del volante mediante el valor del encoder.

- ***Set Odometry:*** se trata de un servició que cuando se llama permite reconfigurar la información de la odometría.

Por otro lado, el nodo del controlador también publica diferentes informaciones mediante diferentes topics. Los publicadores que tiene el controlador son:

- ***State:*** por este topic se publica información referente al estado del controlador, indicando si se encuentra en reposo o ejecutando alguna acción.

- ***Odom:*** este nodo publica qué recorrido está realizando la odometría, y se calcula a partir de los diferentes sensores que se tienen en el vehículo. Esta información se publica también respecto del *frame* base a través de diferentes transformadas.

- ***Cmd_vel_pub:*** este es el topic que publica sobre el nodo del motor, enviándose finalmente la información referente a la velocidad representada en forma diferencial.

- ***Cmd_pos_pub:*** en este topic se publica la información del volante, para indicarle qué posición debe tener.

Después de analizar la información que intercambia este nodo se va a explicar qué métodos se han utilizado para certificar el funciona correcto, así como la frecuencia y amplitud. De esta forma se puede comprobar a la frecuencia mínima a la que debe funcionar.

Para comprobar este funcionamiento se ha capturado un fichero ".bag" que incluye los datos de entrada y salida del controlador de cada uno de los topics de entrada. De esta forma, en la siguiente figura se puede observar la entrada (en verde) la salida del mismo (y en color naranja). En esta prueba se buscaba además obtener el límite de giro del volante.

Se puede observar como la salida tiene 10 veces más mensajes que la señal de entrada, funciona el nodo con un modo de *sampler and holder*.

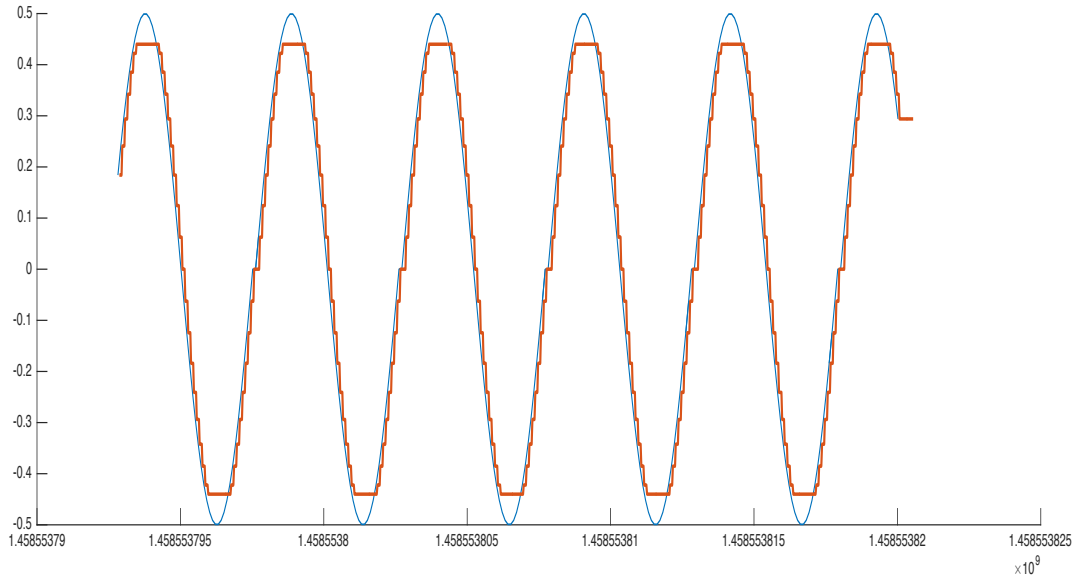

Figura 34. Respuesta antes y después del controlador variable volante.

En la siguiente imagen se puede observar el valor de velocidad absoluta (en color naranja). En color verde se muestra la suma de las componentes lineales y angulares de la velocidad, por lo tanto funciona de la forma esperada.

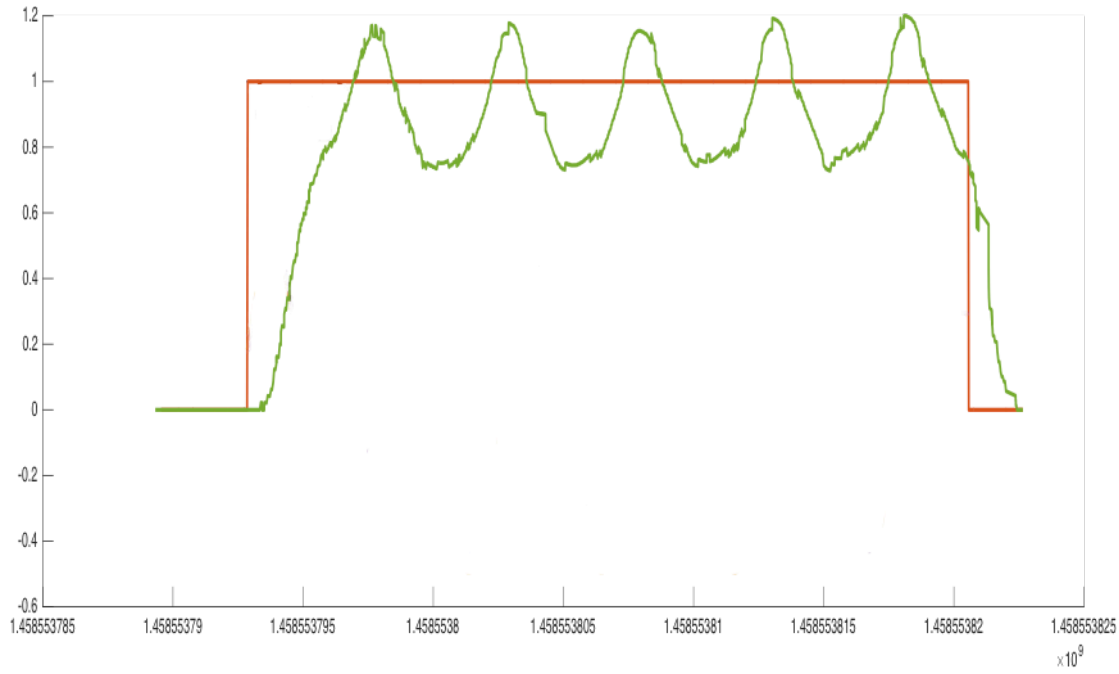

Figura 35. Respuesta

3.2.5) Sensor GPS.

Un tipo diferente de sensor que se ha utilizado es un GPS, desarrollándose para él un nodo que pudiera suministrar la información. Para poder verificar el funcionamiento correcto del nodo se realizaron diferentes recorridos por la Universidad, almacenándose los datos en un fichero ".bag". Las figuras siguientes muestran algunos de los resultados obtenidos.

Figura 36. Zonda de donde adquirimos los valores de GPS.

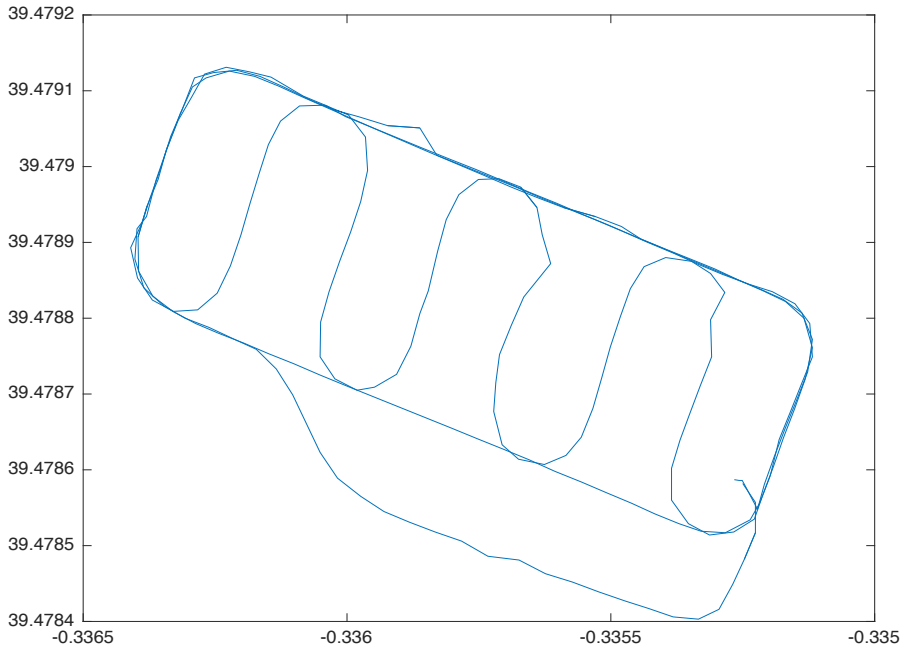

Figura 37. Representación valor en Matlab.

3.3) Paquete de navegación.

Cuando se trabaja con robots móviles y vehículos, uno de las cuestiones más importantes e interesantes es cómo se puede abordar la navegación autónoma. Tal y como se ha podido comprobar en el apartado 2.4 del desarrollo teórico de este trabajo, la gran mayoría de estos desarrollos llevados a cabo constan de los mismos bloques: un localizador y al menos dos planificadores, uno global y otro basado en la planificación local.

El desarrollo que se presenta en este trabajo estará basado en el paquete estándar de navegación de ROS, pero contemplando algunas modificaciones. El paquete de navegación de ROS incluye un planificador global y otro local, y dos mapas de coste. Uno de estos mapas es global y el otro es un mapa de costes local.

El posicionamiento (también llamado localizador) no se encuentra incluido en el paquete inicial pero sí que se puede (y debe) utilizar en el paquete de navegación. De esta forma, en la figura siguiente se muestra el esquema de navegación que proporciona ROS.

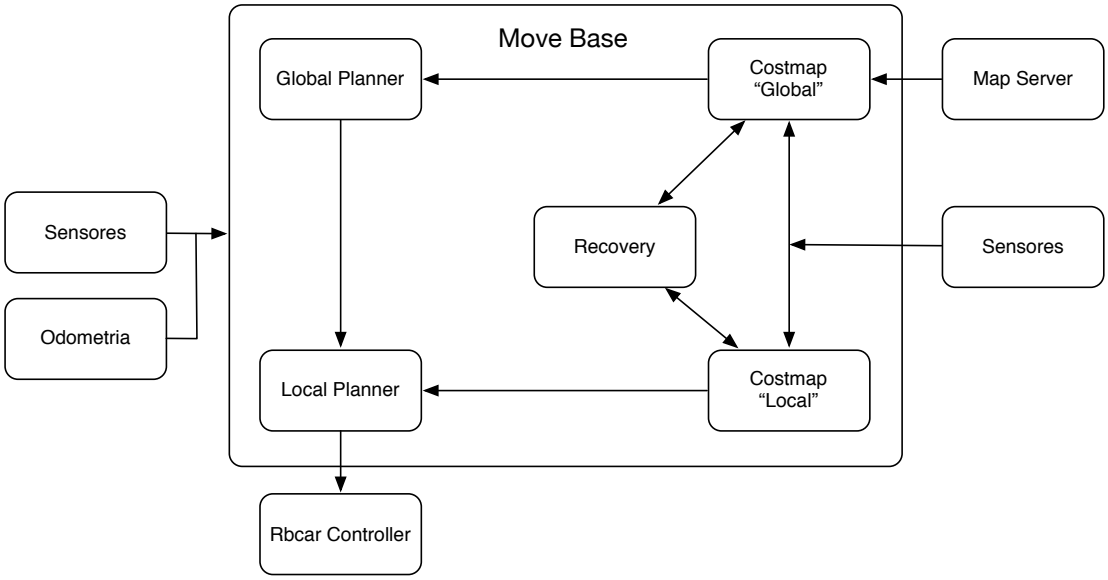

Figura 38. Esquema navegación ROS.

Dependiendo del tipo de aplicación que se desee (navegación de interiores, navegación de exteriores, tipo de algoritmo de planificación del cálculo de ruta, etc.), en este trabajo se han desarrollado diferentes configuraciones.

De esta forma, en el primer diagrama se puede observar la configuración desarrollada para la navegación de exteriores.

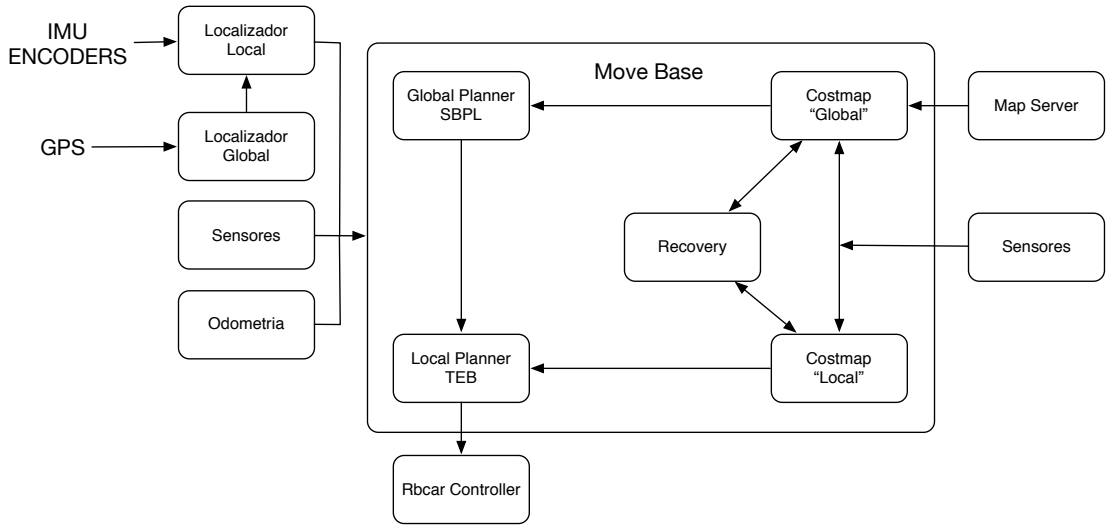

Figura 39. Distribución paquetes navegación de exteriores.

Como se puede observar, en este diagrama de navegación se han incluido dos localizadores: el localizador local y el global.

Una segunda versión que se ha desarrollado es la navegación en entornos conocidos, definida también como navegación de interiores. Esta configuración se muestra en el diagrama del paquete de navegación siguiente.

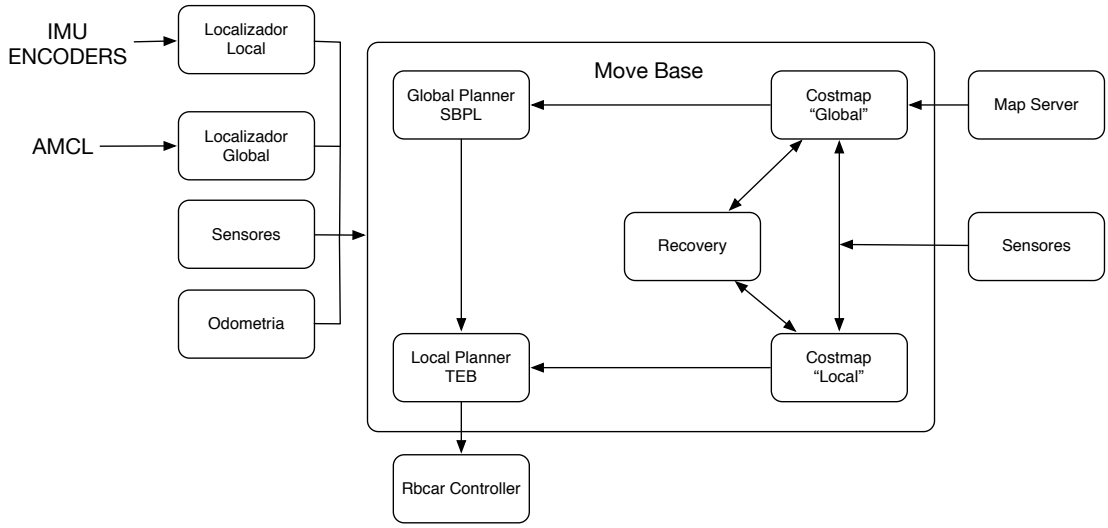

Figura 40. Distribución paquetes navegación de interiores.

En esta configuración se puede observar que no existe (a diferencia de la navegación de exteriores) una realimentación entre el localizador global y el local, utilizándose un algoritmo de Monte-Carlo para la localización global.

Por otra parte, en este proyecto se han utilizado varios planificadores diferentes. Se ha utilizado como planificador global el algoritmo Search-Based Planning Lab (*SBPL)*. Como planificador local se ha utilizado el algoritmo *The Elastic Band (TEB)*.

3.3.1) Localizador.

Cuando se aborda la navegación automática, una de las cuestiones más importantes es poder asegurar dónde se encuentra el robot en cada momento. Esta localización no sólo debe obtenerse en tiempo real (on-line), sino que además debe de proporcionar una información correcta, puesto que de otra forma se obtendría una navegación errática en la cual el vehículo podría no alcanzar el destino especificado.

Para intentar resolver esta problemática se pueden desarrollar varios tipos de localizadores. Dependiendo del tipo de aplicación o del entorno, unos serán más o menos útiles que los otros.

En este trabajo se ha desarrollado un localizador de exteriores basado en los datos obtenidos desde dispositivos GPS. Para el localizador de interiores se dispone de un localizador que compara el mapa por donde circulará el vehículo con los datos obtenidos desde un dispositivo láser.

Para el caso de la navegación de interiores, la conexión entre los localizadores es la siguiente:

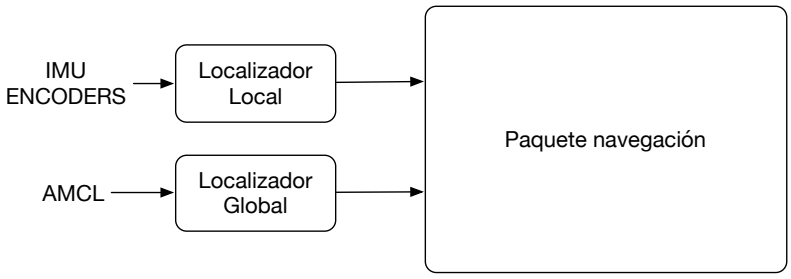

Figura 41. Conexión localizador interior.

El localizador local funciona en modo continuo a una frecuencia mayor que el localizador global. El localizador global funciona en modo discreto de forma que cada cierto tiempo publica una posición que sirve para corregir la aportada de forma continua por el localizador local.

Otra de las diferencias es que el localizador global en este caso es el AMCL. Este algoritmo debe devolver una nube de posibles ubicaciones del vehículo. Para cada uno de esos puntos debe tener una covarianza.

Para el caso de la navegación de exteriores la configuración es la siguiente.

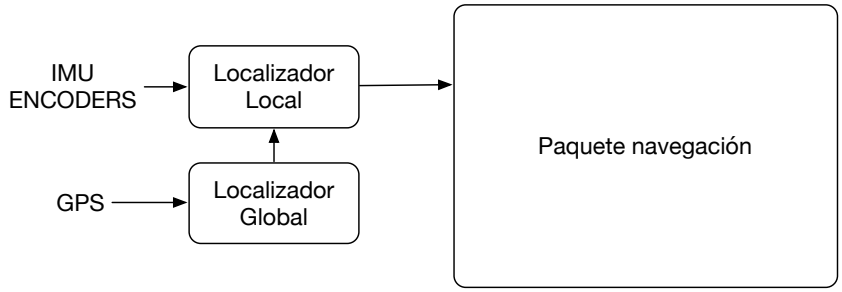

Figura 42. Conexión localizador exteriores.

En esta configuración se puede observar que existe una comunicación entre el localizador global y el local. Es el localizador local el que ofrece la posición del vehículo, de forma que a partir de la información del GPS periódicamente se corrige la información del localizador local.

La implementación de los localizadores ha basado en el paquete *robot_localization* proporcionado por ROS.

3.3.1.1) Robot Localization.

El principal paquete gastado para la localización del vehículo es el "*Robot Localization*" que fue desarrollado inicialmente por la empresa *Charles River Analytics.* Actualmente este paquete se encuentra sobre la licencia BSD (link en bibliografía).

robot localization se encarga de obtener los datos de los diferentes sensores y de devolver una odometría filtrada. Para ello el nodo proporciona una implementación de un Filtro de Kalman Extendido (EKF) que realiza una fusión de los datos obtenidos desde diferentes sensores.

Los parámetros de configuración más destacados de este paquete son los siguientes:

- ***frecuency*:** con este parámetro se indica a qué frecuencia (en Hz.) de la salida del nodo. Es necesario tener en cuenta que el nodo no empezará a publicar la salida hasta que no reciba alguna información proveniente de los sensores.

- ***Sensor timeout*:** es el periodo de tiempo que debe de pasar para considerar que un sensor está inhabilitado. Si se cumple este tiempo se ignora el sensor y no se tiene en cuenta para calcular la salida.

- ***Two d mode:*** este parámetro sirve para trabajar en un plano 2D (x, y). En el caso de tener sensores que nos aporten información 3D el plano Z se ignorará.

En el fichero de configuración de *robot localization* contiene por cada sensor del sistema una matriz de configuración para indicar qué valores se quieren utilizar. La distribución de la matriz de configuración es la siguiente:

X	Y	Z
Roll	Pitch	Yaw
V_X	V_Y	V_Z
V_{Roll}	V_{Pitch}	V_{Yaw}
A_X	A_Y	A_Z

Cada uno de los elementos de la matriz se puede sustituir por *TRUE* o *FALSE* en función de si se quiere o no utilizar.

En este nodo se puede configurar también el ruido producido por el filtrado. Como el suministrador recomienda no cambiar estos valores, en este trabajo se ha optado por no modificar este ruido.

Para el caso de la localización de interiores se dispone de una IMU y de la odometria que proporciona el controlador del coche. Es este caso la configuración del localizador local es la siguiente:

```
<param name="frequency" value="30"/>
<param name="sensor_timeout" value="0.05"/>
<param name="two_d_mode" value="true"/>
<rosparam param="odom0_config">[true, true, false,
                                false, false, true,
                                true, true, false,
                                false, false, false,
                                false, false, false]</rosparam>


<rosparam param="imu0_config">[false, false, false,
                               false, false, false,
                               true, true, false,
                               false, false, false,
                               false,  false, false]</rosparam>
```

En este fichero de configuración se puede observar que se va a coger de la odometria la posición (X, Y), y la rotación sobre el eje Z (YAW) y las velocidades en X e Y. De la unidad inercial se coge la velocidad en X e Y.

De esta forma se realizará la fusión de la velocidad en X e Y de dos sensores diferentes, lo que podrá permitir eliminar errores en la odometria como pueden ser los producidos por las faltas de adherencia que sufre a veces el vehículo.

En el caso de querer trabajar con la navegación en exteriores se deben realizar dos configuraciones: la configuración del localizador local y la del localizador global.

En el caso del localizador local, la única modificación es añadir la odometria filtrada que es proporcionada por el localizador global. Para indicarle de qué topic tiene que coger la información se utiliza la función *remap*, siendo primero el parámetro interno del nodo en este caso “/odometry/filtered” y diciéndole qué valor debe tener, en este caso “/rbcar_odom_filtrada_global”.

```
<remap from="/odometry/filtered" to="/rbcar_odom_filtrada_goblal"/>
<rosparam param="odom0_config">[true, true, false,
                                false, false, true,
                                true, true, false,
                                false, false, false,
                                false, false, false]</rosparam>
```

En la configuración del localizador global, como entrada se tiene la odometria proporcionada por el controlador del vehículo. Se dispone también de la posición proporcionada por el GPS. Para poder utilizar este valor se debe utilizar el nodo que encargado de convertir de coordenadas GPS a coordenadas UTM (más información en la bibliografía, UTM), que se basan en posiciones (X, Y) calculadas en metros.

Para poder añadir este parámetro se ha utilizado la otra posibilidad que nos ofrece *robot Localization*: la posibilidad de indicar una posición. Para utilizar *Pose0* es necesario que el sensor GPS proporcione la posición. Los parámetros vuelven a ser

los mismos que en la tabla de configuración, siendo lo mas importante en este caso los dos primeros posición X e Y.

```
<param name="frequency" value="2"/>
<param name="sensor_timeout" value="0.1"/>
<param name="two_d_mode" value="true"/>
<rosparam param="odom0_config">[true, true, false,
                                false, false, true,
                                true, true, false,
                                false, false, false,
                                false, false, false]</rosparam>
<rosparam param="Pose0_config">[true, true, false,
                                false, false, false,
                                false, false, false,
                                false, false, false,
                                false, false, false]</rosparam>
```

En caso de querer gastar estas diferentes configuraciones se recomienda hacerlo mediante el uso de ficheros ".launch", un ejemplo:

$roslaunch rbcar_2dnav localizador_local.launch

El fichero ".launch" es muy útil puesto que evita tener que cargar cada vez los parámetros que se van a utilizar en el nodo (robot Localization, en este caso). El fichero tiene dos partes. Por un lado se carga el nodo y por otro, se cargan los parámetros a a utilizar por el nodo.

```
<launch>
  <node pkg="robot_localization" type="ekf_localization_node" name="local_localization"
clear_params="true">
    <rosparam command="load" file="$(find rbcar_2dnav)/config/robotlocalization/local.yaml" />
  </node>
</launch>
```

3.3.1.2) AMCL.

AMCL (*Approach Monte-Carlo Localization*) es una implementación que calcula posibles posiciones de manera probabilística para robots que se desplazan en un plano 2D. Para poder utilizar este método de localización se necesita disponer del mapa de la zona por la cual el robot está navegando y de un sensor laser para detectar el entorno. Este paquete está distribuido bajo una licencia LGLP(más información en bibliografía).

En la configuración del nodo AMCL se encuentra una larga lista de parámetros con los que se puede ajustar el funcionamiento del mismo. De todos ellos, los más importantes son los siguientes:

- **Min_particles**: el mínimo número de partículas permitidas para considerar como un objeto.

- **Max_particles**: el máximo número de partículas permitidas por objeto.

- **Use_map_topic**: en este parámetro se le indica el nombre del topic por el cual se publica el mapa. Por defecto esta siempre en *false*, pero si la aplicación dispone de un mapa dinámico será conveniente ponerlo a *true*.

- **First_map_only**: este parámetro es para solo tener en cuenta el primer mapa que recibe por el topic en caso de poner una aplicación con un mapa dinámico se debería poner a *false*.

- **Odom_model_type**: este parámetro sirve para elegir qué tipo de odometria se va a utilizar. En este caso se tienen dos opciones "*omni*" y "*diff*". En este proyecto se utiliza "*diff*" puesto que la odometria viene expresada en forma diferencial.

- **Update_min_d**: este parámetro sirve para indicar cada cuanto movimiento de translación se debe recalcular la salida del filtro. Siendo el valor por defecto de 0.2 metros.

- **Update_min_a**: este parámetro es el equivalente al anterior, tratándose en este caso al movimiento angular de rotación. El valor por defecto es de pi/6.0 radianes.

- **Tf_broadcast**: este parámetro es útil por si se desea publicar la transformada desde el *frame global* y el de la odometria. El valor por defecto es true, aunque es este trabajo se ha colocado como *false*, porque esta transformada la la publica otro nodo.

El nodo del AMCL se puede lanzar de dos maneras, con la primera manera cargamos en nodo con la configuración default, mientras que en segundo caso podemos dentro del launch modificar algunos parámetros para que se ajuste más a las necesidades que tenemos:

```
$rosrun amcl amcl
o
$roslaunch amcl amcl.launch
```

Durante el presente proyecto siempre se ha lanzado de la segunda forma puesto que permite cargar una configuración de parámetros personalizada.

En la siguiente figura se muestra la salida del localizador en un instante determinado. En este caso concreto se muestra la situación de partida del vehículo antes de que éste se ponga en movimiento. Se pueden observar varios factores: en verde se representa las medidas obtenidas por el láser. La nube de puntos rojos simboliza posibles posiciones del robot. Según lo que se ha comentado anteriormente, cuando el vehículo empieza a moverse se actuará sobre el filtro y se posicionará el robot.

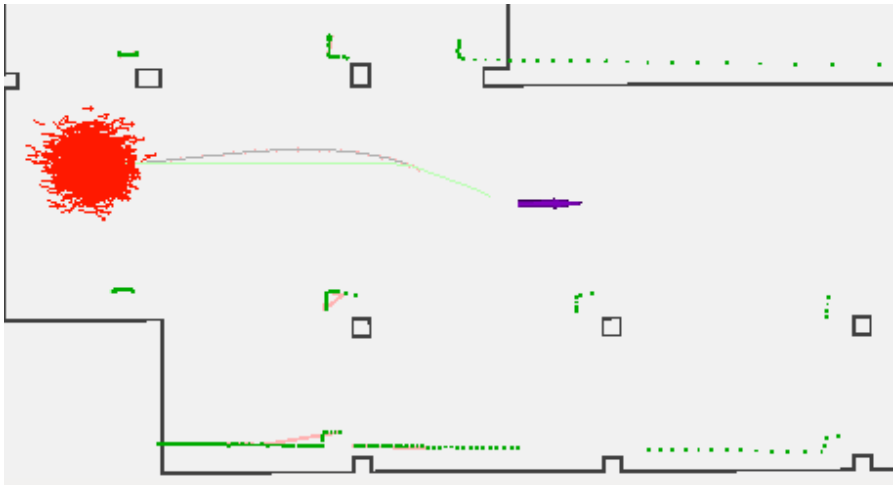

Figura 43. Ejemplo AMCL sin posicionar.

Una vez el vehículo inicia la marcha se puede observar cómo poco a poco se va recolocado el robot respecto de la figura anterior. Se puede observar que la lectura del mapa coincide con las paredes del mapa que se tenían, además, también se observa como la nube con las posibles posiciones se ha reducido.

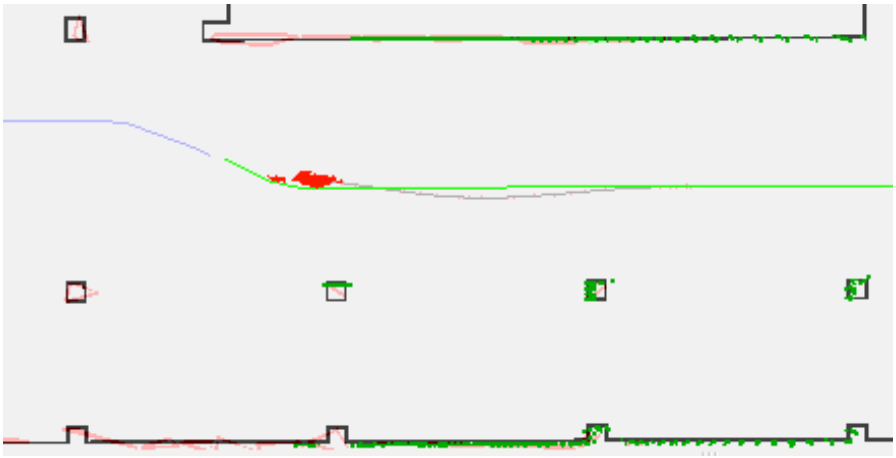

Figura 44. Ejemplo AMCL posicionado.

Como se puede intuir cada uno de los localizadores globales una serie de ventajas respecto del otro. Por un lado, cuando se utiliza el AMCL se obtiene una nube de posibles posiciones del robot. Cuando se utiliza una localización con el GPS nos permite posicionarnos en áreas mucho más grandes, evitando la necesidad de la utilización del mapa de la zona.

3.3.2) Costmaps.

Dentro del cálculo de rutas y de la navegación autónoma seguro que nos encontramos con la necesidad de saber si el vehículo puede circular por un sitio en concreto. Para estos casos, la mejor opción es la utilización este paquete *costmap_2d*. Este paquete es el encargado de generar unos mapas de costes que serán utilizados para poder calcular qué rutas son viables.

En este caso se disponen de varias opciones. La primera es que el mapa de costes sea fijo puesto que está basado en el mapa que se carga para navegar. La segunda opción son los mapas de costes dinámicos, mapas que funcionan basándose en la información obtenida por los sensores.

Para reducir el mapa de costes al máximo lo primero que se realiza es dividir todo el mapa en una cuadricula y se le asignan valores a cada uno de los cuadros.

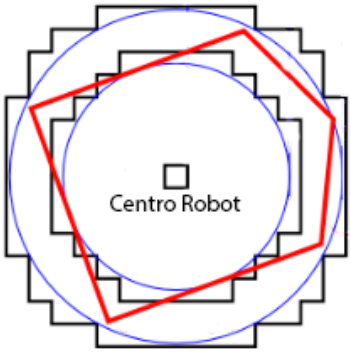

Figura 45. Como se infla las zonas.

En la figura anterior se puede observar en rojo un polígono que nos representa el robot. También se puede observar dos círculos azules. El círculo interior determinaría que el robot ha sufrido una colisión, por lo que el valor del mapa de costes es un alto (254). El segundo círculo (más grande de los dos) es el área donde el vehículo tiene posibilidades de sufrir una colisión. En esta zona el valor del mapa varía desde 128 hasta 253. A partir de la cuadricula definida por el circulo más grande se considera que no hay posible colisión y por lo tanto el valor siempre deberá ser menor de 127 hasta llegar a 1.

En el caso de tener un mapa de costes fijo (también llamado mapa global de costes), este nodo lo que hará simplemente es hinchar o dilatar las paredes y los obstáculos del mapa para que el planificador calcule una ruta que tenga en cuenta los posibles obstáculos.

La configuración para el mapa global de costes es la siguiente:

```
global_costmap:
    global_frame: /map
    robot_base_frame: base_link
    update_frequency: 1.0
    publish_frequency: 1.0
    inflation_radius: 0.40
```

donde:

global_frame indica cuál es el marco (f*rame*) en el que se encuentra el mapa
robot_base_frame indica la transformada principal del coche
update_frequency indica la frecuencia de actualización de los valores del mapa
publish_frequency indica la frecuencia en el que el nodo publica en dirección a los otros nodos el valor del mapa.

El último parámetro (*inflation_radius*) es el más importante. Éste indica la distancia de seguridad que añade a los obstáculos del mapa. En la siguiente figura se puede observar como es el resultado visual de aplicar este nodo, donde en gris veremos la zona de inflado de las paredes/obstáculos que están representadas en negro.

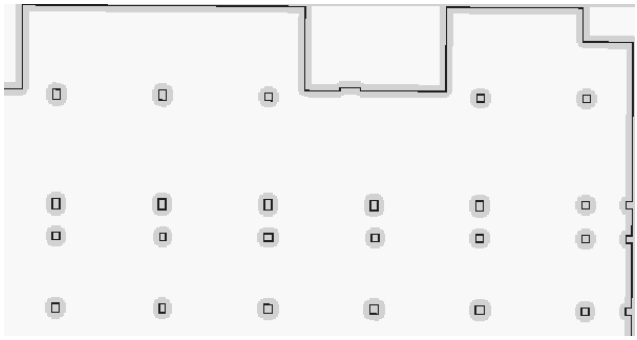

Figura 46. Resultado aplicar mapa global de costes.

El otro mapa de costes que se tiene es el mapa local de costes. Este mapa se basa en la información suministrada por los sensores del vehículo y permite añadir los diferentes obstáculos que inicialmente no estaban en el mapa global. En este proyecto se ha utilizado un sensor láser de rango como fuente de información.
En el mapa local, no se utiliza la información del mapa global, se calcula el mapa de costes local en función de lo que obtenemos por el láser. En cuanto a configuración respecto del mapa global de costes solo varía los siguientes parámetros:

```
local_costmap:
    global_frame: /odom
    static_map: false
    width: 20.0
    height: 20.0
    obstacle_range: 15
    raytrace_range: 15
```

Se puede apreciar como ahora el *global frame* depende de la odometria del vehículo. Además, se añaden algunos parámetros nuevos como *static_map* para indicar que se debe recalcular el mapa cada en cada iteración, *width* y *height* representan el alto y el ancho del mapa (expresados en metros) y *obstacle_range* es la distancia de detección, en la que introducimos obstáculos en el mapa de costes local desde la información obtenida por el láser. Finalmente, *raytrace_range* es a la distancia que eliminamos obstáculos del mapa.

Figura 47. Resultado de aplicar mapa local de costes.

3.3.3) Planificador global.

El bloque del planificador global es uno de los más críticos e importantes puesto que en este componente se calcula la ruta que deberá seguir el vehículo. Por lo tanto se deben evitar rutas imposibles, así como rutas con poco sentido, como por ejemplo realizar excesivas curvas o realizar movimientos bruscos en tramos rectos.

En este trabajo se ha utilizado el planificador llamado *SBPL* (*Search-Based Planning Lab*), que es una implementación de un algoritmo de búsqueda basado en concatenación de primitivas.

En este trabajo las primitivas utilizadas por el algoritmo se han calculado mediante un script de *Matlab*. En este script se calcula, a partir de una resolución tanto espacial como angular, un número de *poses* definidos. En este caso se ha considerado que el volante podría variar en el rango desde -0.6 radianes a 0.6 radianes. La siguiente figura nos sirve para ilustrar los puntos que se calculan.

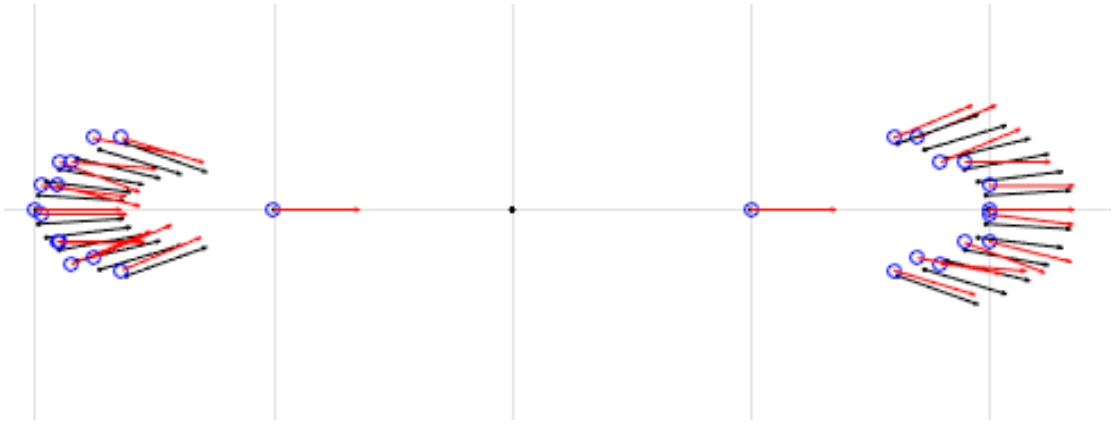

Figura 48. Primitivas calculadas de movimiento.

La figura anterior muestra los posibles movimientos que puede realizar el vehículo, Por una parte las líneas negras son las primitivas calculadas sin tener en cuenta la resolución del sistema tanto. Las líneas rojas son las primitivas calculadas teniendo en cuenta la resolución que admite el sistema.

Al ejecutar el script de *Matlab* se genera un fichero (con la extensión *.mprim*) que tiene el siguiente formato:

```
resolution_m: 0.050000
numberofangles: 16
totalnumberofprimitives: 512

primID: 0
startangle_c: 0
endpose_c: 10 0 0
additionalactioncostmult: 1
intermediateposes: 2
0.0000 0.0000 0.0000
0.5000 0.0000 0.0000

primID: 1
…
```

Las tres primeras líneas son la cabecera del fichero de las primitivas, indicando en este caso que se pueden disponer de 512 primitivas. Este número viene dado por el producto de *numberofangles*, en concreto el producto 16*16 proporciona un total de 256 primitivas, pero como se contempla movimientos hacia detrás, se dispondrá del doble de posibles primitivas (512).

En la segunda parte del fichero de las primitivas disponemos de la siguiente información: *primID* representa el nombre de la primitiva; *startangle_c* indica en qué ángulo empieza la primitiva y *additionalactioncostmult* recoge el coste de aplicar a la primitiva, y las parejas de tres números siguientes nos sirven para indicar los pasos que se realizan la primitiva, en este caso es el punto de inicio y donde acabamos después de realizar el movimiento.

Por último, hace falta configurar el nodo e indicar cómo se les pasan las primitivas. Para ello se puede utilizar el archivo “move_base_rbcar.launch”. En él se encuentran las siguientes líneas:

```
<param name="base_global_planner" value="SBPLLatticePlanner" />
<rosparam file="config/sbpl/sbpl_global_params.yaml" command="load"/>
<param name="SBPLLatticePlanner/primitive_filename" value="config/sbpl/$(arg prim).mprim"/>
<param name="planner_frequency" value="0.0"/>
```

El primer parámetro sirve para indicarle al *move_base* donde se configura los planificadores en ROS. En este case se utilizará el *SBPLLatticePlanner*. A continuación se cargarán unos parámetros (éstos se comentarán posteriormente). Posteriormente se indican las primitivas que se han calculado con el script de Matlab. Por último, el parámetro *planner_frequency* con valor 0.0 nos indica que sólo se calculará ruta cuando le llegue, por lo que no se tendrá que calcular de forma periódica.

El fichero de configuración que se carga en el *launch* contiene los siguientes parámetros definidos dentro del fichero sbpl_global_params.yaml:

```
SBPLLatticePlanner:
    planner_type: ARAPlanner
    environment_type: XYThetaLattice
    allocated_time: 30.0
    initial_epsilon: 1.0
    forward_search: true
```

siendo el *planner_type* el tipo de planificador que se va a utilizar y fue descrito en el apartado teórico. *Allocated_time* es el tiempo disponible hasta encontrar una solución. Environment_type sirve para indicarle en qué tipo de entorno se va a calcular la ruta. En este caso, al ser un entorno que debe tener en cuenta X, Y, Orientación, se debe poner el siguiente valor como configuración "*XYThetaLattice*". Por último, cuando *forward_search* es *true* se le indica que empiece a buscar la solución desde el vehículo (y no desde el destino). Puede parecer que el sentido (vehículo a destino o destino a vehículo) es indiferente, pero esto no es así ya que reviste cierta importancia desde el punto de vista de optimizar el cálculo de rutas.

En la siguiente figura se puede apreciar representada en azul la ruta calculada por el planificador global:

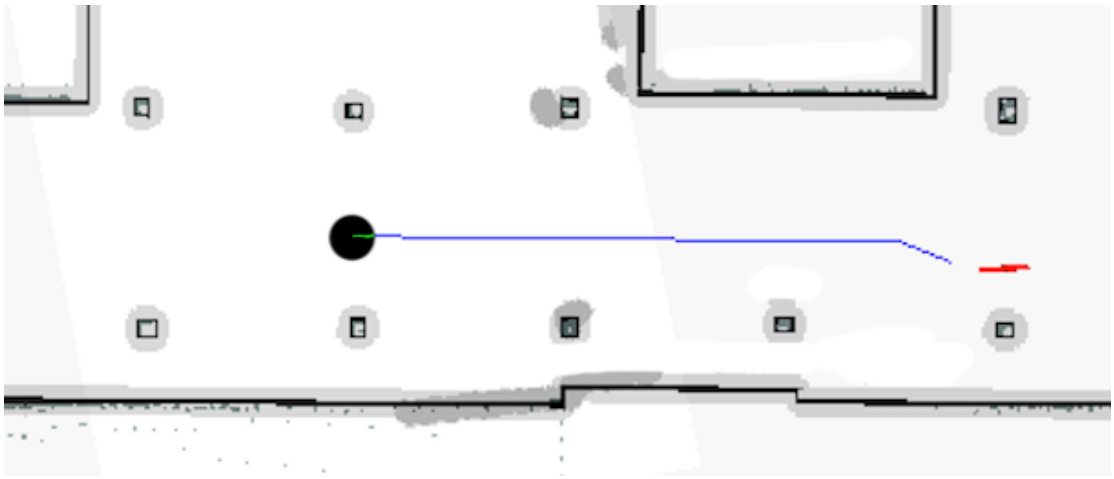

Figura 49. Ejemplo de ruta calculada por el SBPL.

3.3.4) Planificador local.

En este trabajo se ha utilizado como planificador local el paquete *The Elastic Band* (TEB), desarrollado por la universidad de Dortmund. Este paquete es el encargado de recibir una lista de posiciones con orientación y debe ser capaz de generar una trayectoria que el vehículo pueda seguir sin problemas basado en el comportamiento de una banda elástica.

Una de las ventajas que ofrece este planificador local es que dispone de un conjunto de obstáculos dinámicos, lo que permitirá mover dichos obstáculos y ver como esto afecta en la trayectoria generada.

En la figura siguiente se puede observar varias rutas (dibujadas en verde) generadas por el paquete, aunque la mejor es la que incluye las flechas en negro.

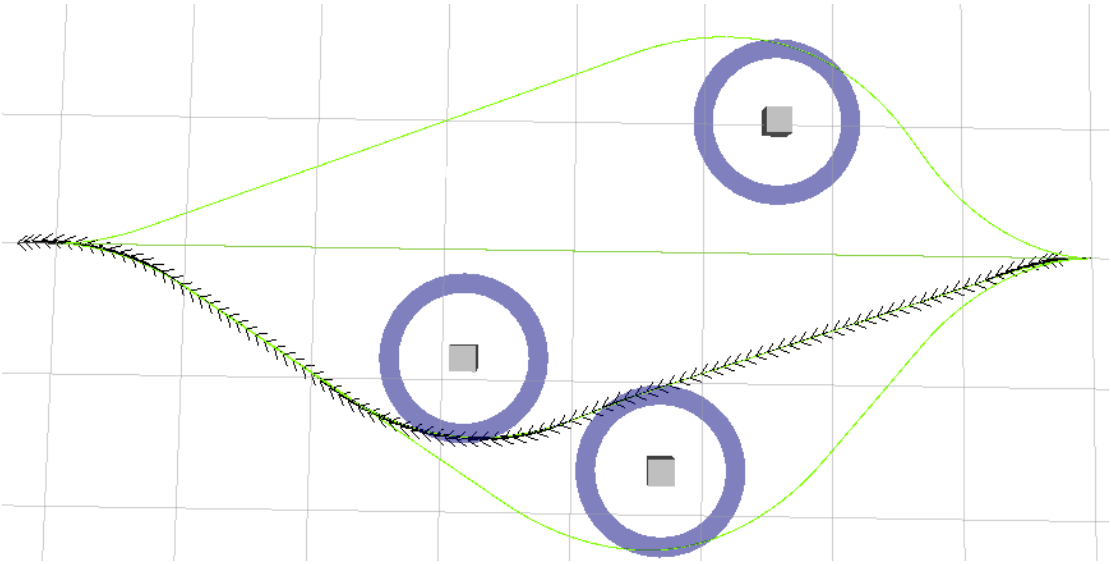

Figura 50. Representación diferentes trayectorias calculadas con TEB.

Una de las razones que motivó la utilización de este paquete fue su reducido tiempo de cálculo para calcular correctamente la ruta. Otra ventaja adicional del paquete TEB es la cantidad de parámetros que tiene y que permite adaptar el funcionamiento del nodo.

Para incluir el planificador TEB en la aplicación es necesario incluir en el fichero *move_base.launch* del vehículo las líneas siguientes:

```
<param name="base_local_planner" value="teb_local_planner/TebLocalPlannerROS" />
<rosparam file="config/teb/teb_local_planner_params.yaml" command="load"/>
<param name="controller_frequency" value="3.0"/>
```

El parámetro *base_local_planner* sirve para indicarle qué planificador local se desea utilizar en el proyecto. A continuación se cargan los valores de los parámetros del fichero yaml, y por último se indica la frecuencia del controlador expresada en hercios.

Los parámetros del fichero .yaml para el planificador son los siguientes:

```
TebLocalPlannerROS:
    global_plan_overwrite_orientation: True
    max_vel_x: 1.5
    min_turning_radius: 2.0
    wheelbase: 1.65
    xy_goal_tolerance: 0.3
    yaw_goal_tolerance: 0.4 #0.5
```

El parámetro *global_plan_overwrite_orientation* sirve para darle potestad al planificador local para sobreescribir la orientación obtenida desde el planificador global. Los otros parámetros más interesantes son: *max_vel_x* (la velocidad máxima de desplazamiento hacia delante, expresada en metros/segundos), el parámetro *min_turning_raduis*, que sirve para indicar el radio de giro del vehículo cuando el volante está girado en un extremo. *wheelbase* es la distancia entre las ruedas delantes y traseras (también conocido como batalla del vehículo). Los dos últimos valores son *xy_goal_tolerance* y *yaw_goal_tolerance*. El primero es la tolerancia al error en posición en la que se puede considerar que se ha llegado al

objetivo. El segundo parámetro lo que indica es la tolerancia a no cumplir con la orientación del destino.

En la siguiente figura se puede observar el funcionamiento del sistema planificador. El punto negro es la posición y orientación actual del robot siendo el destino la zona roja el destino de la misma, en azul esta la ruta calculada por el planificador global SBPL, y en verde se muestra esta ruta es la generada por el planificador local, proporcionando una ruta más suave. Ésta será la ruta que seguirá el vehículo.

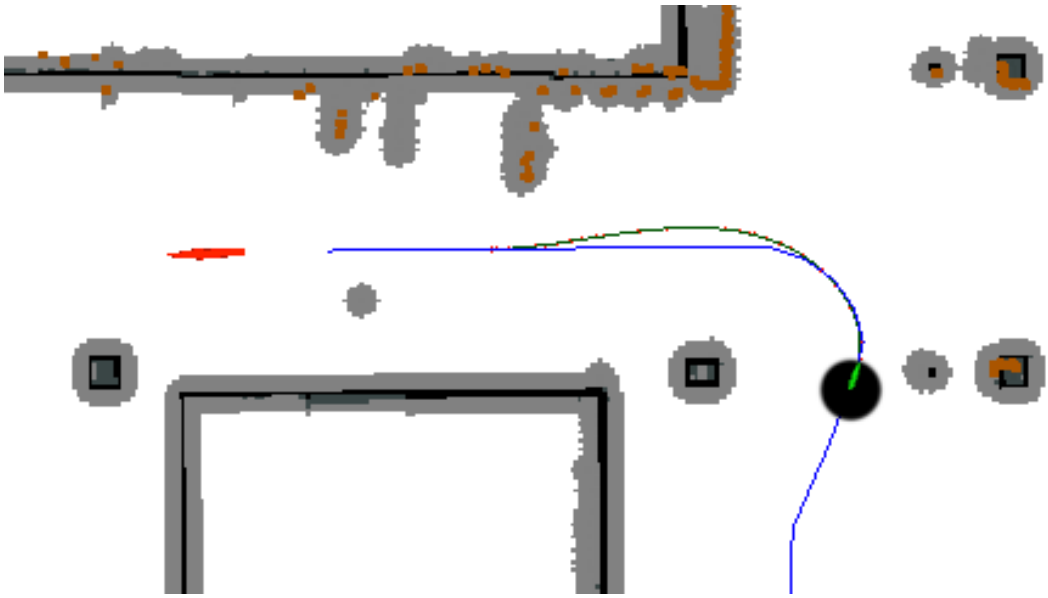

Figura 51. Corrección de trayectoria global por el planificador local.

3.4) Mapeado de Zonas

En paralelo al desarrollo de la aplicación de navegación en entornos conocidos (especialmente en entornos de interior) surgió la necesidad de encontrar un método para la generación de los mapas. Se encontraron varias soluciones, pero finalmente se optó por el paquete *slam_mapping*, que tiene diferentes nodos pero el que nos interesa es el nodo *gmapping*.

Para empezar a grabar el mapa solo hay que ejecutar el nodo lanzando el siguiente comando:

```
rosrun gmapping slam_gmapping scan:=scan odom_frame:=odom
```

En el anterior comando se encuentran varias partes. En el inicio se ve cómo se lanza el nodo en ROS. Después aparecen los parámetros de configuración. En "*scan*:=" se indica el nombre del *topic* del sensor laser que se va a utilizar, mientras que en "*odom_frame*:=" se indica el nombre de la transformada de *odom*.

En caso de utilizar el nodo *gmapping* se observa como a medida que el vehículo empieza a moverse se empieza a autocompletar el mapa. En las figuras siguientes se puede observar el proceso de autocompletando hasta completar el mapa de la zona deseada.

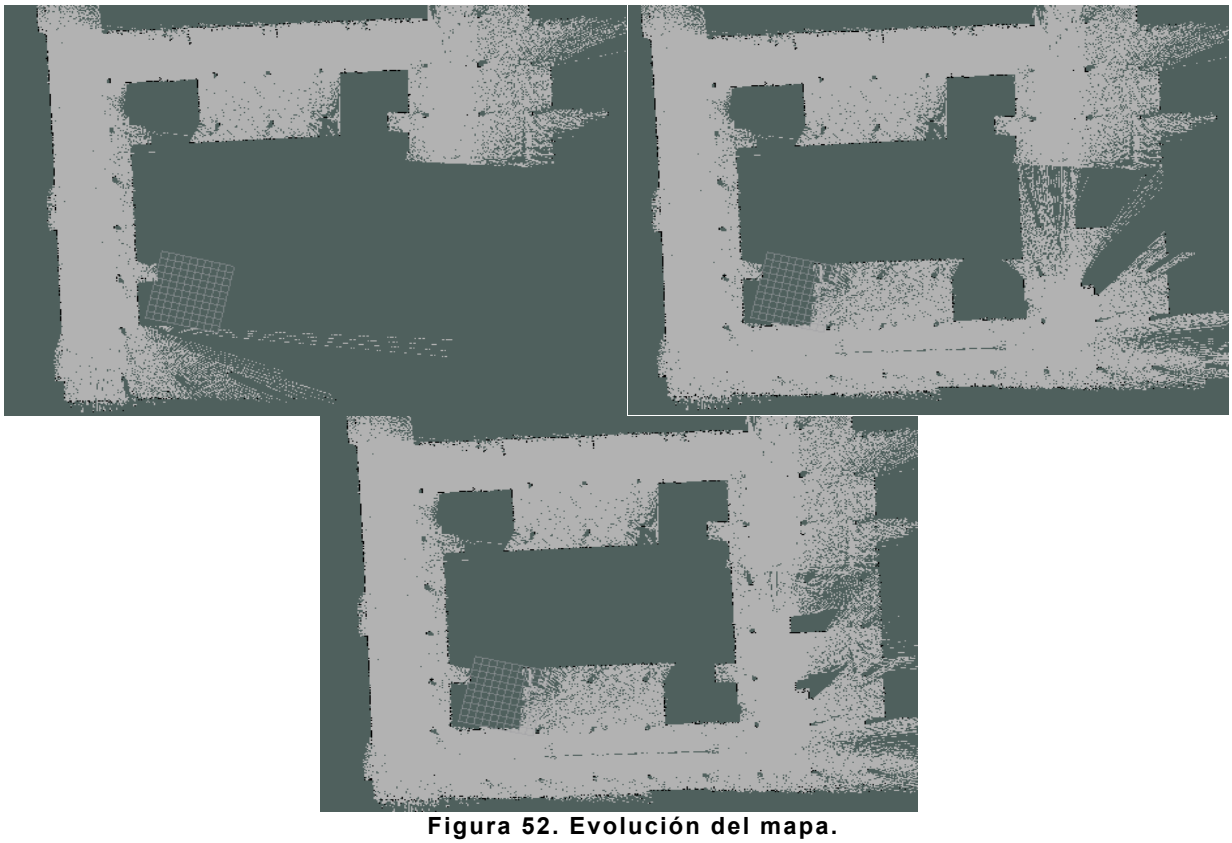

Figura 52. Evolución del mapa.

3.5) Aplicación waypoints.

Después del desarrollo del sistema de navegación autónoma, tanto en espacios de interior como en exteriores, se planteó disponer de una aplicación donde se pudiera indicar una lista de puntos de paso (*waypoints*), con el inicio y un destino. Bajo esta premisa realizamos la aplicación *waypoints*.

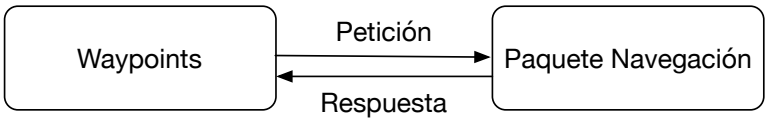

Figura 53: Idea principal aplicación.

Esta aplicación tiene como idea principal la de utilizar los diferentes paquetes desarrollados previamente. Para eso se debe conectar al servidor de acciones que incorpora el paquete desarrollado, devolviendo un valor "*succeeded*" en caso de haber alcanzado el *waypoint* que se le ha indicado previamente. En este caso la aplicación se encarga de pedir el siguiente *waypoint*, repitiendo el proceso hasta que se acabade la lista de *waypoints*.

En caso de llegar al final de lista se comprueba la variable *loop*. En caso de contener un valor *true* se vuelve a empezar todo el proceso desde el principio de la lista.

La figura siguiente muestra el mapa del parking del edificio 8G de la Ciudad Politécnica de la Innovación (CPI) de la Universitat Politècnica de València. Los símbolos en rojo indican los puntos de paso escogidos para la navegación del vehículo.

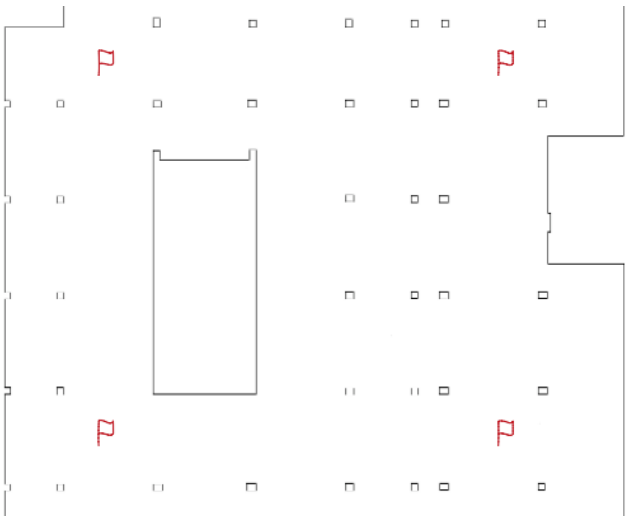

Figura 54. Mapa con puntos de paso.

La única exigencia que tiene este nodo es que los puntos estén expresados en metros, representando la distancia entre dicho punto y el origen del sistema de coordenadas del mapa que se utilice. La estructura del fichero de los diferentes puntos de paso (*waypoints*) debe ser la siguiente:

```
waypoints:
   - {
     name: 'n1',
     frame_id: '/map',
     position: [8.042, 49.166, -1.566 ]
     }
```

name el nombre simbólico del punto de paso, *frame_id* indica de qué *frame* está referenciado el punto. *position* [x, y, orientación] está expresado en metros y radianes respectivamente.

3.6) Navegación autónoma en conjunto.

Después de la descripción de las distintas aplicaciones desarrolladas para la navegación automática del vehículo eléctrico se van a presentar una serie de imágenes obtenidas de la aplicación final desarrollada, para situar al lector el mapa que se observa en las diferentes imágenes es el entresuelo del edificio CPI, para ser más exactos el 8B. La prueba a realizar va a ser completar una vuelta alrededor del cubo azul, mediante la utilización de waypoints.

Para ejecutar la navegación autónoma debemos lanzar los siguientes comandos.

- *Roslaunch rbcar_bringup all_components.launch* (con este comando lanzamos todos los nodos de sensores y configuración del vehículo Rbcar).
- *Roslaunch rbcar_2dnav rbcar_2dnav_real_teb*.launch (con este comando lanzamos todos los nodos de navegación, localización, planificadores).
- *Roslaunch rbcar_wpts rbcar_wpts.launch* (con este comando lanzamos un nodo que controla los puntos de paso).
- *Rosrun rviz rviz* (con este comando lanzamos un visualizador)

Una vez tenemos los nodos anteriores lanzados, en el programa rviz aparece lo presentado en la figura siguiente. En ésta se puede observar la posición actual del vehículo (el punto negro), siendo la zona roja alrededor de dicho punto las posibles posiciones estimadas que devuelve el localizador local. La línea naranja simboliza la ruta recalculada por el planificador local.

En la figura se puede apreciar también como las lecturas obtenidas por el láser no se corresponden con la posición exacta en el mapa.

Figura 55. Navegación en conjunto primera imagen.

En la siguiente imagen se puede observar como el localizador AMCL ha recalculado la posición de forma que corrige la divergencia que se daba en la figura anterior.

Se puede observar también la ruta de referencia (en color azul) calculada por el planificador global.

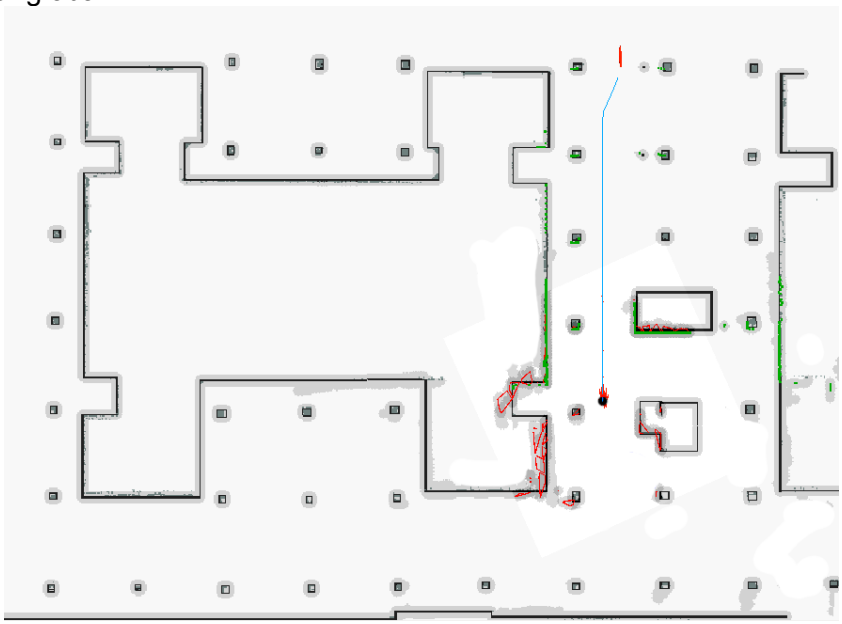

Figura 56. Navegación en conjunto segunda imagen.

En la figura siguiente se sigue mostrando la evolución de la navegación automática. En este caso se puede observar como en el giro a izquierdas del vehículo el sistema realiza una modificación en la ruta a seguir, siendo ésta una modificación muy pequeña respecto de la ruta inicial.

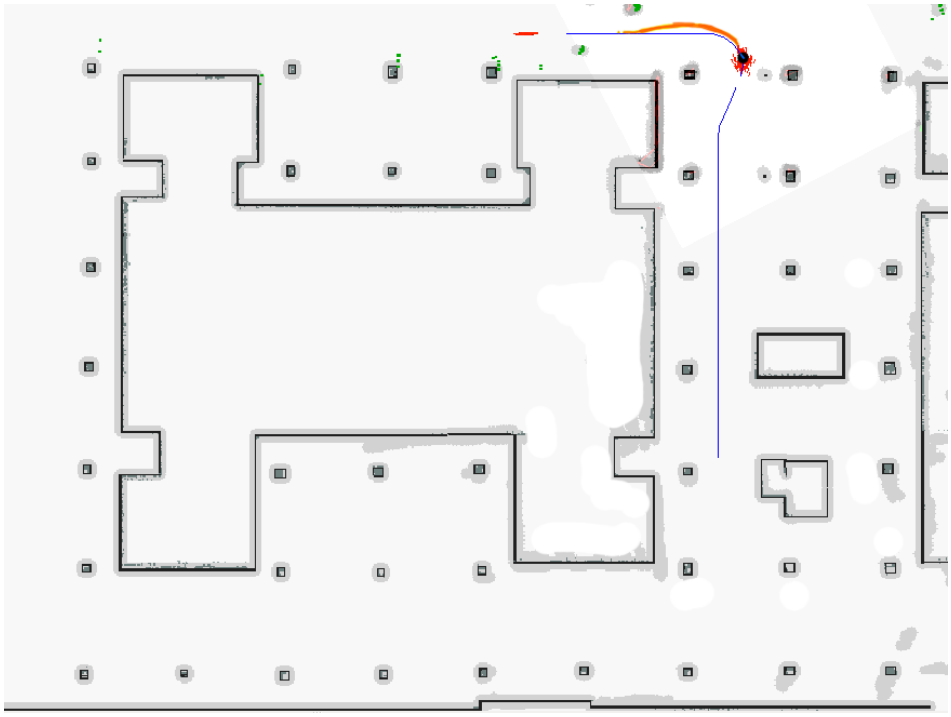

Figura 57. Navegación en conjunto tercera imagen.

En la figura siguiente se puede apreciar como para tomar la última curva se realiza un cálculo nuevo de la trayectoria. Esto estuvo motivado porque el vehículo se había perdido un poco en ese punto. No obstante, después de realizar la curva el localizador actuó de nuevo, reposicionando nuevamente el vehículo.

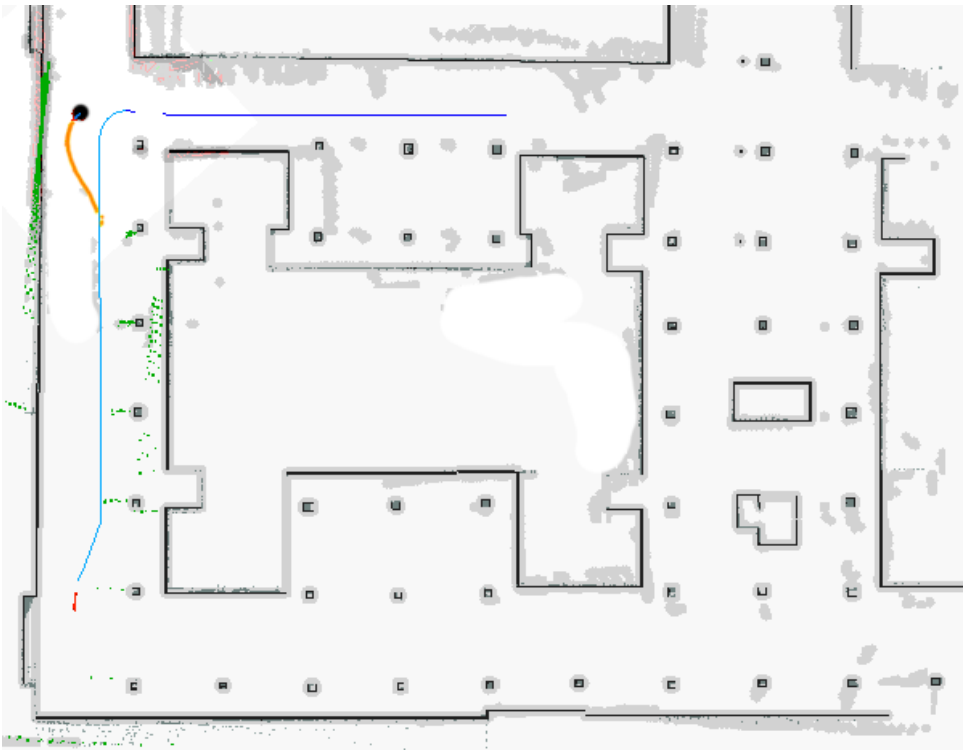

Figura 58. Navegación en conjunto cuarta imagen.

3.7) Protocolo desarrollo vehículos autónomos.

Como último apartado, debido a la experiencia que se ha obtenido con la realización de este trabajo, pareció oportuno realizar un pequeño protocolo que indicara cuáles deberían ser los pasos para el desarrollo de un vehículo autónomo.

Este procedimiento para realizar la automatización de vehículos consta de las siguientes etapas:

- Elegir vehículo como plataforma, determinando las características mínimas que debería tener el vehículo. Se trata de características como la velocidad máxima, la aceleración, el peso, el ángulo de giro máximo, la batalla del vehículo, etc.

- Seleccionar los sensores que se van a colocar en el vehículo. En este sentido debemos destacar que, por ejemplo, cuanta mayor sea la velocidad máxima del vehículo, los sensores láser de rango que se incluyan deben tener un mayor alcance.

- Calibrar el sistema encargado de controlar el volante, así como la de los sensores que van sobre el mismo. Este es un punto muy importante puesto que la precisión del vehículo dependerá de ello.

- Elegir qué distribución y qué componentes se van a utilizar en el vehículo, pensando en cómo realizar la navegación si basada en cámaras, láseres,...

- Después de obtener la distribución elegida para el vehículo se debe verificar el funcionamiento correcto de los diferentes sensores.

- Desarrollar el sistema de localización. Esto es un punto sumamente importante puesto que se de no ser así se inducirán a errores en la planificación

- Posteriormente se debe desarrollar el planificador local. Para ello se deben realizar las diferentes pruebas y ver cómo responde el vehículo en la zona que es capaz de detectar mediante los sensores.

- A continuación se debe desarrollar el planificador global. La planificación global es como planificar en zonas donde no disponemos de información o la información esta desactualizada.

- Integrar el planificador local para que calcule la ruta respecto a lo que se planifica globalmente.

- Realizar un test de seguridad del sistema, como por ejemplo la detección de colisiones para ver si el sistema es capaz de responder ante imprevistos.

4) Conclusiones y futuras aplicaciones.

Viendo la actual revolución existente en este campo. La verdad que intimida poder decir que se puede llegar a dominar por completo. Durante la realización de este proyecto me ha servido principalmente para empezar a entender como algo que los humanos hacemos cosas con cierta facilidad, por ejemplo el conducir desde un origen aun destino. Nos supone realmente un gran reto, no por la dificultad de los desarrollos en sí mismo. Sino por su complejidad al abarcar una amplia cantidad de áreas diferentes.

Pero centrándonos en los diferentes objetivos marcados al inicio del proyecto se pueden dar por conseguidos los siguientes:

- Analizar las ventajas e inconvenientes y limitaciones del modelo cinemático de Ackermann.
- Seleccionar un entorno de desarrollo y el lenguaje de programación para la realización del trabajo.
- Estudiar y adaptar los sensores disponibles en el vehículo eléctrico que pueden ser utilizados en el entorno de programación seleccionado.
- Realizar la fusión de varios sensores que permita aumentar la precisión de las medidas obtenidas.
- Estudio y desarrollo de un planificador de rutas que permitan al vehículo realizar la navegación autónoma.
- Cálculo, en un tiempo razonable, de la ruta a seguir.
- Analizar y comprobar el correcto funcionamiento del sistema.

Aunque alguno de los objetivos son relativos, como puede ser la elección de los sensores en nuestro caso ha sido basándonos en los que disponíamos en el laboratorio y otro de los puntos que se han considerado como relativos es el cálculo en coste de tiempo a seguir.

En caso de intentar realizar un proyecto similar a este, en el mundo empresarial. Nos podemos encontrar realmente frente a un reto, que bajo mi punto de vista actual, debería ser desarrollado por un equipo multidisciplinar. Donde sin ninguna duda necesitaríamos un psicólogo, puesto que para entender y como poder hacer más amigables estos sistemas. Durante el desarrollo del sistema ha sido un gran número de personas diferentes las que se han subido al vehículo y aunque hemos limitado la velocidad a 6 kilómetros por hora, encontramos reacciones muy llamativas, que no sabría definir si se encuentran entre pánico y otras emociones relacionadas con el estrés. En nuestro caso las pruebas fueron realizadas donde no había tráfico rodado o situaciones de estrés, por lo tanto solo habían factores entre el proyecto y la reacción humana a estos sistemas.

Pero volviendo al lado más técnico que es el que se ha buscado abordar con este proyecto, podemos afirmar que este tipo de proyectos, sobre todo donde el I+D y los equipos son reducidos. Nos supondrá un reto para el desarrollo técnico donde contar con una gran motivación y una gran tolerancia al fracaso, serán factores necesarios para la consecución del mismo. Esto último se debe a la gran cantidad de pruebas que se realizaron con resultados no satisfactorios, donde observamos como la calibración del volante, nos costó cerca de un mes y medio.

En cuanto al enfoque que deberían dar las empresas en este sector creo que deberían buscar la forma de integrar correctamente imágenes de dos cámaras estéreo, sensores laser y sin duda una gran dependencia del GPS. Siendo de vital importancia poder determinar límites de funcionamiento de los sistemas. Así como la necesidad de agilizar todo lo que en el proceso de planificadores se refiere, actualmente o disponemos de ordenadores muy potentes incluso algunos rack de servidores para conseguir mover un vehículo, sin duda creo que el poder minimizar estos cálculos de movimiento nos será muy útil, de cara a un futuro.

Pero más de esto lo que sí que creo que será la revolución es el campo que nos abre este tipo de vehículos donde la comunicación entre varios vehículos o incluso la toma de decisiones conjunta será un el gran salto siguiente a dar, aunque aquí se necesitará más que voluntad por parte de las diferentes empresas fabricantes de vehículos. Por otro lado veo aparte de la comunicación entre los diferentes vehículos autónomos posiblemente otra de las cosas a tener sea la comunicación con las diferentes señales de tráfico, siendo un punto de partida interesante el proyecto “lights” del MIT interesante de cara a futuros desarrollos.

De este proyecto espero que sea capaz de ser un punto de partida hacia otros proyectos similares.

5) Bibliografía.

- Darpa Grand Challenge. https://en.wikipedia.org/wiki/DARPA_Grand_Challenge
- Tartán http://www.tartanracing.org
- Junior http://cs.stanford.edu/group/roadrunner/
- Odín http://www.mechatronic.me.vt.edu/About/aboutPage.html
- REP 103 http://www.ros.org/reps/rep-0103.html
- REP 105 http://www.ros.org/reps/rep-0105.html
- Sick lms 291-s05 http://sicktoolbox.sourceforge.net/docs/sick-lms-technical-description.pdf
- Hokuyo urg04-lx http://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG-04LX_UG01_spec_en.pdf
- Autonomous Driving in Urban Environments: Boss and the Urban Challenge http://www.fieldrobotics.org/users/alonzo/pubs/.../JFR_08_Boss.pdf
- Darpa Urban Challenge Technical Paper team: VictorTango http://archive.darpa.mil/grandchallenge/TechPapers/Victor_Tango.pdf
- Junior: The Stanford Entry in the Urban Challenge
- robots.stanford.edu/papers/junior08.pdf
- A Perception-Driven Autonomous Urban Vehicle https://april.eecs.umich.edu/pdfs/mitduc2008.pdf
- http://wiki.ros.org/
- http://wiki.ros.org/navigation
- Algoritmo A*: ARA*: Anytime A* with Provable Bounds on Sub-Optimality
- http://papers.nips.cc/paper/2382-ara-anytime-a-with-provable-bounds-on-sub-optimality.pdf
- Definición mensaje Ackermann_msgs - http://docs.ros.org/jade/api/ackermann_msgs/html/msg/AckermannDriveStamped.html
- Paquete navegación estándar ROS - http://wiki.ros.org/navigation/Tutorials/RobotSetup
- Esquema navegación autónoma ROS - http://wiki.ros.org/navigation/Tutorials/RobotSetup?action=AttachFile&do=view&target=overview_tf.png
- Robot Localization - http://wiki.ros.org/robot_localization
- UTM - https://es.wikipedia.org/wiki/Sistema_de_coordenadas_universal_transversal_de_Mercator
- LGPL - https://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License

Artículos:

- Planning Of Multiple Robot Trajectories In Distinctive Topologies. Cristoph Rösmann, Frank Hoffma, Torsten Bertram de la tu-dortmund.
- Collision Avoidance Using Elastic Band For an Autonomous Car. Stefan K. Gehirg, Fridtjof J. Stein Research Institute DaimlerChrysler AG Stuttgart Germany.

Libros:

- A Gentle Introduction to ROS - Jason M O'Kane.
- Learning ROS for Robotics Programming-Aaron Martinez, Enrique Fernández
- Probabilistic Robotics, Sebastian THRUN, Stanford University, CA.