UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIEROS INDUSTRIALES VALENCIA

TRABAJO FIN DE GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

DISEÑO, MONITORIZACIÓN Y CONTROL DE UN HEXÁPODO CON ROS

AUTOR: CARLES IGUAL BAÑÓ

TUTOR: LEOPOLDO ARMESTO ÁNGEL

COTUTOR:

Curso Académico: 2013-14

ÍNDICE GENERAL

MEMORIA:
MEMORIA DESCRIPTIVA

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. RESUMEN

En menos de 30 años la robótica pasó de ser un mito, propio de la imaginación de algunos autores literarios, a una realidad imprescindible en el mercado productivo. Tras los primeros albores, tímidos y de incierto futuro, la robótica experimentó entre las décadas de los setenta y ochenta un notable auge, llegando en los noventa a lo que por muchos ha sido considerado su mayoría de edad, caracterizada por una estabilización de la demanda y una aceptación y reconocimiento pleno en la industria.

Poco a poco la robótica se ha ido introduciendo en nuestras vidas. Los robots han huido de las fábricas para invadir nuestros hogares, calles, oficinas. Desde principios del siglo XXI ha aparecido una nueva generación de robots con un aspecto amigable y de tamaño reducido, algo que hace que su presencia en nuestro día a día cada vez sea más natural. Su propósito va desde limpiar nuestras casas hasta hacernos pasar buenos ratos con su compañía. Hoy en día el robot se proyecta con la idea de sustituir al ser humano en la realización de cosas sencillas y rutinarias, permitiendo a las personas dedicarse a otras tareas más elaboradas.

Esta familiaridad con la que nuestra sociedad trata al robot es más llamativa cuando se compara con el amplio desconocimiento que se puede tener de otras máquinas o aparatos, aún siendo de mayor antigüedad o utilidad como por ejemplo sería el osciloscopio o los parientes cercanos de los robots: máquinas de control numérico. Posiblemente una de las causas principales que haya dado popularidad al robot sea su mitificación, propiciada o amplificada por la literatura y el cine de ciencia ficción.

En este trabajo se pretende diseñar y resolver el manejo de uno de estos pequeños robots que cada vez son más comunes en nuestro día a día y que llegan a ser los nuevos juguetes de una generación de niños que los ven como algo igual de común que un balón. Para ello, se diseñará el robot y posteriormente se resolverá su movimiento, bien mediante una simulación o bien mediante un cálculo matemático, para posteriormente tratar y enviar esos datos al robot, intentando que este se mueva de la manera deseada. Esto se puede diseñar de múltiples maneras, pero en este caso se ha elegido una herramienta denominada ROS, un sistema operativo para el control de robots, que plantea una integración de todo lo necesario para llevar a cabo el proyecto, que nunca anteriormente se había visto, y que ofrece un marco de posibilidades muy amplio que empujará a la robótica a otro nivel, uno mucho más cercano a la gente y no solo al alcance de algunos. En el marco de ROS se realizará el proceso utilizando dos herramientas diferentes de las que ofrece, para determinar cual es la forma más sencilla y las ventajas de una con respecto a la otra.

2. INTRODUCCIÓN

2.1. Antecedentes Históricos

Desde hace siglos el ser humano ha intentado construir máquinas que imitaran las partes del cuerpo humano. Ya los antiguos Egipcios unían brazos mecánicos a las estatuas de sus dioses, que luego eran movidos por sacerdotes los cuales aseguraban actuar en nombre de los dioses. Los Griegos por su parte, construyeron estatuas que podían moverse hidráulicamente.

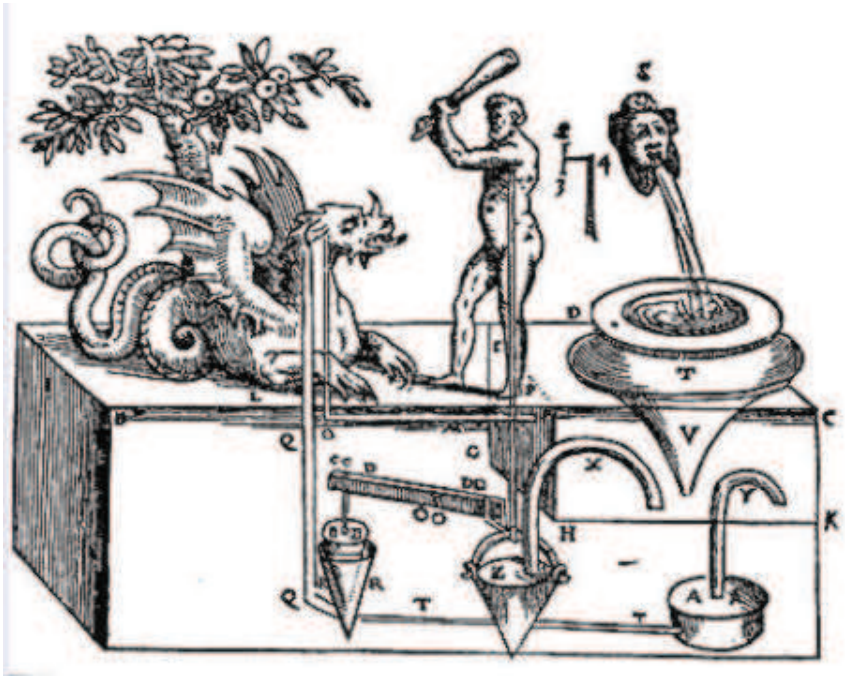

Figura 1. Estatua griega con mecanismo hidráulico.

En el siglo I, Herón de Alejandría diseñó una serie de dispositivos que actuaban en función de la acción del agua, de chorros de vapor y de equilibrio de pesos. En el siglo VI los bizantinos crearon un reloj operado por agua para una estatua de Hércules.

La cultura árabe (siglos VIII a XV) heredó y difundió los conocimientos griegos, utilizándolos no sólo para realizar mecanismos destinados a la diversión, sino que les dio una aplicación práctica, introduciéndolos en al vida cotidiana de la realeza. Ejemplo de éstos son diversos sistemas dispensadores automáticos de agua para beber o lavarse.

Figura 2. Sistema dispensador de agua árabe.

Durante los siglos XV y XVI alguno de los más relevantes representantes del renacimiento se interesan también por los ingenios descritos y desarrollados por los griegos. Es conocido el *León mecánico* construido por Leonardo Da Vinci.

Figura 3. El León mecánico de Leonardo Da Vinci.

Posteriormente, en los siglos XVII y XVIII aparecen autómatas (máquina que imita la figura y movimientos de un ser animado) y mascotas mecánicas, construidos en Europa con la funcionalidad de entretener y divertir a la gente. Estos dispositivos fueron creados en su mayoría por artesanos del gremio de la relojería. Cada autómata estaba mecánicamente programado para realizar una tarea, pero no se podía ejecutar otra diferente sin construirlo completamente de nuevo. Estos robots no interaccionaban con el mundo exterior.

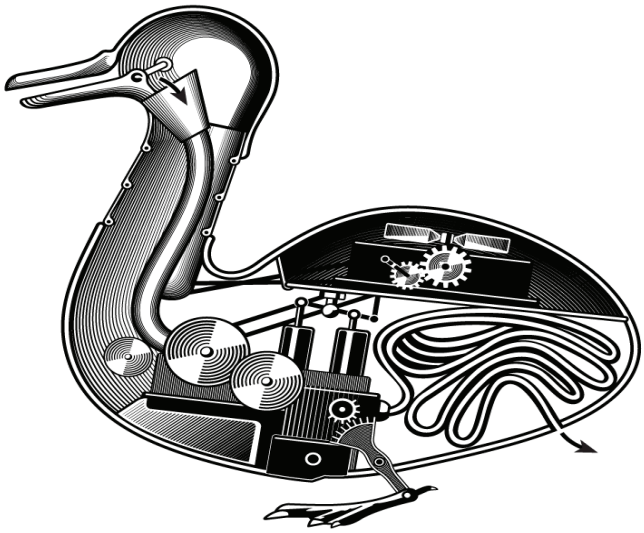

Figura 4. Mascota mecánica con forma de pato.

La tecnología avanzó en tal medida que a mediados del siglo XVIII, Jacques Vaucanson construyó una especie de humanoide con labios de goma que se movían de manera que controlaban el flujo de aire, y de este modo eran capaces de emitir notas musicales con el uso de una flauta. Colocando los dedos de forma adecuada sobre los agujeros de la flauta, el humanoide podía tocar un repertorio de doce notas musicales.

Hasta los comienzos de la automatización industrial, hacia finales de siglo XVIII, los autómatas tenían este carácter lúdico o meramente doméstico, en ningún caso hasta la fecha se había buscado la creación con fines productivos. Pero es a partir de este momento cuando se empiezan a utilizar dispositivos automáticos en la producción, ya no se busca la imitación física del ser humano, sino más bien facilitar y sustituir al trabajador en labores repetitivas. A mediados del siglo XX, surge la automatización flexible, en la que se persigue obtener la "máquina universal" como contraposición a la máquina especial.

Muchos factores favorecieron al avance de los robots en la segunda mitad del siglo XX. El desarrollo de la electrónica, asociado a los avances de otras técnicas como la mecánica, la hidráulica, la neumática y la electricidad, da origen a las primeras máquinas-herramienta de control numérico lo que hará más flexibles a las máquinas desde el punto de vista de la reconfigurabilidad mediante sus programas. Las investigaciones en inteligencia artificial desarrollaron medios para simular el procesamiento de la información realizado por los humanos mediante ordenadores y una gran variedad de mecanismos electrónicos.Los ingenieros más creativos se vieron atraídos por la posibilidad de construir máquinas programables e imitar el comportamiento humano.

La robótica siempre estuvo muy ligada a las películas y libros en los que se hablaba de ella. La palabra robot fue usada por primera vez enel año 1921, cuando el escritor checo Karel Capek estrena en el teatro nacional de Praga su obra *Rossum's Universal Robot*. Su origen es la palabra eslava *robota*, que se refiere al trabajo realizado de manera forzada.

Figura 5. Fotograma de "Metrópolis" (1927).

Otro gran hito tuvo lugar en 1950 con la publicación de *I Robot*, Isaac Asimov. En este se trata la inteligencia de los humanoides y se redactan tres famosas leyes:

- Un robot no puede hacer daño a un ser humano, o por inacción, permitir que un ser humano sufra daño.
- Un robot debe obedecer las órdenes dadas por los seres humanos, excepto si estas órdenes entrasen en conflicto con la 1ª Ley.
- Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la 1ª o la 2ª Ley.

Aunque haya una gran relación, el robot como máquina lleva un desarrollo independiente del término robot. Los telemanipuladores (los progenitores más directos de los robots) desarrollados a finales de los cuarenta para el manejo de materiales radiactivos fueron determinantes para el posterior desarrollo de los primeros robots industriales. Estos sistemas de telemanipulación eran de tipo maestro-esclavo, con un brazo manipulador movido directamente por el operario en la zona segura, y el esclavo, que, acoplado mecánicamente al maestro, reproducía los movimientos en la zona radiactiva. Uno de los pioneros en la construcción de estos sistemas fue el ingeniero R. C. Goertz, de Argonne National Laboratories, quien en 1948 desarrolló un manipulador maestro-esclavo de tipo mecánico, el cual posteriormente en 1954 se convirtió en el primer telemanipulador con servocontrol bilateral, sustituyendo la transmisión mecánica por otra eléctrica.

Otro de los pioneros de la telemanipulación fue Ralph Mosher, ingeniero de la General Electric que en 1958 desarrolló un dispositivo denominado Handy-Man, consistente en dos brazos mecánicos teleoperados mediante un maestro del tipo denominado exoesqueleto.

La idea de dotar de autonomía a brazos manipuladores (como los de Goertz) dio paso al concepto de robot. La primera patente de un dispositivo robótico fue solicitada en marzo de 1954 por el inventor británico C. W. Kenward. Sin embargo, fue George C. Devol, ingeniero norteamericano, inventor y autor de varias patentes, el que estableció las bases del robot industrial moderno. En 1954 Devol concibió la idea de un dispositivo de *transferencia de artículos* programada que patentó en Estados Unidos en 1961. Se trataba de un manipulador cuya tarea podía ser ejecutada siguiendo una secuencia de pasos de movimiento previamente registrados en un programa. Por otra parte el primer robot móvil que se construyó fue el ELSIE en 1953, cuya función era de seguimiento de un foco luminoso.

En 1956 se pone esta idea en conocimiento de Joseph F. Engelberger, ávido lector de Asimov y director de ingeniería de la división aeroespacial de la empresa Manning Maxwell y Moore en Stanford, Conneticut. Juntos, Devol y Engelberger comienzan a trabajar en la utilización industrial de sus máquinas, fundando la Consolidated Controls Corporation, que más tarde se convierte en Unimation (Universal Automation), e instalando su primera máquina Unimate (1960) en la fábrica de General Motors de Trenton, Nueva Jersey, en una aplicación de fundición por inyección.

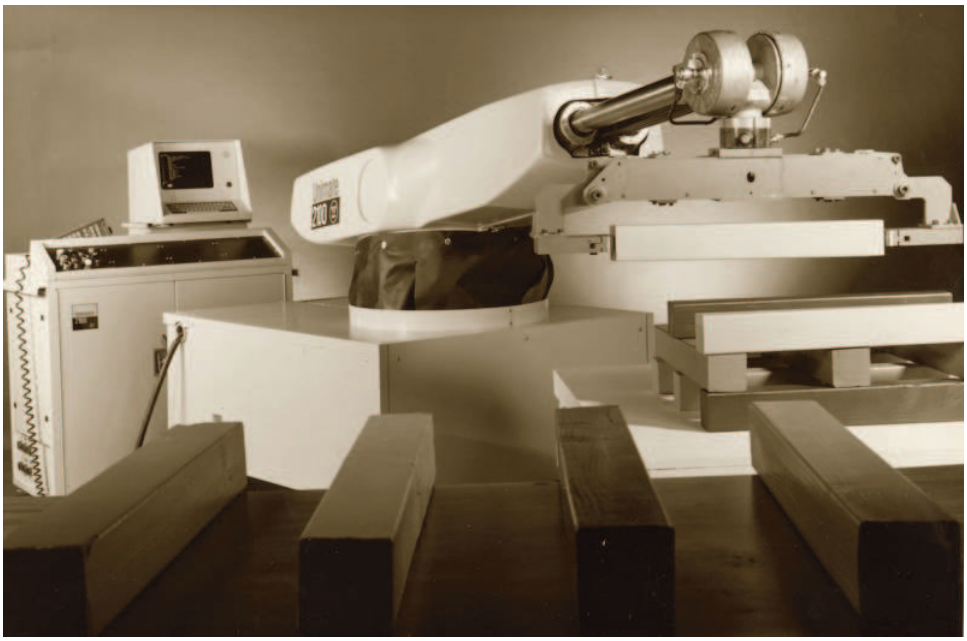

Figura 6. Primeros robots Unimate.

En 1968 aparece el SHACKEY del *Standford Research Institute*, el cual ya era un robot móvil mucho más equipado con múltiples sensores y una cámara de visión para desplazarse por el suelo. Este tipo de robots han sido muy utilizados posteriormente por la NASA en sus exploraciones en terrenos hostiles como las superficies de otros planetas.

El crecimiento de la robótica en Japón aventajó en breve a los Estados Unidos gracias a Nissan, que formó la primera asociación robótica del mundo, la *Asociación de Robótica Industrial de Japón* (JIRA) en 1972. Por su parte Europa tuvo un despertar más tardío. En 1973 la firma sueca ASEA construyó el primer robot con accionamiento totalmente eléctrico, el robot IRb6, seguido un año más tarde del IRb60. En 1980 se fundó la *Federación Internacional de Robótica* con sede en Estocolmo, Suecia.

El desarrollo de la microelectrónica, los avances en la tecnología de los ordenadores, la disponibilidad de servomecanismos electromecánicos e hidromecánicos fiables y el desarrollo de la teoría de control y de las máquinas de control numérico, entre otros, han sido factores claves en la evolución de la robótica. Gracias a estos avances, los robots de principios de 1980 resultaron comparativamente más fiables, más precisos, incluso más flexibles, debido a las nuevas capacidades de programación.

El robot industrial que se conoce y utiliza en nuestros días nace con el objetivo de aumentar la productividad, reducir los costes, mejorar la calidad de las piezas fabricadas y evitar la realización de trabajos tediosos o peligrosos para el hombre. El número de robots industriales existentes actualmente en el mundo supera con creces el millón de unidades. La evolución de los robots industriales desde sus primeros balbuceos ha sido vertiginosa. En poco más de 30 años las investigaciones y desarrollos sobre robótica industrial han permitido que los robots tomen posiciones en casi todas las áreas productivas y tipos de industria. En pequeñas o grandes fábricas, los robots pueden sustituir al hombre en aquellas tareas repetitivas y hostiles, adaptándose inmediatamente a los cambios de producción solicitados por la demanda variable. Los avances en

herramientas CAD (Computer-Aided Design) influyeron decisivamente en el desarrollo de herramientas de simulación en la robótica. A través de estas simulaciones se pretendía eliminar al máximo no sólo el tiempo de diseño previo a la instalación, sino también de puesta a punto de las instalaciones, llegando a eliminar la necesidad de programar el robot en la línea (a esto se le llama programación *off-line*).

Un robot industrial se suele caracterizar por sus grandes dimensiones y su posicionamiento fijo. Se distingue en él un manipulador o brazo mecánico al que se acoplan las diversas herramientas según la tarea, y es controlado por un sistema basado en un procesador. Los más sofisticados poseen sensores para detectar parámetros del entorno y alterar su funcionamiento en función de la información recibida, aunque éste no es un factor determinante. Trabajos que hasta hace poco era inconcebible que los realizase una máquina, hoy los efectúa un robot con rapidez y precisión. La industria que más robots aglutina es la del automóvil.

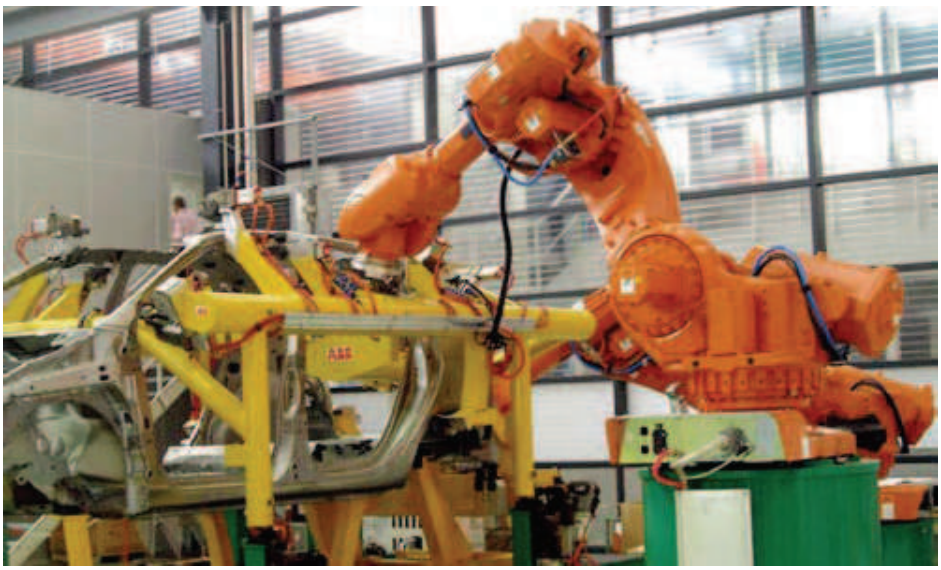

Figura 7. Robot en una planta industrial automovilística.

Paralelamente a la robótica industrial hemos visto la evolución de la robótica móvil (en la cual se centra este trabajo). En comparación con los grandes robots industriales, los robots móviles, parecen un simple juguete. Pueden tener forma de vehículo, de animal, de criatura robótica… pero todos tienen una característica común: la inteligencia necesaria para realizar tareas en un entorno que puede variar y al cual se pueden adaptar.

Estos robots son pequeños robots móviles y programables que realizan una sencilla tarea. Su cerebro consiste en un microcontrolador que gobierna todas sus acciones y según el trabajo encomendado precisa de un programa concreto, y de unas entradas de información y salidas para los actuadores. Su tamaño, potencia y precio es considerablemente menor al de un robot industrial, pero sobre todo difiere de éste en el tipo de tareas a las que se dedica.

Los robots móviles no intentar quitar el puesto a los robots industriales. No están diseñados para llevar a cabo operaciones que exijan elevadas potencias y complicados algoritmos. Estos robots están destinados a resolver tareas pequeñas con rapidez y precisión. Limpian, detectan o buscan elementos concretos, miden y toman el valor de magnitudes, transportan pequeñas piezas, vigilan y

supervisan, guían a invidentes, ayudan a minusválidos y hasta son magníficos compañeros de juegos.La movilidad es un factor clave en la filosofía de este tipo de robots para posibilitar la realización de muchas actividades.

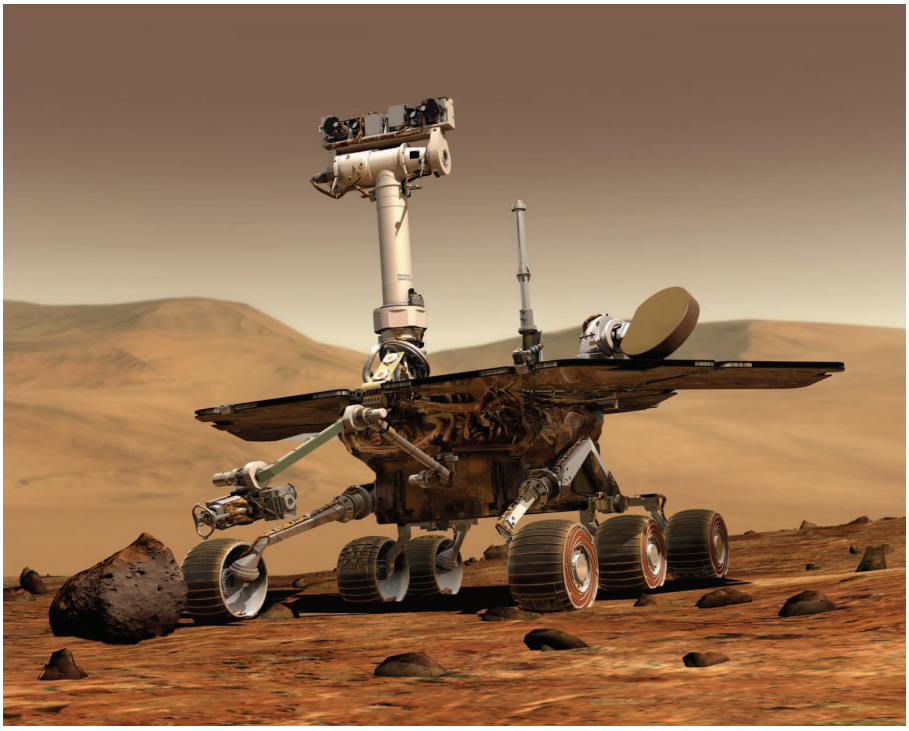

Figura 8. Robot móvil de exploración.

La robótica ha evolucionado de una manera exponencial, y parece que con el avance de la tecnología este desarrollo no va ha cesar en los próximos años. Metas anteriormente inalcanzables se han hecho realidad gracias a esta industria. La robótica será un pilar fundamental de nuestro futuro tecnológico más cercano y su evolución permitirá al hombre un nuevo abanico de posibilidades. El futuro de esta ciencia parece no tener fin.

Figura 9. Avance histórico en el robot ASIMO de Honda.

2.2. Motivación

El interés por esta materia y el objetivo de especializarme en automática han llevado a que realice este trabajo. El proyecto se centra en la rama de la robótica más atrayente para mi persona, la robótica móvil. La complejidad del mismo y el hecho de ser completamente distinto a lo que se ha dado a lo largo del grado aumentan el interés por realizar el trabajo y llegar así más preparado a futuros retos como el Máster en Ingeniería Industrial y otros proyectos. Además el desarrollo que está viviendo la robótica hace que cualquier adquisición de conocimientos respecto a esta temática pueda ser realmente útil.

2.3. Justificación

Como se ha comentado anteriormente, la aplicación de la robótica (concretamente de la robótica móvil) en la actualidad, está cada vez más en auge. Sin embargo, todos estos robots destinados a realizar tareas sencillas deben de ser diseñados, controlados y monitorizados de alguna manera, por ello, en este trabajo se busca una manera de poder realizar todas estas tareas sobre uno de estos robots, un hexápodo.

La principal cualidad de un robot móvil, como su propio nombre indica, es la movilidad de la que se le dota. Por lo tanto, el interés tecnológico en hacer que este pequeño hexápodo se mueva, intentando conseguir que la secuencia de movimientos genere una rutina de avance, es más que justificado ya que sin la movilidad, la funcionalidad del robot es prácticamente nula. Además, la amplia expansión de la robótica en la industria genera la necesidad de profesionales cualificados en esta temática, tanto para el uso del equipo ya instalado como para la fabricación y diseño de nuevos robots. Desde un punto de vista académico, este trabajo permite al alumno ampliar los conocimientos en electrónica, informática (sobretodo la rama de programación) y mecánica, además de aplicar lo obtenido durante su formación.

Concretamente, el robot motivo de estudio en este trabajo es útil en múltiples actividades, desde un nivel lúdico hasta actividades de exploración, como puede ser el reconocimiento de la superficie de otro planeta (salvando las distancias entre el robot disponible para el trabajo y los utilizados por la NASA para dichas tareas). Sin embargo, aunque el proyecto esté enfocado en un robot concreto, la primera parte del mismo dota al lector de los conocimientos necesarios para aplicar la herramienta utilizada (ROS) sobre cualquier modelo de robot, permitiéndole su diseño, monitorización y control y por lo tanto amplía el alcance del trabajo a un sin fin de posibilidades, tantos como robots se deseen utilizar.

3. OBJETIVOS

El objetivo general del proyecto se centra en conseguir que un hexápodo, del tipo *Helium Frog HF08 Hexapod Robot*, camine utilizando la herramienta ROS . Dentro de esta línea de trabajo se definen objetivos secundarios más concretos que irán guiando el trabajo a medida que éste avance:

- Estudio y dominio de la principal herramienta a utilizar en el proyecto, el sistema operativo ROS, para ello se seguirá en primer lugar unos tutoriales planteados en la propia web de la empresa, y posteriormente se practicará con ejemplos sencillos.
- Obtención de un modelo virtual del hexápodo que sea lo más fiel posible a la realidad para que en su posterior simulación los datos obtenidos sean correctos y el movimiento sea similar.
- Generación de la simulación mediante MoveIt! a partir del modelo creado anteriormente y que permita manipular el robot en su entorno y planificar el desarrollo del movimiento de un punto inicial a uno final.
- Definición de un paquete donde podamos controlar el movimiento de la simulación mediante el envío de mensajes a MoveIt! indicándole la posición final en cartesianas, que deseamos para los extremos de las patas.
- Implementación de un ejecutable(*listener*) que reciba el resultado de la simulación y envíe por USB a la tarjeta de adquisición de datos del robot las instrucciones necesarias para que éste adopte las posiciones de la solución.
- Estudio de la cinemática inversa que resuelve el movimiento del robot para posteriormente sustituir el simulador por un ejecutable que realice las operaciones y obtenga él mismo la solución y la envíe por USB a la tarjeta del robot.
- Generación de un paquete que permita interactuar al usuario con el robot permitiéndole elegir el paso y la altura que quiere que tengan las patas a la hora de caminar.
- Comparación entre el proceso utilizando el simulador MoveIt! con la resolución manual. Ventajas y desventajas de uno frente al otro.

4. EL HEXÁPODO

4.1. Construcción Material

Para el desarrollo del presente proyecto el principal e indispensable elemento es el robot. En este caso se ha elegido un hexápodo, concretamente el *Helium Frog HF08 Hexapod Robot*[7], un hexápodo imprimible de bajo coste. En la página web anterior se muestra un link que lleva a la plataforma *GrabCAD* donde aparecen los archivos necesarios para imprimir, mediante una impresora 3D, cada una de las piezas utilizadas.

Un ejemplo de estas piezas se puede ver en la imagen a continuación:

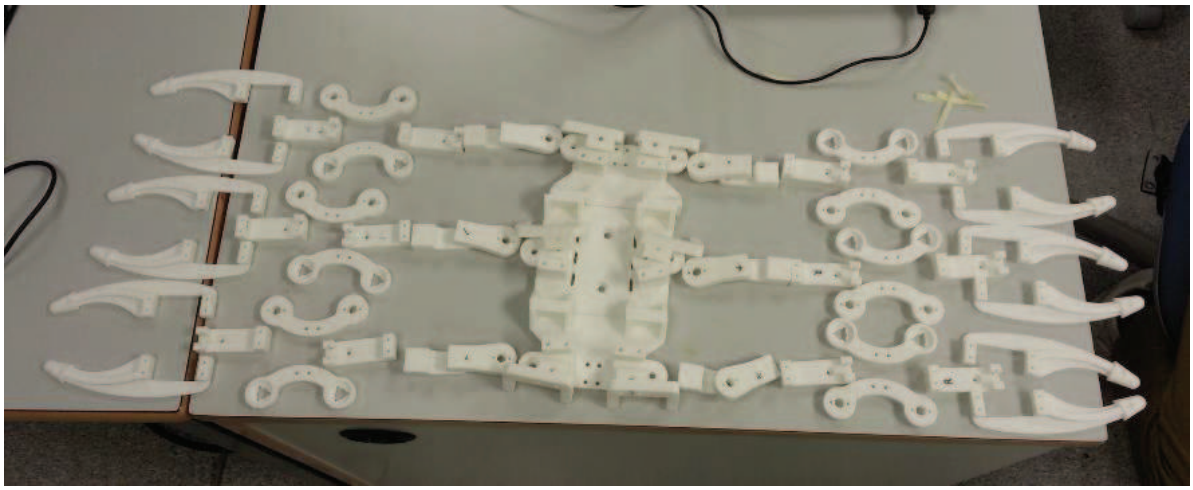

Figura 10. Piezas imprimibles que componen el hexápodo.

Este tipo de impresión es uno de los avances tecnológicos que mayor protagonismo está cogiendo en los últimos años, y se caracteriza por crear piezas a partir de cero añadiendo material a un lugar donde antes no había nada. Ésta es la diferencia principal con el mecanizado habitual, que parte de un bloque de material eliminando parte de éste para obtener la pieza deseada. Hay varias tecnologías a la hora de imprimir en 3D, las tres más importantes son *Fusion Deposition Modeling* (FDM), *Estereolitografía* (SLA) y *Selective Laser Sintering* (SLS). Las dos últimas están relacionadas con el uso del láser, lo que aporta mayor precisión al proceso. Pero en este caso se ha utilizado la FDM, cuyo procedimiento consiste en una bobina de plástico sólido que alimenta a un extrusor, el cual calienta dicho plástico hasta su punto de fusión permitiendo controlar la cantidad de plástico expulsado, mientras que a su vez se mueve por el espacio dibujando la forma deseada. Las capas de plástico se irán superponiendo hasta darle el volumen necesario.

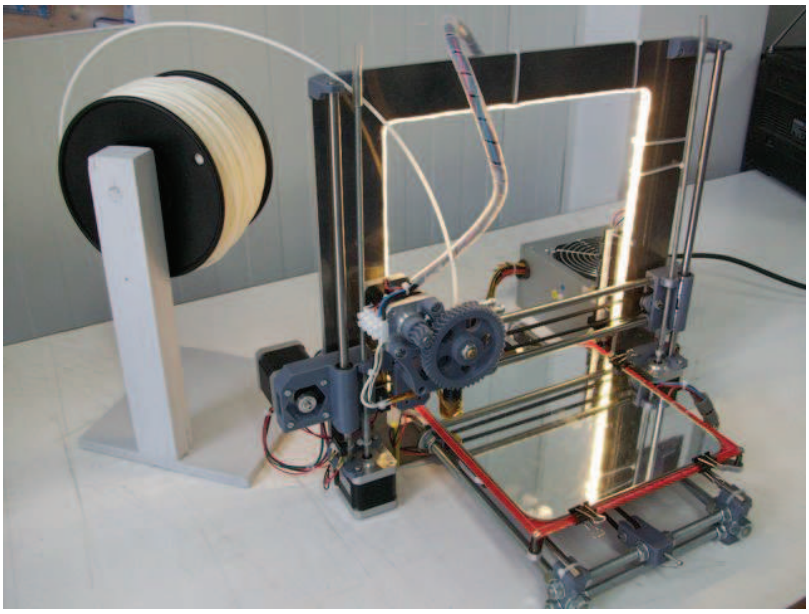

Figura 11. Impresora 3D.

Volviendo al robot, éste está compuesto de seis patas iguales, cada una con tres servos (en este caso se utilizan los servos TowerPro SG-5010).

Figura 12. Servo utilizado en la construcción del hexápodo.

Cada uno de estos servos o articulaciones une una parte del cuerpo con otra. El cuerpo central se unirá con cada uno de los fémures de las patas, a continuación vendrá la tibia y por último el radio. Por lo tanto el sistema entero dispone de 18 servos que dotaran de movimiento a las 6 patas del hexápodo en las tres direcciones del espacio. Además hay que tener en cuenta que los servos tienen un rango simétrico de giro desde -π hasta π, aunque muchas veces este rango no será alcanzable por el hecho de que algunas piezas choquen entre si.

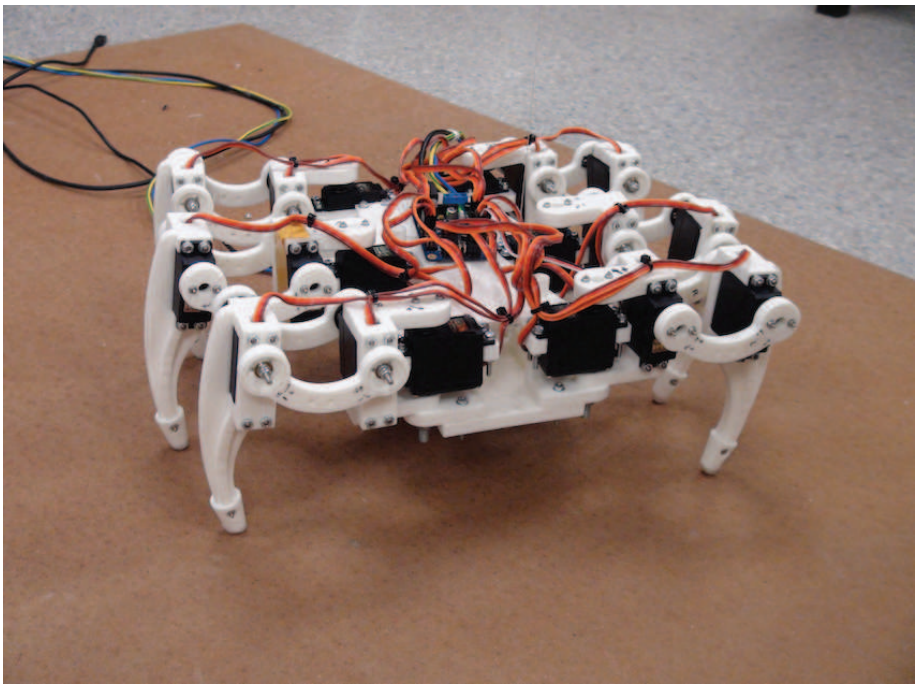

Figura 13. Hexápodo en su posición de reposo.

4.2. La Tarjeta de Adquisición de Datos: Torobot

Una vez que el robot está construido se necesitará algún medio para conectar desde el ordenador con el robot y transferirle la información. Para esta tarea se va a utilizar una tarjeta de adquisición de datos, la *Torobot*, con capacidad para 32 servos, aunque solo se utilizarán 18 (como se ha visto anteriormente). Además de poder enviarle información y que la ejecute, la tarjeta permite el almacenamiento de posiciones predefinidas, esto puede ser útil a la hora de programar el robot. La comunicación con la tarjeta se realizará vía puerto USB, por el cual se debe enviar un mensaje o telegrama con un formato concreto. En dicho telegrama los valores que podemos transferirle a la tarjeta, que estableceran la posición de los servos, son todos los números enteros incluidos en el intervalo de 500 a 2500.

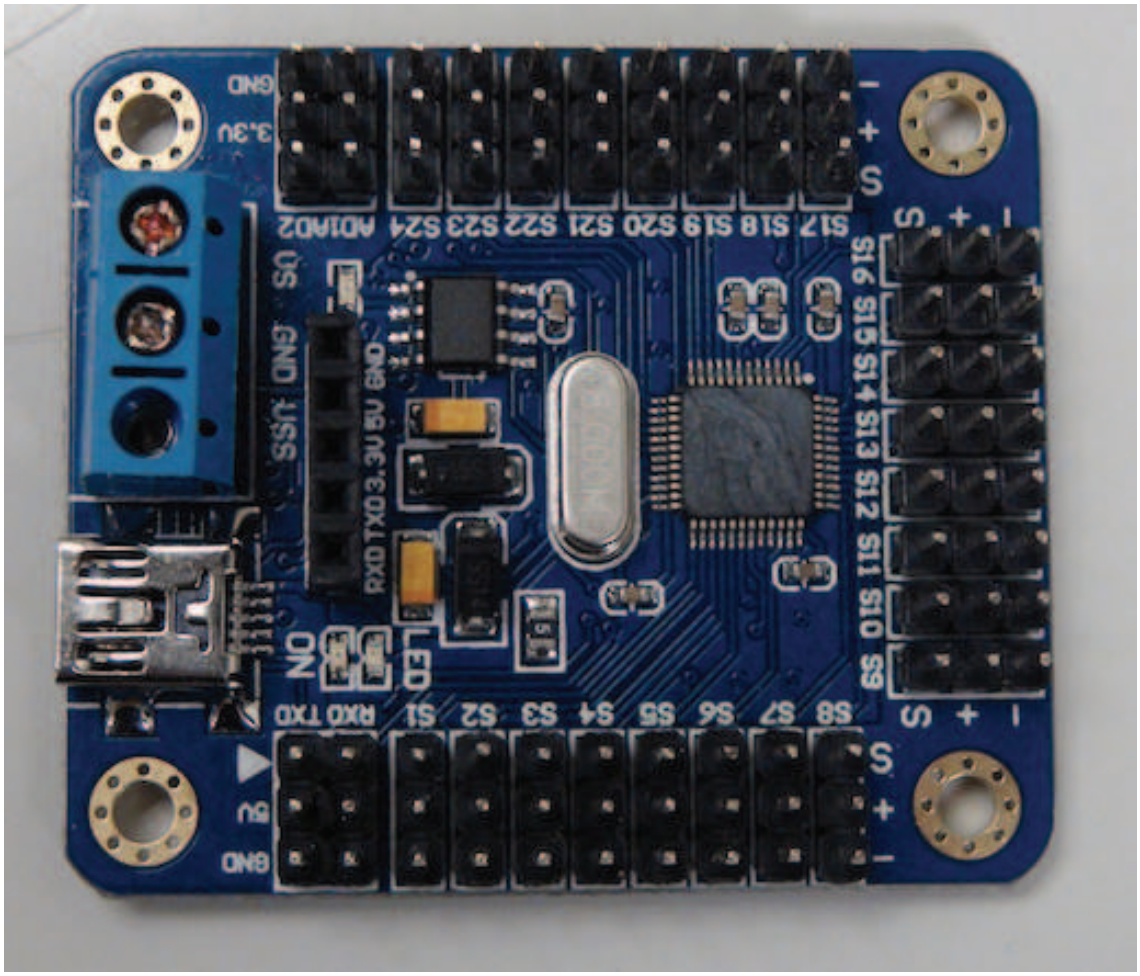

Figura 14. Tarjeta TOROBOT de 24 pines.

La sintaxis para enviarle mensajes a esta tarjeta nos viene definida en la siguiente tabla, donde se explica como controlar uno o varios servos, y el significado de cada parte del telegrama que se envía:

Nombre	Comando	Descripción
Controlar un único servo	#1P1500T100\r\n	El dato 1 hace referencia al canal del servo. El dato 1500 hace referencia a la posición del servo, con un rango de 500-2500. El dato 100 hace referencia al tiempo de ejecución y representa la velocidad, con un rango de 100-9999.
Controlar varios servos	#1P600#2P900#8P2500T100\r\n	Los datos1, 2, y 8 hacen referencia a los canales de los servos. Los datos 600, 900, y 2500 hacen referencia a la posición de los servos y los sitúa a los tres. El dato 100 hace referencia al tiempo de ejecución y representa la velocidad de los tres servos. De acuerdo al número de los servos hay sólo un tiempo o una T. El comando se ejecuta al mismo tiempo, es decir, todos los servos operan simultáneamente.
Ejecutar una única acción	#1GC2\r\n	El dato 1 se refiere al número/nombre del grupo de acción. El dato 2 se refiere al número de ciclos.
Ejecutar varias acciones	#1G#3G#1GC2\r\n	Ejecuta el primer, tercer y primer grupo de acción. El número de ciclos es 2. Un grupo en particular puede aparecer repetidamente. Sólo puede haber un único número de ciclos o C. El comando es ejecutado en secuencia, esto significa que los grupos de acciones se ejecutan en secuencia.

Tabla 1. Telegramas de la tarjeta TOROBOT.

En la propia pagina web de la empresa de la tarjeta se ofrece la posibilidad de usar un programa que permite controlar los valores que se envían a la tarjeta de manera rápida mediante una interfaz sencilla. Este programa podría ser útil para posteriores procesos como la calibración.

4.3. Calibración

Una vez establecido como comunicarse con el robot, se lleva a cabo una tarea muy importante en su configuración para su posterior uso. Dicha tarea será regular su calibración para que todas la extremidades estén ajustadas a la misma posición, dotándolo así de estabilidad. Para ello, se plantea un sistema de calibración consistente en dibujar en el suelo cuatro círculos concéntricos de 1 cm de radio el primero, 2 cm el segundo, 3 cm el tercero y así sucesivamente. Este proceso se realizará seis veces, una vez por cada pata, y en cada caso el centro de los círculos debe coincidir con la posición geométrica de la pata en su posición de reposo (ésta se obtiene a partir de las medidas geométricas del hexápodo).

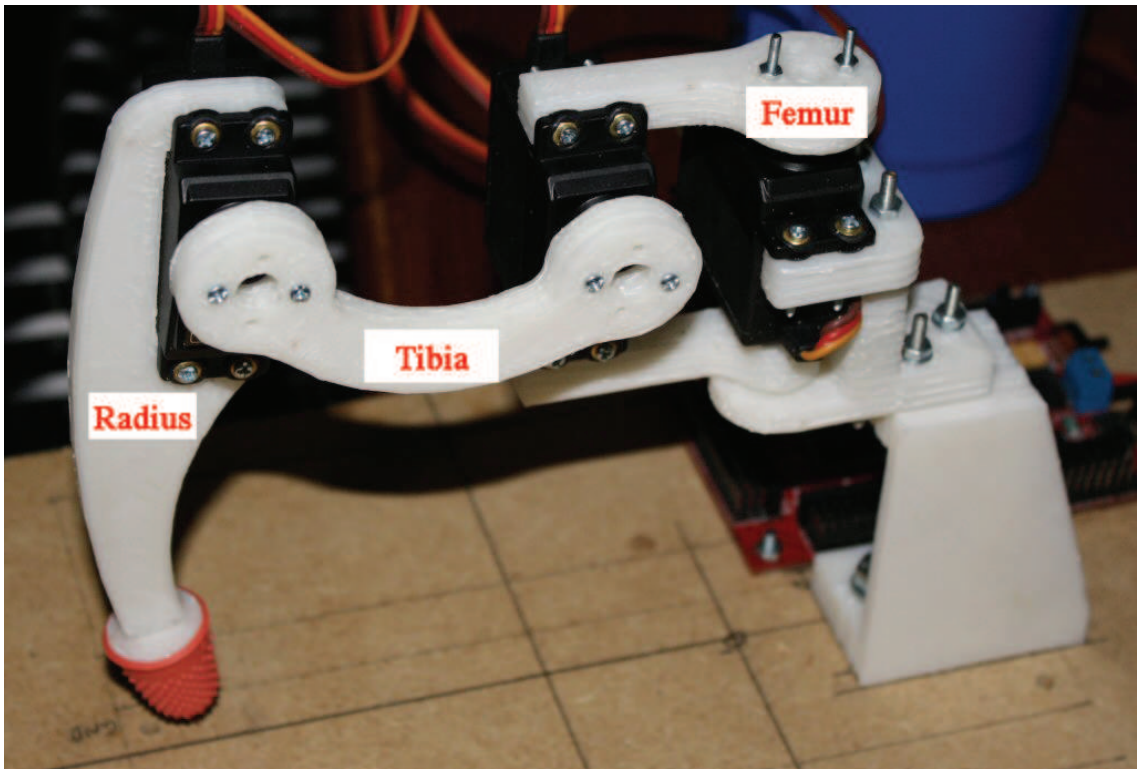

Figura 15. Partes de una pata del hexápodo.

A la hora de construirlo y colocar los servos se debe intentar que para el valor medio del servo las patas estén en una posición próxima a la inicial y no perder de esta manera la simetría del servo, ya que en la calibración es preferible que todos los valores ronden los 1500, lo que significa que todas las patas tienen el mismo margen de movimiento en las dos direcciones.

Así, una vez montado y con las patas en su posición inicial (con todos los servos a 1500) se va cambiando el valor de los servos hasta que todas las patas estén lo más próximas al centro de sus respectivos círculos. Se puede aceptar que se salgan del primer círculo, en caso de no poder ajustarse más, pero no se aceptará una posición fuera del segundo círculo. Una vez calibradas todas las extremidades, el valor de los servos para los cuales se ha conseguido la posición deseada son los valores de calibración.

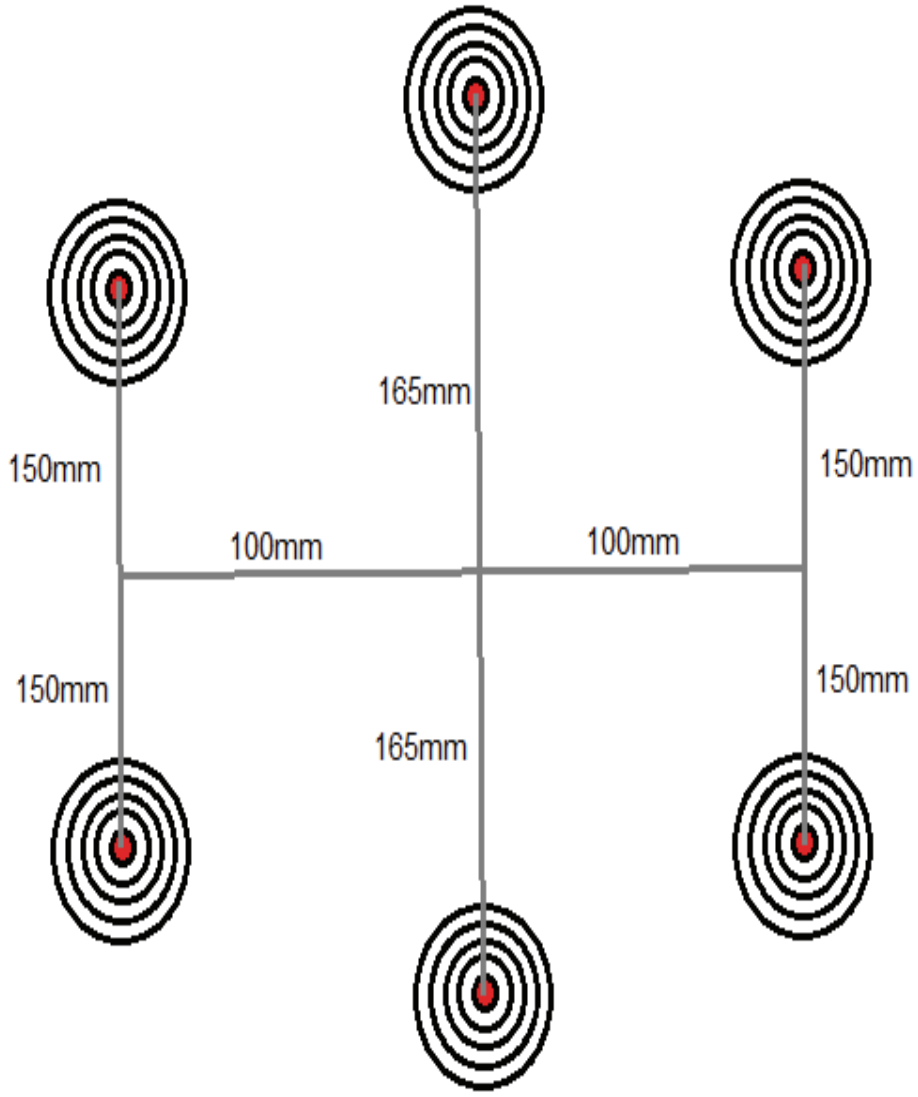

Figura 16. Sistema de calibrado.

Estos valores tienen su utilidad posteriormente cuando se calculan los ángulos a los que deben girar los servos para llevar al robot a una determinada posición. Cuando se resuelve la cinemática se considera que en la posición inicial los servos están en su ángulo de giro 0º, lo cual es un valor analógico de 1500 en la tarjeta. Pero, en la realidad, la posición inicial no se obtiene con 1500 en todos los servos debido a errores de precisión en el montaje. Por lo tanto, la solución obtenida se deberá de corregir sumando o restando al valor obtenido la diferencia del valor de calibración de cada servo con los 1500 supuestos como iniciales. Los resultados obtenidos en el proceso de calibrado para los 18 servos en su orden de colocación son:

75,0,-50,25,50,-50,40,-25,-50,-25,125,100,-75,-50,0,75,-50,-75.

5. ROBOT OPERATING SYSTEM (ROS)

Una vez diseñado y construido el robot se necesita algo con lo que controlar dicha estructura y dotarla de movimiento, algo con lo que conectar e interactuar con la tarjeta para que esta accione los servos necesarios para conseguir el objetivo de que el hexápodo camine.

ROS es la principal herramienta que se usará en el desarrollo del presente trabajo para resolver esa tarea. Como podemos ver en la propia página web[8], ROS es un sistema flexible para la escritura de software a la hora de controlar robots. Se trata de una colección de aplicaciones, bibliotecas y convenciones que tienen como objetivo simplificar la tarea de crear un comportamiento robótico complejo y robusto en una gran variedad de sistemas robóticos.

ROS surge con la idea de simplificar una tarea cada vez más utilizada como es la programación y control de robots que actualmente funcionan en muchos mercados. Simular y copiar el comportamiento humano es algo muy complejo que no esta al alcance de cualquier empresa. La tarea de ROS es recoger todo el conocimiento actual sobre la programación de la robótica e integrarlo en una única estructura. Pero ROS no es un sistema cerrado, su principal característica es el constante crecimiento aportado por aquellos que desarrollan cada una de las ramas de las que está constituido. Con ROS se consigue agrupar el trabajo de distintas personas en distintas áreas.

Esta herramienta está creciendo día a día y ya ha sido nombrada por muchos el futuro del software robótico.

Ahora mismo ROS solo está desarrollado plenamente para el sistema operativo Ubuntu, pero las versiones experimentales de Windows y Macintosh están avanzadas y pronto estará disponible para los sistemas operativos más utilizados. Su instalación es muy sencilla y se puede seguir fácilmente en su web[9].

En las nuevas versiones de ROS (*hydro* es la utilizada en este proyecto) se ha sustituido el anterior sistema de construcción *rosbuild* por *catkin*, donde se combina tanto *C++* como *Python*. Éste nuevo sistema de construcción se encargará de generar, a partir de archivos de código, los ejecutables, las librerías, y todo aquello que no sea código estático.

5.1. Espacio de Trabajo

Así que lo primero que se necesita es generar un espacio de trabajo con *catkin* (al cual se nombrará posteriormente como *catkin workspace*), donde se almacenará todo lo que se genere con esta herramienta. Para generarlo se ejecutarán los siguientes comandos:

```
mkdir –p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

El *catkin workspace* generado se divide en cuatro carpetas:

- *src* (*source*), donde se colocan los paquetes que se quieren construir, y donde se modificarán los ya existentes.

- *build*, donde el *CMake* (el compilador) se invoca para crear los paquetes.
- *devel*, donde se guardan los instaladores antes de ser instalados.
- *install*, donde se almacenan dichos instaladores una vez instalados, aunque esta carpeta no siempre existe.

Otro aspecto a tener en cuenta es que antes de trabajar siempre hay que indicar un archivo que inicializa el sitio de trabajo y permite usar las funciones de ROS. Esto se realizará en cada nuevo terminal que se abra.

```
source devel/setup.bash
```

5.2. Construcción de Paquetes

Trabajar con ROS se basa en la construcción de paquetes, que se situarán dentro de la carpeta *src* del *catkin_ws*. Para crear un paquete hay que situarse en el directorio donde se quiere crear, darle un nombre y establecer las dependencias del mismo. Estas dependencias son otros paquetes de los cuales se quiere utilizar alguna aplicación o librería a la hora de ejecutar algún elemento del nuevo paquete. Sirve para incluir en este último todas las funciones de aquellos que se definen como dependencias.

```
cd ~/catkin_ws/src
catkin_create_pkg <package_name> [depend1] [depend2]
```

El paquete es la unidad de organización de ROS. Dentro de uno habrá siempre un archivo *.xml* que contendrá la explicación del paquete, las dependencias, la versión, datos del usuario que lo ha generado y la licencia, la cual se establecerá como BSD (Berkeley Sotware Distribution). Dependiendo del uso que se le vaya a dar a cada paquete que se incluya en las dependencias su definición será distinta. Por otro lado también se creará el *CMakeLists.txt*, el código al que se llama a la hora de construir el paquete. La función de este documento es contener toda la información necesaria para la correcta construcción del nuevo paquete. En este archivo se incluirán también las dependencias, pero además se añadirán los ejecutables que se quieran generar y a partir de qué códigos se debe realizar esta tarea.

Un ejemplo de un archivo *.xml* podría ser el siguiente:

```
<!-- Versión, nombre del paquete y breve descripción del mismo-->
<?xmlversion="1.0"?>
<package>
<name>driver</name>
<version>0.0.0</version>
<description>The driver package</description>

<!-- Email del autor-->
<maintaineremail="carigba@etsii.upv.es">carles</maintainer>

<!-- Licencia -->
```

```
<license>BSD</license>

<!-- Dependencias -->

<!-- Dependencias necesarias en la compilación: -->
<build_depend>message_generation</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>trajectory_msgs</build_depend>

<!-- Dependencias necesarias en la ejecución: -->
<run_depend>message_runtime</run_depend>
<run_depend>roscpp</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>trajectory_msgs</run_depend>

<!-- Dependencias necesarias en la construcción del paquete: -->
<buildtool_depend>catkin</buildtool_depend>

<!-- Información adicional -->
<export>
</export>

</package>
```

Por su parte un *CMakeLists* tiene una apariencia similar a este ejemplo:

```
## Versión y nombre del proyecto
cmake_minimum_required(VERSION 2.8.3)
project(driver)

## Paquetes a buscar (dependencias)
find_package(catkin REQUIRED COMPONENTS
roscpp
  std_msgs
  trajectory_msgs)

##########################################
## Declaración de mensajes, servicios y acciones ##
##########################################

## Generar los siguientes mensajes
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generar los siguientes servicios
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
```

```
# )

## Generar las siguientes acciones
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

##################################
## Configuración específica de catkin ##
##################################
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES driver
#  CATKIN_DEPENDS roscpp std_msgs trajectory_msgs
#  DEPENDS system_lib
)

###############
## Construcción##
###############

include_directories(include
${catkin_INCLUDE_DIRS}
)

## Declarar un cpp ejecutable
add_executable(listener src/listener.cpp)

## Añadir dependencias del propio ejecutable
add_dependencies(listener driver_generate_messages_cpp)

## Librerías a las que vincular el ejecutable
target_link_libraries(listener ${catkin_LIBRARIES})
```

Aunque en este ejemplo no se haya incluido ningún mensaje, esto sí que puede ser necesario en paquetes que se verán posteriormente donde se enviarán o recibirán mensajes. Por otra parte, en este *CMakeLists*, los apartados tratados son los de la configuración específica de catkin, y la construcción. Aunque también existen apartados como la instalación, o las pruebas, que no son de utilidad pero que también podrían aparecer.

5.3. Elementos del Sistema

Con los paquetes, de los que se ha estado hablando, se organiza ROS, pero el sistema realmente funciona con la interacción principal de otros tres elementos: los nodos (ejecutables), los mensajes (datos) y los *topics* (temáticas). Aunque también existe un cuarto elemento necesario, el que se encarga de vigilar que todo lo anterior funcione correctamente: el *Master* (la ayuda de ROS). Este último es necesario ejecutarlo siempre en un terminal aparte para que los nodos puedan funcionar, por lo tanto con este elemento se activan todos los prerrequisitos de ROS. El comando es:

```
roscore
```

El sistema de ROS funciona de manera sencilla. Para enviar mensajes entre un nodo y otro se utilizan los *topics*. Un nodo se puede suscribir a un *topic* o publicar en él. Cuando un nodo publica algo en un *topic* todos los nodos suscritos reciben ese mensaje. Para que todo esto funcione, el tipo de mensaje que envía el publicador tiene que ser el mismo que el que espera recibir el suscriptor y por tanto, cada *topic* tiene un tipo de mensaje que se determina en su creación.

Otra manera de comunicarse, aunque no se utilizará en el presente trabajo, serían los servicios. Éstos son mensajes que tienen una pregunta y una respuesta. De esta manera el proceso no acaba en la publicación del mensaje sino que el nodo que lo ha publicado espera una respuesta por parte del suscrito a su publicación.

Los nodos se crearán dentro de la carpeta *src* del paquete que los contenga y serán un simple archivo de texto con el código (en *C* o *Python*) que queremos que se ejecute. Una vez escrito, como hemos visto en el ejemplo anterior, deberemos asociarlo al *CMakeLists* para que cuando se construya el paquete se genere el ejecutable.

Cuando el paquete esté definido, con todos los nodos creados y se quiera construir y generar su contenido, deberemos lanzar el catkin. Esta función es la principal, la que se encarga de generar instaladores, librerías, ejecutables, … a partir de archivos de código. Anteriormente a esto los archivos son simplemente texto, no se ha generado ninguna aplicación que se pueda ejecutar. Para llevarla a cabo se utilizan los siguientes comandos:

```
cd ~/catkin_ws
catkin_make
```

Una vez construido el paquete ya podemos ejecutar sus nodos, indicando el nombre del paquete donde se encuentra el nodo y el nombre del propio nodo.

```
rosrun<package_name><executable_name>
```

Otra opción para arrancar los nodos son los archivos *.launch*. Éstos nos permiten ejecutar varios nodos a la vez y obtener información de otros archivos para la correcta ejecución de los mismos. Pero en el trabajo únicamente se utiliza un archivo de este tipo, que además es creado por la propia aplicación de manera automática, por lo que no se estudiará a fondo la tipología de estos archivos. En este caso para ejecutarlos también se indica el nombre del paquete y el nombre del archivo (incluyendo la extensión *.launch*):

```
roslaunch<package_name><file_name>
```

Ante un sistema tan abierto las posibilidades con ROS para controlar el robot son muy amplias, desde utilizar las propias aplicaciones ya creadas hasta crear uno mismo esas aplicaciones. En este trabajo se tratarán dos concretamente. Por un lado, se utilizará la aplicación MoveIt! para simular el movimiento deseado del robot y que sea la propia aplicación la que calcule la solución de la

cinemática inversa, que luego se traducirá al lenguaje de la tarjeta y se le enviará. Por otro lado, se creará un nodo en el cual se implementara el código que resuelve dicha cinemática, e introduciéndole las posiciones a las que se desea que vayan las patas, será este nodo el que lo resuelva y él mismo lo enviará a la tarjeta, no habrá una parte de simulación.

6. PRIMERA SOLUCIÓN ADOPTADA: MOVEIT!

A continuación se desarrollará la primera de las alternativas de las que se compone el trabajo.

6.1. Presentación de las Aplicaciones MoveIt! y Rviz

MoveIt! es una nueva aplicación introducida en ROS para la manipulación de robots móviles. Permite la generación de trayectorias, resolución de la cinemática inversa, movimiento en planos en 3D, control del movimiento … Esta herramienta es la más moderna que incluye ROS, aunque existen otras como Rviz o Gazebo en las cuales, de manera completamente distinta, se podrían realizar las mismas operaciones. Sí que es cierto que se utilizará Rviz a partir del propio MoveIt! pero únicamente como simulador, serán las funciones de MoveIt! las que se utilicen.

Por su parte Rviz es un simulador de 3D donde se puede cargar un modelo de un robot y manipularlo, y además nos permite editar un espacio en el que se mueva el robot. También se puede utilizar para visualizar que está "viendo" el robot y que "hace". La información que reciben los sensores del robot se carga en el simulador para generar un mapa virtual de la realidad que vive el robot, Rviz permite ver el mundo exterior a través de los "ojos" del robot. Además de ver, también se nos concede la opción de actuar sobre ese mundo mediante la programación del movimiento de las partes del robot, pudiendo tener en cuenta objetos a esquivar y obteniendo las mejores planificaciones para llegar a una posición final deseada.

6.2. El Modelo Virtual

Pero como es obvio, lo primero que se necesita en este proceso es un modelo, hace falta un robot virtual que se pueda manejar y programar, y que el propio simulador pueda ver donde esta y que acciones tiene que hacer para llevarlo a donde se quiere llegar. A la hora de generar modelos estos se escriben en un archivo *.urdf*o *.xml* (es indiferente). Estos archivos tienen una sintaxis especifica que permitirá reproducir lo más fielmente nuestro robot.

Un archivo *.urdf* consiste en describir el robot a partir de sus*links*, las piezas del robot, y *joints*, uniones entre los *links*. Así partiendo de un centro se irán definiendo las piezas diciendo donde se encuentra su centro geométrico respecto al *joint* anterior. Una vez definidos varios *links* se definen los *joints*, indicando su posición respecto al *joint* anterior, y nombrando de que *link* provienen (*parent link*) y a que *link* van (*child link*). Así, como si fuera un árbol con varias ramas, se van nombrando los *links* y *joints* que constituyen el modelo completo del robot.

Una vez conocidos los dos elementos fundamentales del modelo, y una explicación muy sencilla de los mismos, se puede ampliar la información de cada uno de ellos.

Para los *links* lo primero que debemos hacer es darles un nombre. A continuación se pueden definir varios aspectos, algunos de ellos opcionales y que en este trabajo no han sido necesarios.

Uno de ellos es la inercia de la pieza, donde habrá que indicar el centro de gravedad de la pieza con respecto al *joint* anterior, la masa del *link* y los valores de la matriz de inercia. Este aspecto es influyente a la hora de interactuar con el mundo, como en el presente trabajo el interés reside únicamente en la resolución de la cinemática, no se ha desarrollado en el modelo.

El siguiente elemento a definir es el aspecto visual del *link*, donde se volverá a definir un origen pero en este caso será el origen geométrico de la pieza respecto al *joint* anterior, a continuación se definirá la geometría de la pieza existiendo la posibilidad de ser un cilindro, una esfera, un prisma rectangular o una forma predefinida en un archivo (*meshes*). En el último caso será necesario disponer de un archivo *.stl* o *.dae*,en el que se defina la forma de la pieza, dentro del paquete del modelo, y hacer referencia a ese archivo a la hora de definir la geometría. Y por último en la parte visual también se podrá definir el color de la pieza y su textura.

Para acabar, también se puede establecer la colisión del *link*. Esto es como el espacio propio de la pieza y no tiene porque ser igual que la pieza visual. Cuando se calculen trayectorias y movimientos, dos *links* chocarán, y por lo tanto no será una solución válida, cuando sus modelos de colisión entren en contacto entre sí. Muchas veces se hace el modelo de colisión más sencillo que el visual para ahorrar tiempo computacional, aunque a veces se puedan dar posiciones no válidas para el modelo pero que en la realidad sí que lo serían. En la parte de colisión se volverá a definir un centro geométrico de la pieza y la geometría que debe tener. En el caso de que no se defina la colisión, el programa la define automáticamente con el mismo modelo que la visual. Como el modelo del hexápodo se ha hecho de forma aproximada esta opción será valida y no se definirá la colisión.

Un ejemplo de cómo se define un *link* en un *.urdf* es el siguiente:

```
<linkname="my_link">
<inertial>
<originxyz="0 0 0.5"rpy="0 0 0"/>
<massvalue="1"/>
<inertiaixx="100"ixy="0"ixz="0"iyy="100"iyz="0"izz="100"/>
</inertial>

<visual>
<originxyz="0 0 0"rpy="0 0 0"/>
<geometry>
<boxsize="1 1 1"/>
</geometry>
<materialname="Cyan">
<colorrgba="0 1.0 1.0 1.0"/>
</material>
</visual>

<collision>
<originxyz="0 0 0"rpy="0 0 0"/>
<geometry>
<cylinderradius="1"length="0.5"/>
```

```
</geometry>
</collision>
</link>
```

Por su parte, los *joints* se definen de manera más breve y diferente. Lo primero que hay que hacer es darles un nombre y una tipología. El tipo establecerá la clase de movimiento que tendrá esa articulación, así pueden ser: *revolute* (giran respecto a un eje con unos límites), *continuous* (giran respecto a un eje sin limites de giro), *prismatic* (se desplaza a lo largo de un eje), *fixed* (no se mueven), *planar* (se mueven a lo largo de un plano) y *floating* (son articulaciones libres con 6 grados de libertad).

Una vez definido el tipo y el nombre del *joint* hay que fijar varios parámetros obligatorios: su origen respecto al *joint* anterior, el *parent link* y el *child link*. A continuación si hemos definido algún movimiento respecto a un eje también habrá que indicar que eje es. Y por último se pueden fijar algunos límites, como los límites de giro o desplazamiento alrededor de un eje (*upper* y *lower*), los límites de velocidad de dicho eje y el esfuerzo que puede soportar el *joint*.

Un ejemplo de cómo se define un *joint* en un *.urdf* es el siguiente:

```
<jointname="my_joint"type="floating">
<originxyz="0 0 1"rpy="0 0 3.1416"/>
<parentlink="link1"/>
<childlink="link2"/>

<limiteffort="30"velocity="1.0"lower="-2.2"upper="0.7"/>
</joint>
```

Una vez ya se conocen las diferentes posibilidades a la hora de diseñar un modelo se crea el del robot. Lo primero es darle un nombre, y posteriormente definir todos sus *links* y *joints* uniendo entre si las piezas pertinentes hasta tener modelado todo el hexápodo. A continuación se muestra un ejemplo de cómo se definiría una pata en el archivo *.urdf* del hexápodo, no se incluye entero por su larga extensión pero sí se puede ver completo en el anejo al final del documento.

```
<robotname="hexapod">

<linkname="body_central">
        <visual>
        <originxyz="0 0 0.065"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.2 0.1 0.13"/>
        </geometry>
        <materialname="white">
        <colorrgba="1 1 1 1"/>
        </material>
        </visual>
</link>

<linkname="frontal_right_femur">
        <visual>
        <originxyz="0 0.0175 0"rpy="0 0 0"/>
```

```
        <geometry>
        <boxsize="0.05 0.035 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_to_frontal_right"type="revolute">
        <axisxyz="0 0 1"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0.1 0.05 0.0975"/>
        <parentlink="body_central"/>
        <childlink="frontal_right_femur"/>
</joint>

<linkname="frontal_right_tibia">
        <visual>
        <originxyz="0 0.0325 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.07 0.065 0.065"rpy="0 0 0"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_right_femur_to_tibia"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.035 0"/>
        <parentlink="frontal_right_femur"/>
        <childlink="frontal_right_tibia"/>
</joint>

<linkname="frontal_right_radius">
        <visual>
        <originxyz="0 0.015 -0.0325"rpy="0 0 0"/>
        <geometry>
        <cylinderlength="0.13"radius="0.015"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_right_tibia_to_radius"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.065 0"/>
        <parentlink="frontal_right_tibia"/>
        <childlink="frontal_right_radius"/>
</joint>

<linkname="frontal_right_knee">
```

```
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_right_radius_to_knee"type="continuous">
        <axisxyz="1 0 0"/>
        <originxyz="0 0.015 -0.0975"/>
        <parentlink="frontal_right_radius"/>
        <childlink="frontal_right_knee"/>
</joint>

<linkname="frontal_right_feet">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_right_knee_to_feet"type="continuous">
        <axisxyz="0 0 1"/>
        <originxyz="0 0 0"/>
        <parentlink="frontal_right_knee"/>
        <childlink="frontal_right_feet"/>
</joint>
```

Como se puede apreciar, la pata está formada por cinco elementos, cuando en la realidad son tres piezas las que la forma. La inclusión de los *links* asignados como *knee* y *feet* es debido a la necesidad de tener completa libertad a la hora de mover la pata en el simulador. Esto se explicará posteriormente en mayor detalle.

Cuando se genera un archivo *.urdf* existe una herramienta para comprobar que su sintaxis es correcta y presenta de manera breve como es el modelo creado. El comando es el siguiente, donde en el nombre del archivo se debe incluir la extensión *.urdf*:

```
check_urdf <urdf_file_name>
```

Una vez creado el archivo se forma un paquete donde almacenarlo. En este caso se le denomina *hexapod*, y se establece como dependencia el paquete *roscpp*, aunque no sería estrictamente necesario ya que no se va a escribir ningún archivo en *C++*.

6.3. Generación del Paquete del Simulador

Ahora que ya esta definido el modelo hay que cargarlo en el simulador. Esto podría hacerse directamente desde el Rviz, y posteriormente moverlo con algún nodo que le enviara instrucciones al simulador. Pero esta herramienta es muy floja en este ámbito, por ello se usará el MoveIt!.

Lo primero que se necesita hacer es generar el paquete que contenga toda la información para que el simulador permita el control y manejo del robot. Para ello se utilizará una herramienta fundamental del MoveIt!, el moveit_setup_assistant. Con esta herramienta se introduce el modelo, y a partir de este se generan todos los archivos y elementos necesarios para la simulación.

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

Al comenzar con el moveit_setup_assistant se debe localizar el paquete donde se ha creado el modelo, y cargarlo en la aplicación.

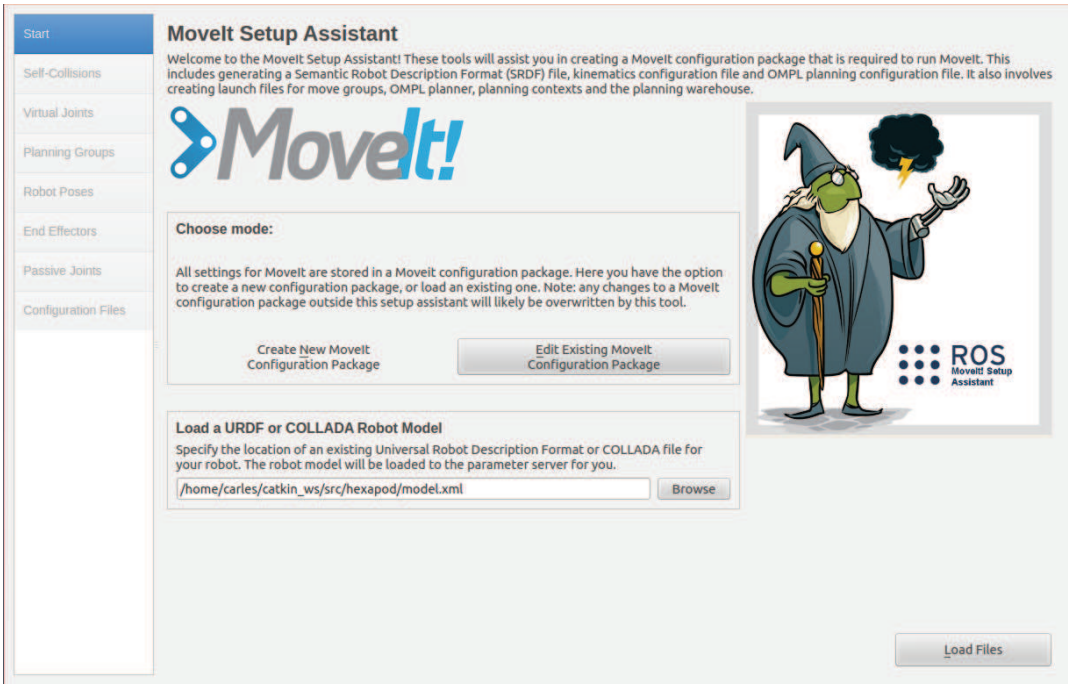

Figura 17. Moveit_setup_assistant: Ventana de inicio.

Una vez realizado, lo siguiente es generar lo que se denomina la *Self-Collision Matrix*. Esta tarea consiste en leer todos los *joints* del modelo y ver cuales no deben de ser estudiados cuando se chequee la colisión al planificar una trayectoria debido a múltiples razones, como puede ser que siempre estén en colisión o que no puedan estarlo nunca. Esto ahorrará tiempo a la hora de planificar el movimiento. Esta tarea la realiza automáticamente la aplicación clicando en el botón que aparece en pantalla. El resultado sería algo similar a esto:

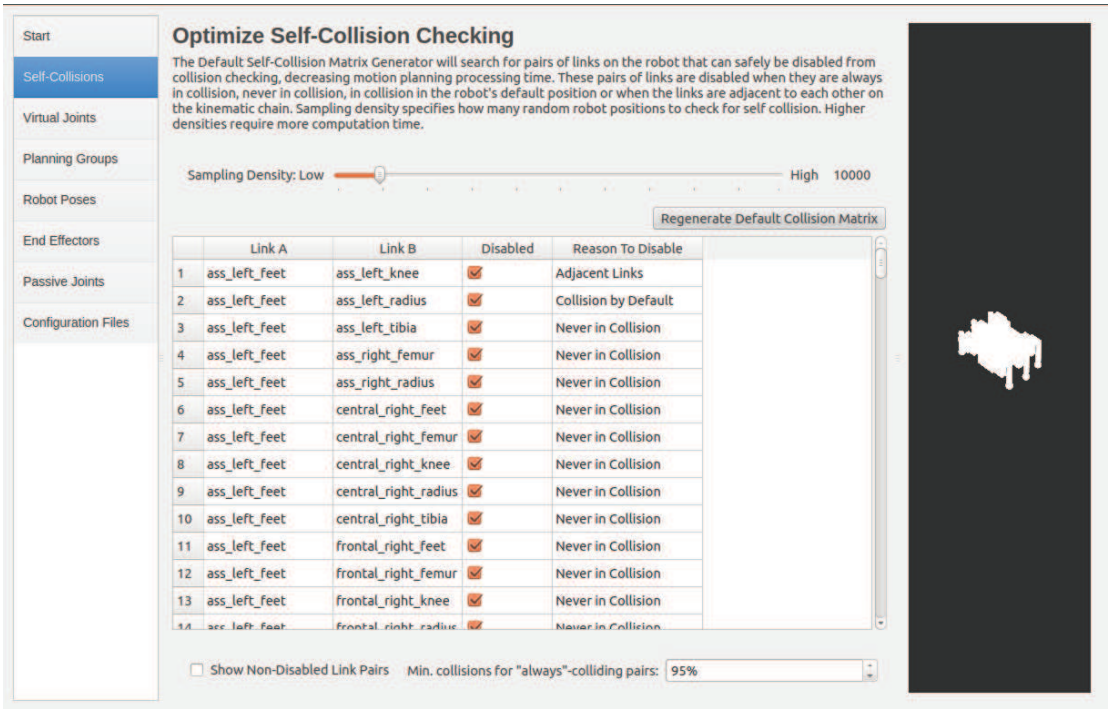

Figura 18. Moveit_setup_assistant: Ventana de la matriz de colisiónes.

En segundo lugar se definen los *virtual joints*, *joints* que no existen realmente en el robot pero sirven para unirlo con el mundo. En este caso no es necesario ya que el hexápodo no está unido por ningún punto al mundo, se mueve libremente por él. Pero la aplicación te permite generar un *floating joint* que haría el mismo efecto.

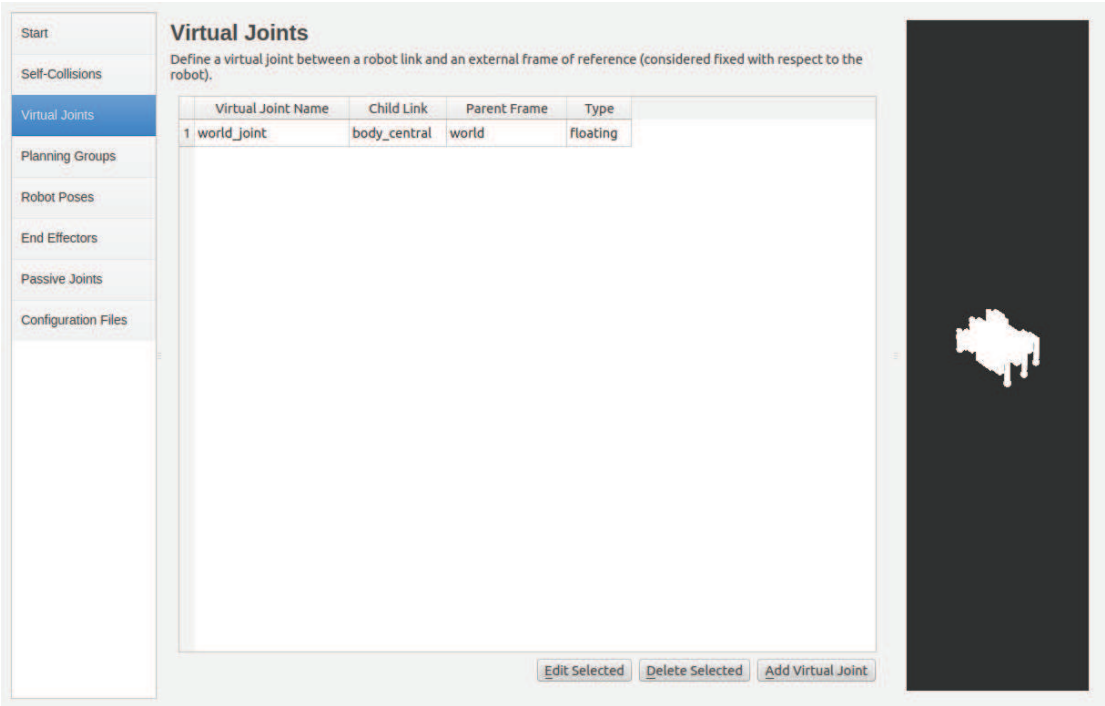

Figura 19. Moveit_setup_assistant: Ventana de los virtual joints.

A continuación viene una de las partes más importantes del moveit_setup_assistant, la generación de grupos dentro del modelo. En esta ventana se pueden generar grupos que luego se podrán manipular individualmente. Para generar uno de ellos primero se le debe dar un nombre, luego decidir con que modelokinemático se quiere que se resuelva la cinemática inversa de ese grupo, y por último decidir que partes forman este grupo. Esto último lo podemos hacer eligiendo los *joints* que deben incluirse, eligiendo los *links* que lo forman, formando una *chain* (cadena) de *links* que lo constituyen eligiendo simplemente el primero y el último, o formar un grupo a partir de la unión de otros.

En este proyecto se creará un grupo independiente para cada pata que incluya fémur, tibia, radio, rodilla y pie de cada una, otro denominado base que incluirá únicamente el *virtual joint*, otro uniendo los dos *links* que forman el cuerpo del hexápodo, un grupo que agrupe a todas las patas, y un último grupo que incluya todos los *joints* del modelo. Este último grupo será el que se utilice a la hora de programar el movimiento en el simulador.

En las siguientes imágenes se pueden ver todos estos grupos creados, y la creación de uno de ellos a modo de ejemplo:

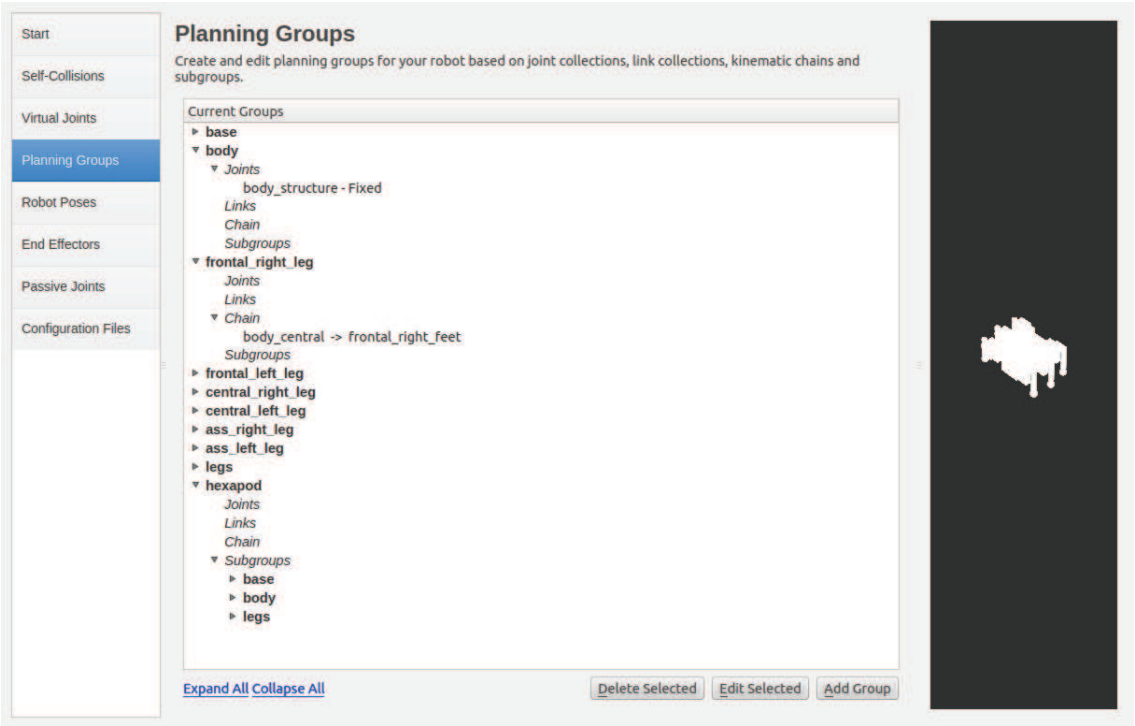

Figura 20. Moveit_setup_assistant: Ventana de la planificación de grupos.

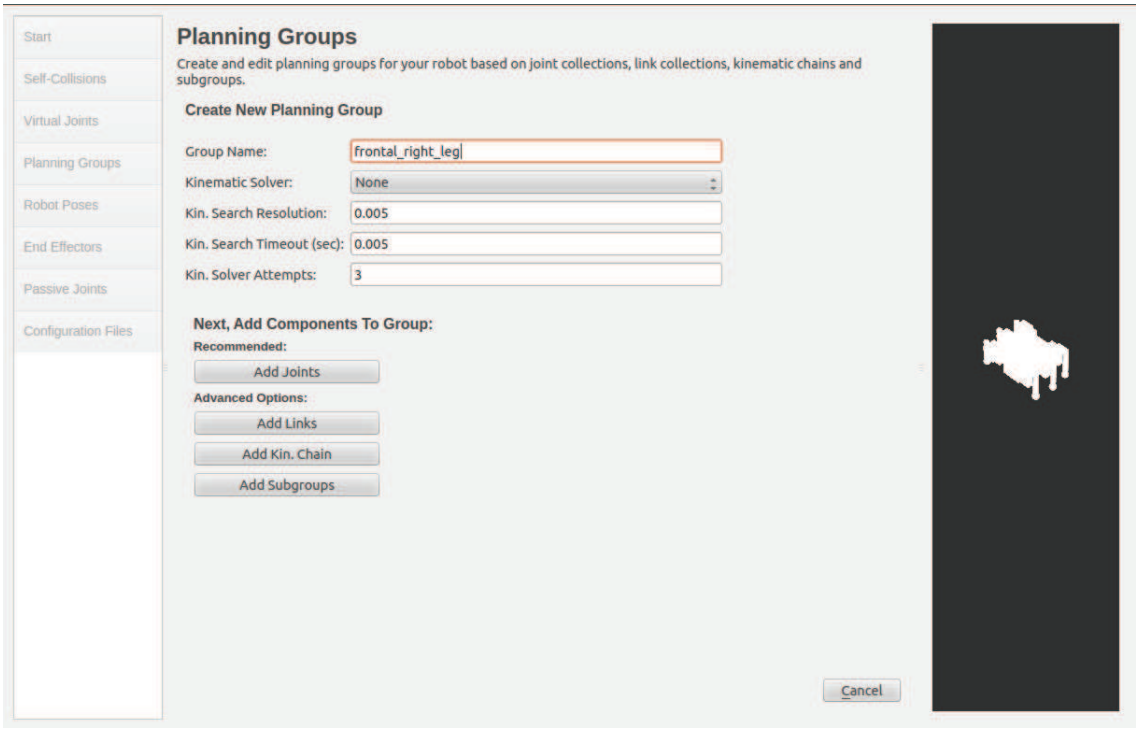

Figura 21. Moveit_setup_assistant: Ejemplo de generación de un nuevo grupo.

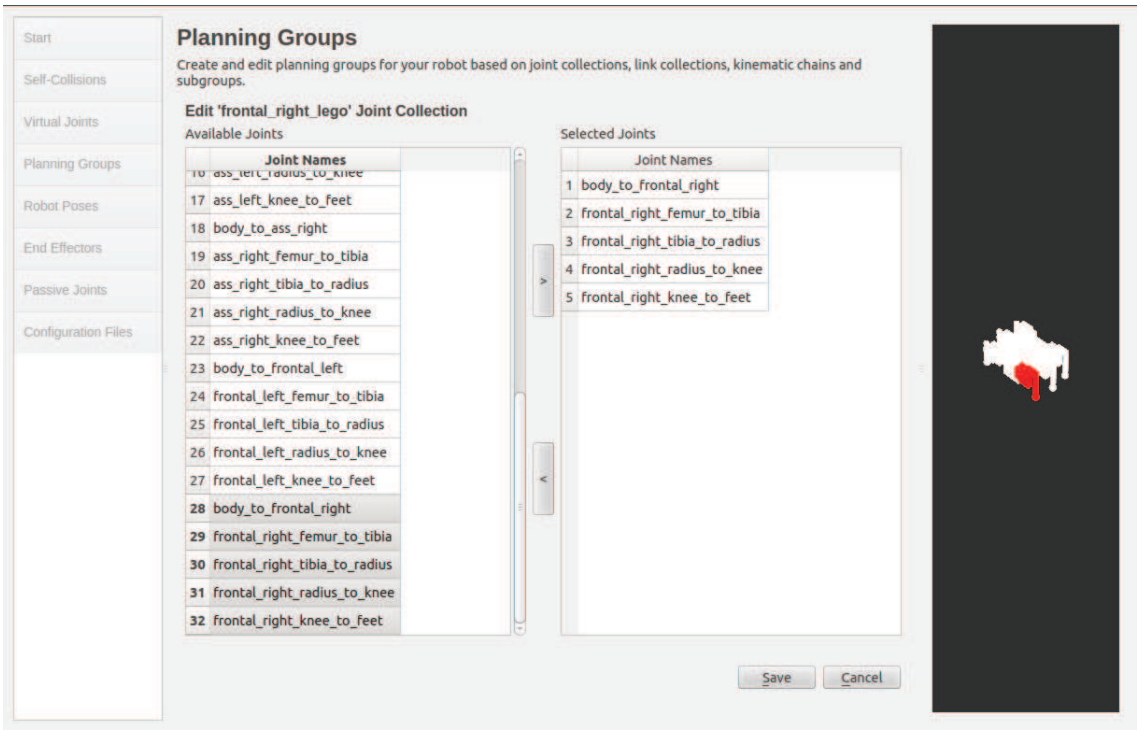

Figura 22. Moveit_setup_assistant: Ejemplo de selección de partes de un grupo.

Una vez ya se han creado los grupos, cada uno de ellos se podrá manipular de manera completamente independiente, y por lo tanto programar o controlar. Además si se ha creado un grupo que incluye varios, como es el caso, se podrán controlar varios grupos a la vez. Por lo tanto en este trabajo se ha buscado la posibilidad de poder programar el movimiento de las seis patas en conjunto y no por separado.

La siguiente ventana ofrece una herramienta muy útil, pero que no será usada en este caso por el enfoque del proyecto. Aquí se oferta la posibilidad de almacenar posiciones de cada grupo que luego se podrán establecer como posiciones iniciales o finales. La desventaja que presenta es que estas posiciones se definen a partir de ángulos de giro de los *joints*, y por lo tanto si nosotros definimos una posición a partir de ángulos no necesitamos una simulación para que nos diga que giro deben de tener los servos para llegar a dicha posición, ese dato ya es conocido, es el dato que se introduce. Lo que interesa en este proyecto es decir una posición en cartesianas (en las tres dimensiones del espacio) y que él resuelva la cinemática inversa y dé como solución el giro de los servos para llegar. Por eso esta ventana carece de utilidad en este caso, pero en otras aplicaciones podría ser de gran importancia. En la siguiente imagen se muestra un ejemplo de su utilización:

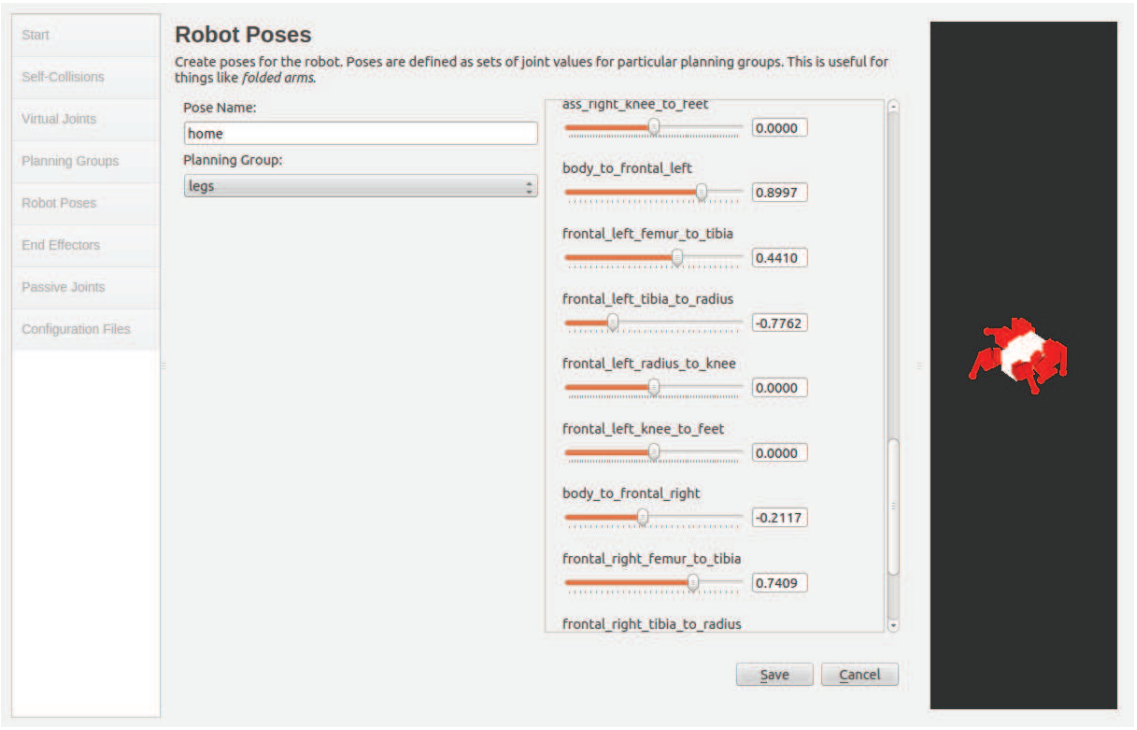

Figura 23. Moveit_setup_assistant: Ventana de configuración de posiciones.

Lo siguiente son la definición de los *end effectors*. Al igual que la posibilidad de fijar posiciones, esta herramienta es muy útil pero no en el caso del hexápodo. Su función es definir a un grupo como límite de otro. Un ejemplo muy claro puede ser un brazo robótico, donde el *end effector* del brazo puede ser la propia pinza (que es un grupo de *links* y *joints*). Esto te aporta nuevas posibilidades como por ejemplo, permitir desplazar la pinza en el simulador y que el brazo se mueva para que la pinza esté en la posición indicada. Lo que ocurre es que si no se define ninguno, el simulador por defecto tomará como *end effector* el último *link* del grupo. En el caso del hexápodo no es necesario definir un *end effector* ya que el último *link* de cada pata es lo que se quiere mover por el espacio, por lo tanto la asignación de este por defecto es válida.

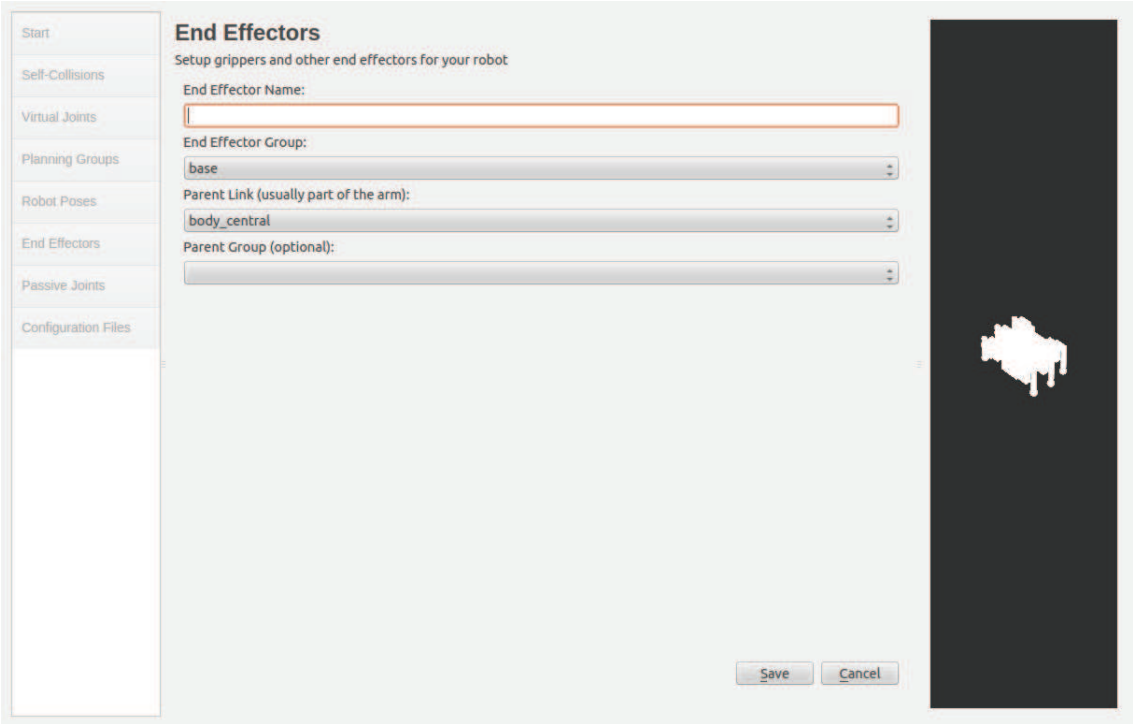

Figura 24. Moveit_setup_assistant: Ventana de definición de los end effectors.

La penúltima ventana trata sobre los *passive joints*, estos se definen para indicarle al planificador que estos *joints* no tiene que tenerlos en cuenta. En este proyecto se podría utilizar para el mismo *virtual joint* o el *joint*que une las dos piezas del cuerpo central. Aquí podemos ver un ejemplo:

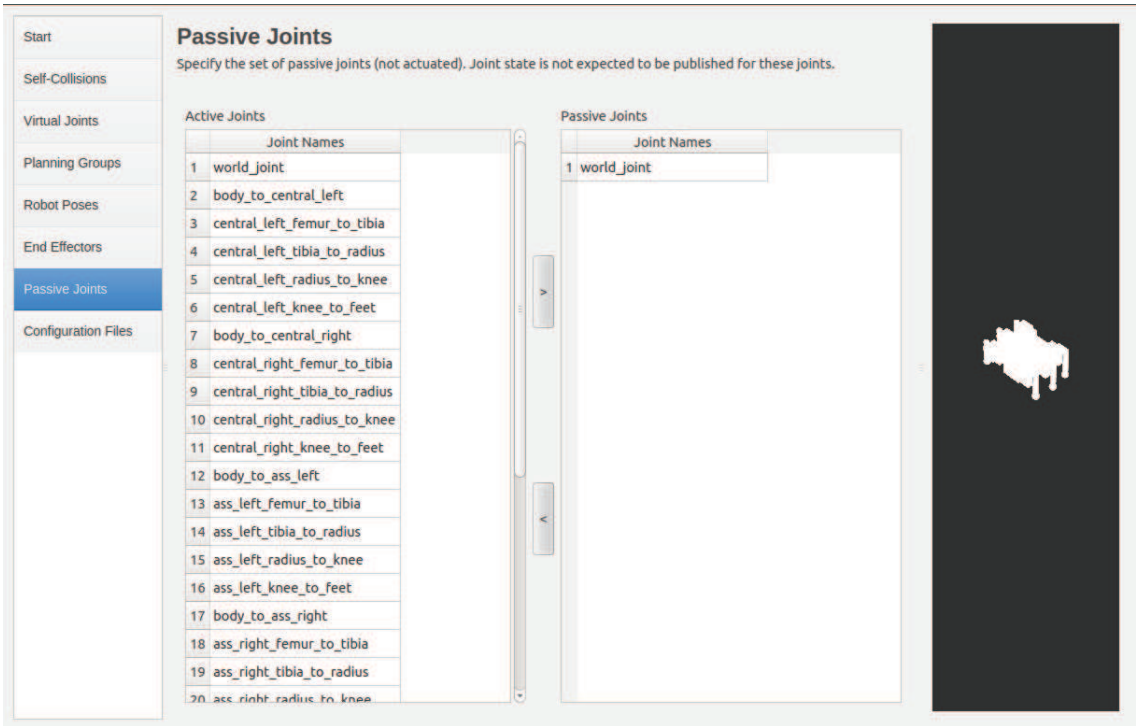

Figura 25. Moveit_setup_assistant: Ventana de los pasive joints

Y ya por último, para acabar con esta aplicación, la ventana donde se generan los archivos necesarios para el correcto funcionamiento de MoveIt! y la utilización del simulador. En este caso los archivos se crean por defecto, y muchos de ellos se pueden modificar, incluso es interesante añadir algunos para obtener diversas funciones, como puede ser la generación de controladores, o un sistema de resolución kinemático distinto al genérico de MoveIt!. Así para cerrar la aplicación, se deberá buscar el paquete en el que se quiere que se generen los archivos, seleccionarlo y clicar en generar paquete. Previamente se deberá haber creado esta carpeta con el nombre deseado dentro del *src* del *catkinworkspace*. En este caso el paquete se nombra como *moveit_hexapod*. Posteriormente sería necesario construir el paquete con la herramienta *catkin_make*. La última ventana del *moveit_setup_assistant* por lo tanto resultaría:

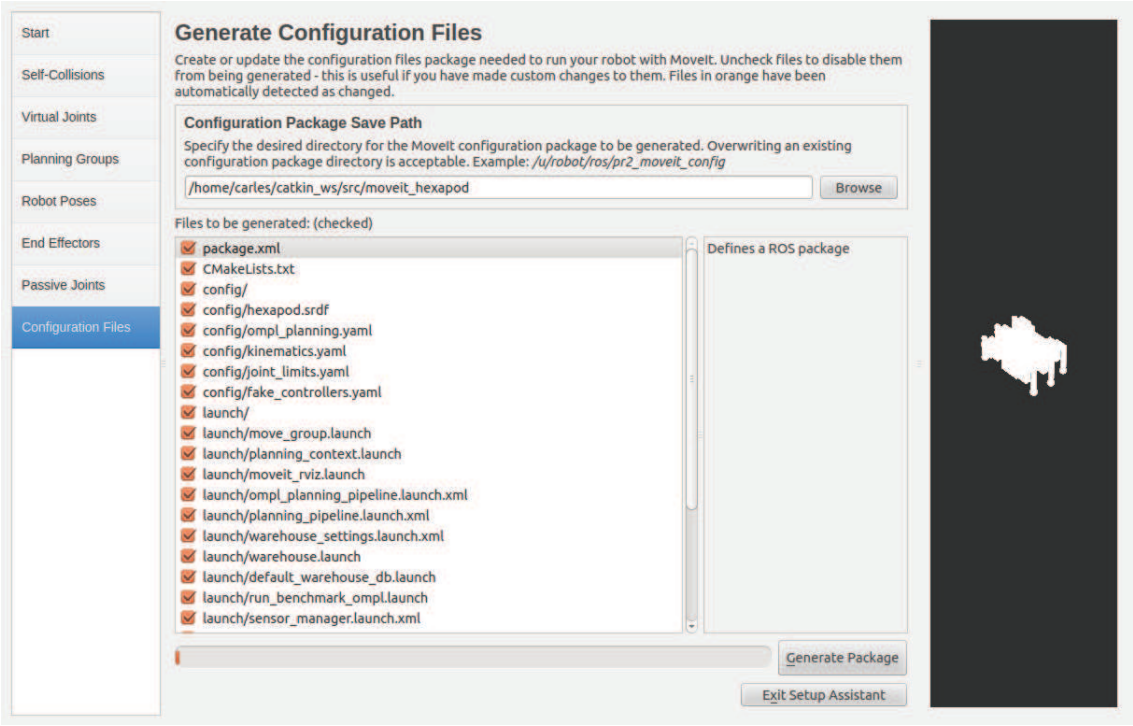

Figura 26. Moveit_setup_assistant: Ventana de la configuración de archivos.

Dentro de este paquete se habrán generado dos carpetas tras el uso de la aplicación *moveit_setup_assistant*, una denominada *config* y la otra *launch*. La primera, como su propio nombre da a entender, contiene los archivos que permiten la configuración del paquete. Aquí es donde, si se desea, se puede crear un archivo donde se generen unos controladores distintos a los establecidos por defecto, denominados *fake controllers*. Un ejemplo sería el siguiente archivo denominado *controllers.yalm*, donde se define el nombre, la acción que debe hacer sobre los *joints*, el tipo de controlador y los *joints* a controlar:

```
controller_list:
  - name: hexapod_controller
    action_ns: follow_joint_trajectory
type: FollowJointTrajectory
default: true
    joints:
[ass_left_femur_to_tibia,ass_left_knee_to_feet,ass_left_radius_to_knee,ass_left_tibia_to_radius,
ass_right_femur_to_tibia,ass_right_knee_to_feet,ass_right_radius_to_knee,ass_right_tibia_to_radi
```

us,body_to_ass_left,body_to_ass_right,body_to_central_left,body_to_central_right,body_to_fronta
l_left,body_to_frontal_right,central_left_femur_to_tibia,central_left_knee_to_feet,central_left_radi
us_to_knee,central_left_tibia_to_radius,central_right_femur_to_tibia,central_right_knee_to_feet,ce
ntral_right_radius_to_knee,central_right_tibia_to_radius,frontal_left_femur_to_tibia,frontal_left_k
nee_to_feet,frontal_left_radius_to_knee,frontal_left_tibia_to_radius,frontal_right_femur_to_tibia,f
rontal_right_knee_to_feet,frontal_right_radius_to_knee,frontal_right_tibia_to_radius,world_joint]

En este caso se le han dado nombres reales. Se podrían haber escrito una numeración de *joints* (joint_1, joint_2, …) y crear otro archivo, *joint_names.yalm* donde se indicasen los nombres únicamente:

controller_joint_names:
[ass_left_femur_to_tibia,ass_left_knee_to_feet,ass_left_radius_to_knee,ass_left_tibia_to_radius,
ass_right_femur_to_tibia,ass_right_knee_to_feet, …frontal_right_tibia_to_radius,world_joint]

Aunque todo esto no es necesario ya que no se va a conectar directamente el robot al simulador, sino simplemente obtener la solución de la simulación y enviarla por otro lado.

Otro aspecto importante dentro de este directorio es el archivo *hexapod.srdf*, este es un resumen de todo lo que se ha realizado dentro del moveit_setup_assistant, generación de grupos, posiciones, la matriz de colisión, … Este archivo puede verse en el anejo del documento.

La otra carpeta que se genera es la de archivos *launch*. En esta la cantidad de documentos es mayor, todos ellos documentos ejecutables con diversas funciones. En este caso se utiliza el más básico, el *demo.launch*, aunque con las herramientas necesarias se podría generar otro parecido pero que no usara las funciones por defecto del MoveIt!, incluir tus propios controladores, tu propio sistema de resolución cinemático… Como se verá el archivo es muy sencillo y hay opciones más avanzadas que no se implementan.

El código es el siguiente:

```
<launch>

<arg name="db" default="false" />

<arg name="debug" default="false" />

<!-- Se carga el archivo .urdf y en caso de otros archivos .yalm como los controladores-->
<include file="$(find moveit_hexapod)/launch/planning_context.launch">
<arg name="load_robot_description" value="true"/>
</include>

<node pkg="tf" type="static_transform_publisher" name="virtual_joint_broadcaster_0" args="0 0
0 0 0 0 world body_central 100" />


<!-- Al no tener un robot conectado se publican los estados de los nodos de manera “falsa” -->
<node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
<param name="/use_gui" value="false"/>
<rosparam param="/source_list">[/move_group/fake_controller_joint_states]</rosparam>
</node>
```

```
<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="true" output="screen" />

<!-- Se ejecuta el nodo move_group -->
<include file="$(find moveit_hexapod)/launch/move_group.launch">
<arg name="allow_trajectory_execution" value="true"/>
<arg name="fake_execution" value="true"/>
<arg name="info" value="true"/>
<arg name="debug" value="$(arg debug)"/>
</include>

<!-- Se arranca el simulador-->
<include file="$(find moveit_hexapod)/launch/moveit_rviz.launch">
<arg name="config" value="true"/>
<arg name="debug" value="$(arg debug)"/>
</include>

<include file="$(find moveit_hexapod)/launch/default_warehouse_db.launch" if="$(arg db)"/>

</launch>
```

6.4. El Simulador

Por lo tanto, una vez visto el *demo.launch*, lo siguiente es ejecutarlo. Cuando se hace esto se abre el simulador, Rviz, y aparece en pantalla el modelo del hexápodo. En esta nueva ventana hay tres partes diferenciadas, tal y como se observa en la siguiente imagen. En el panel de arriba a la izquierda, habrá una sección con varias opciones, una de ellas denominada *MotionPlanning*. Aquí es donde se pueden fijar los parámetros de la planificación. Debajo de esta habrá otro panel que nos permitirá interactuar con la simulación desde otro punto de vista. Y a la derecha está la simulación del modelo en un mundo virtual.

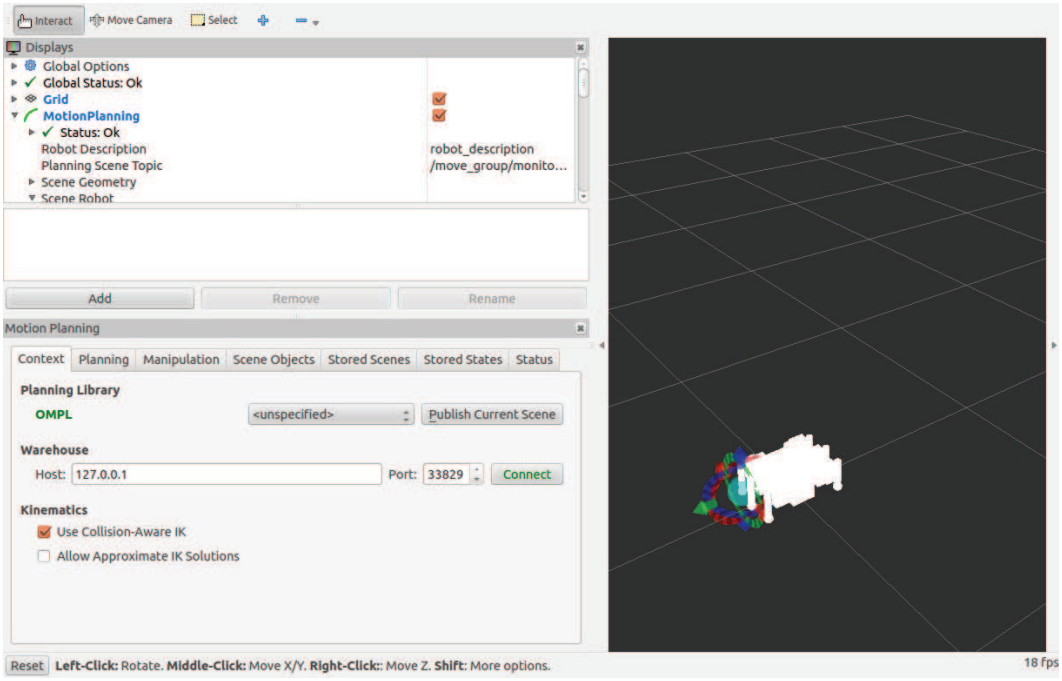

Figura 27. Ventana del simulador.

En la parte superior izquierda, como se ha indicado, existen diferentes desplegables. Las opciones generales, el estado del robot, … pero la parte importante es la de *MotionPlanning*. De todas las opciones que se ofrecen, solo se utilizarán tres de ellas: *Scene Robot, Planning Request* y *PlannedPath*. La primera es útil para ver la posición del robot, esto ocurrirá si la casilla *Robot Scene* está activa. La siguiente, más importante, es donde se elige el grupo que se desea mover de los que se han creado en el *moveit_setup_assistant* (este se selecciona en el desplegable de *Planning Group*), y donde podremos definir que se nos muestren las posiciones iniciales y finales definidas para el movimiento. Por defecto, la posición inicial se indicará en verde y la posición final en naranja. Por último en *Planned Path* se ofrece la posibilidad de ver la posición del robot mientras ejecuta la planificación (*Show Robot Visual*) y también se puede seleccionar la opción de que la trayectoria se muestre en estado sólido, es decir, mostrar a la vez todas las posiciones por las que pasa el robot (*Show Trail*). Visualmente sería:

Figura 28. Panel de display del simulador.

El siguiente panel, el inferior izquierdo, permite interactuar con el robot y el mundo. Este está compuesto por varias ventanas. En este proyecto se ha utilizado únicamente la segunda de ellas, la de *Planning*, donde se ofrece la posibilidad de fijar posiciones finales o iniciales, y donde se encuentran los botones para iniciar la planificación del robot para que se mueva entre dichas posiciones y ejecutar el movimiento. En otras ventanas se podrían añadir objetos, previamente creados, en el entorno del robot para que este interactúe con ellos y almacenar algunos de estos entornos para futuros usos.

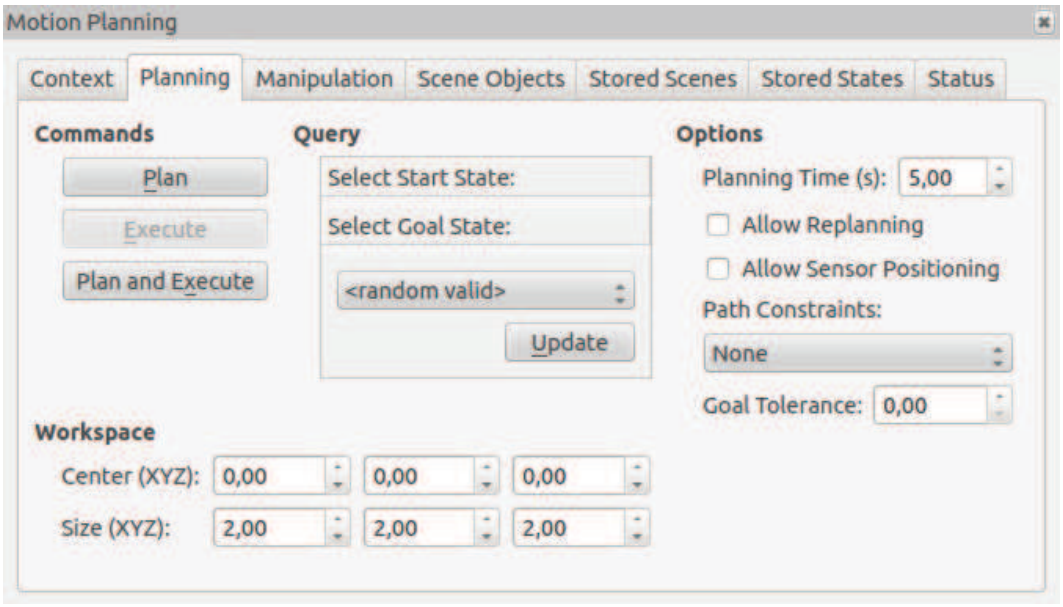

Figura 29. Panel de Motion Planning del simulador.

Y por último se muestra el panel donde se visualiza todo lo que se simula. En este vemos el modelo de nuestro robot situado en un mundo virtual. A parte de fijar las posiciones iniciales por la ventana *Planning*, anteriormente comentada, se ofrece también la posibilidad de hacerlo desde aquí. En este caso, si el grupo es sencillo (una única cadena de *links*), aparece al final del grupo unos marcadores que pueden utilizarse para desplazar el *end effector* por el espacio, y con ello arrastrar el grupo entero.

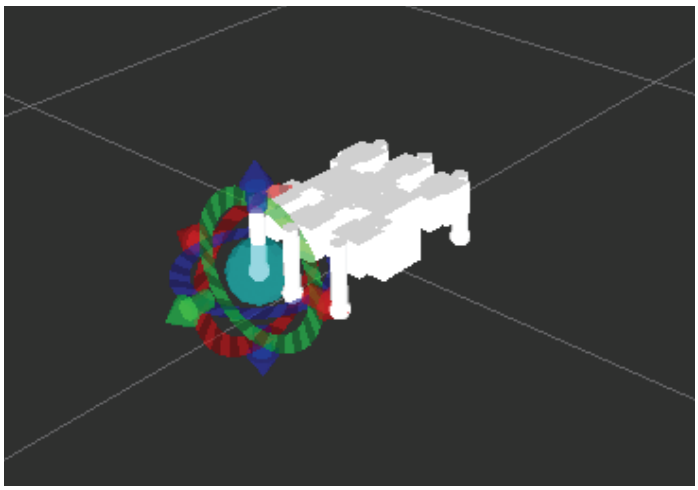

Figura 30. Panel de visualización del simulador.

En el presente trabajo, los *end effector* se establecen por defecto, en los pies de las patas. Así si en *Planning Request* se selecciona uno de los grupos de una pata, en esa misma nos aparecerá el marcador para poder desplazarla. Se podrá desplazar tanto la posición inicial como la final a la posición deseada. Si no hubiéramos generado los *links* imaginarios (*knee* y *feet*), este movimiento no sería posible, ya que al sistema le faltaría grados de libertad, debido a que al desplazar el *end effector*, éste se mantiene en su inclinación inicial y son los demás *links* los que pueden inclinarse libremente.

En la realidad el extremo último de la pata puede moverse por el espacio cambiando la inclinación de las tres piezas que la componen. El hecho de que el simulador mantenga la inclinación del *end effector* obliga a generar un nuevo *link (knee)* para que las tres partes puedan variar su inclinación . Pero al realizar esto surge otro problema, el *joint* que creemos entre el *link* imaginario y los demás debería tener rotación en dos ejes y esto no existe en las distintas tipologías disponibles para los *joints*. Esto obliga a crear un nuevo *link (feet)* que gire con respecto al otro imaginario (que ya gira con respecto a un eje sobre el último *link* real) y poder mover la pata como se mueve en la realidad. De ahí la razón de estos dos *links*.

He aquí un ejemplo de cómo manipular la posición final e inicial de una de las patas, y la resultante visualización:

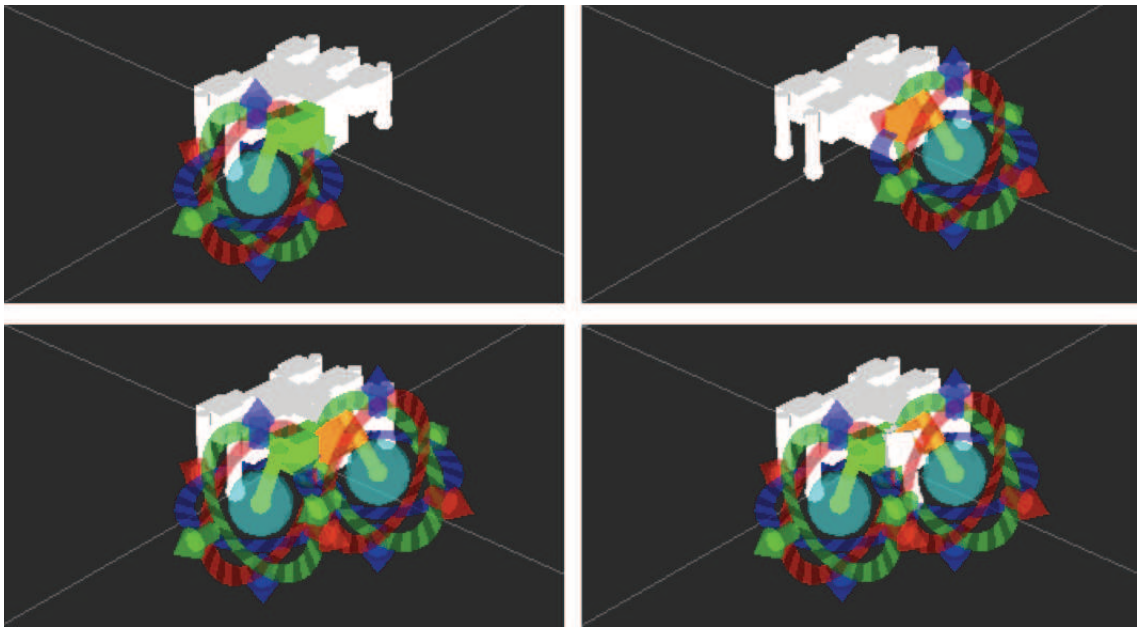

Figura 31. Ejemplo de asignación de posición inicial y final de un grupo y ejecución de trayectoria en el simulador.

Con todo esto podría hacerse ya una simulación de un movimiento, leer la solución de dicho movimiento y enviársela al robot. Una vez establecidas posiciones iniciales y finales, se planificaría y ejecutaría observando en pantalla al modelo realizar el desplazamiento definido. Pero aunque ya se ha avanzado en el proceso ya que ahora no se fija la posición con los grados de giro de los *joints* (y por lo tanto de los servos), sigue sin ser lo deseado. Aunque no se utilicen ángulos, las posiciones finales e iniciales se fijan sin ninguna exactitud, simplemente moviendo a simple

vista uno de los grupos. No se sabe a que punto del espacio se está llevando el *end effector*. Pero ya se puede empezar a vislumbrar el camino a seguir, la cuestión radica en utilizar alguna herramienta interna para enviar al simulador las coordenadas de la posición final (*goal state*) y luego ya planificar y ejecutar. Aquí es donde aparecen las herramientas de MoveIt!.

6.5. Control por Posiciones Cartesianas de la Posición del Robot

Como se ha visto, con el Rviz ya se ofrece un abanico de posibilidades muy amplio para manipular un robot, pero en caso de precisar más exactitud a la hora de definir posiciones y trayectorias, como el presente trabajo, se necesita algo más. MoveIt! ofrece una gran variedad de funciones y posibilidades internamente. *Move_group* es el nodo donde se lleva a cabo toda la planificación del movimiento. Este nodo proporciona una amplia gama de prestaciones a la hora de la planificación. Ejemplos de alguna de ellas son funciones que permiten obtener la posición en la que se encuentra el robot, ajustar los errores permitidos a la hora de alcanzar las posiciones indicadas, generar planes… Pero de todas ellas hay una que tiene especial interés, la que permite ajustar la *goal state* del *end effector*. Con esta función queda resuelto el problema, aquí se le podrá indicar a que posición cartesiana se desea que vaya y él hará los cálculos.

Para implementar y utilizar esta función, y otras de interés, se genera un nuevo paquete donde se almacenará un archivo de código denominado *move_control.cpp* que realizará la función de conectar con el simulador y enviar la posición a la que debe ir el robot. Por lo tanto, lo primero será crear dicho paquete, en este caso se le ha llamado *move_control*, y se establecen como dependencias *roscpp* y, muy importante, *moveit_ros_planning_interface*. Esta última es la que nos permite utilizar las funciones del *move_group*. Una vez creado el paquete se escribirá, dentro de la carpeta *src* del mismo el archivo de código, que para este caso es el siguiente:

```
#include <moveit/move_group_interface/move_group.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>

#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

int main(int argc,char**argv)
{
ros::init(argc, argv,"move_group_interface_tutorial");
  ros::NodeHandle node_handle;
ros::AsyncSpinner spinner(1);
spinner.start();

//Generamos una variable de tipo MoveGroup y la asignamos al grupo deseado.
moveit::planning_interface::MoveGroup hexapod("hexapod");

//Cambiamos el tiempo de planificación.
  hexapod.setPlanningTime(180.0);

//Obtenemos el nombre de la estructura de referencia.
ROS_INFO("Reference frame: %s", hexapod.getPlanningFrame().c_str());
```

```
//Ahora se define la secuencia de movimiento, en este caso avanzar, y consistirá en cuatro posiciones.
//Se define la primera posición de cada pata y se le asigna a su extremo.
geometry_msgs::Pose target_pose1;
  target_pose1.orientation.w =1.0;
  target_pose1.position.x =0.14;
  target_pose1.position.y =0.165;
  target_pose1.position.z =0.05;

hexapod.setPoseTarget(target_pose1,"frontal_right_feet");

  geometry_msgs::Pose target_pose2;
  target_pose2.orientation.w =1.0;
  target_pose2.position.x =0.04;
  target_pose2.position.y =-0.165;
  target_pose2.position.z =0.05;

hexapod.setPoseTarget(target_pose2,"central_left_feet");

  geometry_msgs::Pose target_pose3;
  target_pose3.orientation.w =1.0;
  target_pose3.position.x =-0.06;
  target_pose3.position.y =0.165;
  target_pose3.position.z =0.05;

hexapod.setPoseTarget(target_pose3,"ass_right_feet");

  geometry_msgs::Pose target_pose4;
  target_pose4.orientation.w =1.0;
  target_pose4.position.x =0.1;
  target_pose4.position.y =-0.165;
  target_pose4.position.z =-0.03;

hexapod.setPoseTarget(target_pose4,"frontal_left_feet");

  geometry_msgs::Pose target_pose5;
  target_pose5.orientation.w =1.0;
  target_pose5.position.x =0.0;
  target_pose5.position.y =0.165;
  target_pose5.position.z =-0.03;

hexapod.setPoseTarget(target_pose5,"central_right_feet");

  geometry_msgs::Pose target_pose6;
  target_pose6.orientation.w =1.0;
  target_pose6.position.x =-0.1;
  target_pose6.position.y =-0.165;
  target_pose6.position.z =-0.03;

hexapod.setPoseTarget(target_pose6,"ass_left_feet");

//Ahora planificamos el movimiento a esa posición y lo ejecutamos en el simulador.
moveit::planning_interface::MoveGroup::Plan hexapod_plan1;
bool success1 = hexapod.plan(hexapod_plan1);
```

```
sleep(4.0);

  ros::shutdown();
return0;
}
```

Como se puede ver, lo primero que se hace es inicializar un nodo. A continuación se toma el control del grupo a mover, en este caso *hexapod*. Es importante resaltar que el tiempo de planificación es un problema en esta parte del proyecto, por defecto éste es de 5 segundos pero se deberá ampliar a 180 segundos para poder obtener una solución valida. Posteriormente ya se generan variables del tipo de mensajes que se van a enviar con las posiciones, y se le dan valores a los parámetros de ese mensaje. Finalmente se fija como posición final de cada *end effector* la posición establecida, y se planea y ejecuta una trayectoria hasta dichos puntos.

En el ejemplo anterior se ha visto el código para ejecutar una única posición, el archivo completo se puede ver en el anexo al final del documento, donde sí se implementan las cuatro posiciones necesarias para que el robot camine.

Una vez escrito el código, deberá añadirse el ejecutable al *CMakeLists* del paquete y construir el mismo para que se compile.

Cuando ya se ha generado el ejecutable se lanzan los dos archivos que proporcionarán la solución al problema. Primero se lanza el *demo.launch* para que se abra el simulador y se cargue el modelo. A continuación se ejecuta el nodo *move_control* (de la manera que se ha explicado anteriormente con la herramienta *rosrun*). En el terminal donde se ha ejecutado el nodo veremos como se toma el control de un grupo (en este caso el *hexapod*) y empieza a buscar una solución de la posición final que se le ha indicado. Cuando resuelva la cinemática inversa y la encuentre pasará a la siguiente, así las cuatro veces que se ha definido. Este proceso puede tardar unos segundos incluso minutos debido a que la solución implica situar 6 *end effectors* en una posición.

Mientras se van encontrando las posiciones mandadas podemos ir viendo en el simulador como el hexápodo desplaza sus extremidades a esas posiciones. Por lo tanto el cálculo está realizado y el movimiento definido, solamente queda extraer esa solución y procesarla para enviársela a la tarjeta y que ésta haga que el robot se mueva al igual que la simulación.

6.6. Conexión con la Tarjeta Torobot y Envío de la Información

Aquí es donde aparece el *listener*, un nodo cuya función, como se puede ver en el nombre, es escuchar al MoveIt! y suscribirse al *topic* donde se estén publicando los mensajes que indiquen al simulador el giro de los servos para llegar a esa posición final. Este *topic* es donde se publican las soluciones de la cinemática una vez se han calculado.

Para ver de que *topic* se trata se utilizarán las herramientas de ROS. Con el siguiente comando se presentará en el terminal todos los *topics* que estén activos en ese momento.

```
rostopic list
```

A continuación se buscará donde se están publicando los datos deseados utilizando un comando que nos muestra que se publica dentro de cada *topic*.

```
rostopic echo <topic_name>
```

En este caso el *topic* que nos interesa es */move_group/display_planned_path*. Ahora que se sabe donde está la información se necesita saber enviarla, el tipo de mensaje que define ese *topic*. Para ello se utiliza:

```
rostopic type <topic_name>
```

Con esta herramienta se ve que el tipo de mensaje que se utiliza es *DisplayTrajectory*, pero si se busca este mensaje en los paquetes de instalación del MoveIt!, se puede comprobar que no es un único mensaje sino uno formado a partir de muchos otros, donde al final en uno de ellos se da la posición de los *joints*. Esto habrá que tenerlo en cuenta a la hora de coger las variables del mensaje cuando le llegue al *listener*.

En el momento en el que se conoce donde está la solución es cuando se necesita crear el nodo que la lea y haga las operaciones necesarias para transferirlo a la tarjeta. Por ello primero se creará un nuevo paquete, en este caso llamado *driver*, con *roscpp*, *std_msgs* y *trajectory_msgs* como dependencias. Está última es fundamental, ya que es el paquete donde se encuentra el tipo de mensaje del *topic* y se necesita para que el nodo que creado conozca y entienda ese mensaje.

Cuando ya se ha creado el paquete, en su carpeta *src* es donde se generará el código *listener.cpp*. El funcionamiento de éste consiste en una función principal donde se inicializa el nodo y se le declara como subscriptor del *topic* que se desea. Asociado a esto, se generará la función *hexapodCallback*, que será llamada cada vez que llegue un nuevo mensaje al *topic*. Desde el punto de vista que se ha enfocado el nodo *move_control*, cada vez que se calcula una de las posiciones se enviará un mensaje, por lo tanto se deberá esperar a que se reciban los cuatro mensajes para enviar la información a la tarjeta, mientras tanto se irá almacenando la información de las posiciones en vectores declarados como variables globales. Este recuento de los mensajes se llevará a cabo por una variable global también.

Lo primero que aparece en el código son las librerías donde no se debe olvidar poner el directorio donde se define el *.h* del tipo de mensaje. A continuación se definen como variables globales el contador de mensajes, la variable fichero en la que guardaremos la dirección USB y las cuatro variables destinadas a almacenar las cuatro posiciones (igualando la longitud a la cantidad de *joints* del modelo).

Una vez dentro de la función *hexapodCallback*, lo primero que aparece es la declaración de una variable para utilizar en los bucles como contador, y otras dos que se utilizarán para conocer las dimensiones de los vectores de datos (esto sirve para que el código sea genérico). Lo siguiente son parámetros de los servos y de la tarjeta, estos valores se utilizarán para obtener la recta que relacione ángulos de giro del servo con el valor analógico de la tarjeta.

El mensaje que se quiere leer contiene varias posiciones por las que se ha pasado a lo largo de la trayectoria, con la primera variable, se obtendrá de cuantos puntos se han calculado la posición de los servos, para así quedarse con el último, el punto final. A continuación se leerá de cuantos *joints* se compone el modelo.

Cuando se han obtenido estos valores se crean tres vectores muy importantes. El primero de ellos es el vector que se utilizará para almacenar el valor (en ángulos) de los *joints*. El segundo es el vector de calibración del cual se ha hablado al principio del documento. Y por último, el vector de signos, cuya función es invertir el sentido de los ángulos en aquellos servos que en el montaje se han puesto en sentido contrario al sistema que tiene el MoveIt! para medir los ángulos.

Ahora ya se disponen de todas las herramientas para llevar a cabo la funcionalidad del *listener*. Primero se entrará en un bucle donde se irán copiando los valores de los *joints* en la última posición de la trayectoria, en el vector *joints_angles*. A su vez se le cambiará el signo a aquellos datos que sea necesario. Este proceso de lectura de datos se deberá realizar las cuatro veces que llegue una posición, ya que cada vez los valores serán de posiciones distintas.

La siguiente parte esta dividida en cuatro bucles, uno para cada vez que llegue un mensaje. En todos los bucles se convierten los ángulos en los valores utilizados por la tarjeta y se les aplica la calibración. Los cuatro bucles corresponden a las cuatro posiciones, en el primer mensaje que llegue se entrará únicamente en el primero y se almacenarán los datos en la variable *servo_param1*, en el segundo mensaje se entrará en el segundo bucle y se almacenarán los valores en la variable *servo_param2*, así sucesivamente hasta tener listos los cuatro vectores con los valores a enviar a la *Torobot*.

Por último queda conectar con el USB del cual se deberá conocer en que puerto está conectado para acceder a él. Una vez establecida la dirección y asociada a la variable fichero, se entra en un bucle donde se enviarán cuatro telegramas consecutivos, las cuatro posiciones, hasta que se presione una tecla por el usuario para que el robot se detenga.

El código completo quedará de la siguiente forma:

```
#include "ros/ros.h"
#include "moveit_msgs/DisplayTrajectory.h"
#include <fstream>
#include <stdio.h>
#include <iostream>
#include <vector>
#include <string>

//Variable contador de mensajes.
int j=0;

//Variable para asignar el puerto.
  FILE* pFile;

//Variables para guardar las posiciones de los servos.
int servo_param1 [31],servo_param2 [31],servo_param3 [31],servo_param4 [31];

//Función Callback a la espera de que el simulador resuleva la cinemática inversa y
```

```
//para posterior envío de la información al USB.
void hexapodCallback(const moveit_msgs::DisplayTrajectory::ConstPtr& msg)
{
int length_point, length_position, i;

//Datos del servo para la conversión.
float max_servo=2500, min_servo=500, max_angle=3.1416, min_angle=-3.1416, constant;
constant=(max_servo)-(((max_servo-min_servo)/(max_angle-min_angle))*max_angle);

//Obtención de la longitud de los vectores.
length_point=msg->trajectory[0].joint_trajectory.points.size();
length_position=msg->trajectory[0].joint_trajectory.points[0].positions.size();

//Generación del vector de ángulos de los servos, el de calibrado, y el de cambio de signo.
float joints_angles [length_position];
int calibration []={25,50,-50,0,0,-75,-50,0,0,0,75,0,-50,0,0,75,-50,-75,0,0,40,-25,-50,0,0,-
25,125,100,0,0,0};
float signes []={1,1,-1,1,1,-1,-1,1,1,1,1,-1,1,1,1,1,-1,1,1,1,1,1,-1,1,1,1,1,-1,1,1};

//Almacenamiento de la solución de la simulación en el vector de ángulos y corrección del mismo.
for(i=0;i<=length_position-1;i++)
{
joints_angles[i]=msg->trajectory[0].joint_trajectory.points[(length_point-1)].positions[i];
joints_angles[i]=joints_angles[i]*signes[i];
}

//Conversión a valores del telegrama y almacenamiento en cada vector de posición dependiendo
//de la posición que acabamos de recibir, y posterior calibración.
if(j==0)
{
for(i=0;i<=length_position-1;i++)
{
servo_param1[i]=(joints_angles[i]*((max_servo-min_servo)/(max_angle-min_angle))+constant);
servo_param1[i]=servo_param1[i]+calibration[i];
}
}
if(j==1)
{
for(i=0;i<=length_position-1;i++)
{
servo_param2[i]=(joints_angles[i]*((max_servo-min_servo)/(max_angle-min_angle))+constant);
servo_param2[i]=servo_param2[i]+calibration[i];
}
}
if(j==2)
{
for(i=0;i<=length_position-1;i++)
{
servo_param3[i]=(joints_angles[i]*((max_servo-min_servo)/(max_angle-min_angle))+constant);
servo_param3[i]=servo_param3[i]+calibration[i];
}
}
if(j==3)
{
for(i=0;i<=length_position-1;i++)
```

```
{
servo_param4[i]=(joints_angles[i]*((max_servo-min_servo)/(max_angle-min_angle))+constant);
servo_param4[i]=servo_param4[i]+calibration[i];
}
}

//Conexión con el puerto USB y envío del telegrama de las cuatro posiciones.
if(j==3)
{
pFile=                                                    fopen("/dev/serial/by-id/usb-
TOROBOT.com_mini_USB_servo_controller_8D8F138C5557-if00","w");
if(pFile!=NULL)
{
    printf("Puerto abierto\n");
while(1)
{

fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param1[10],servo_para
m1[11],servo_param1[12],servo_param1[0],servo_param1[1],servo_param1[2],servo_param1[20],
servo_param1[21],servo_param1[22],servo_param1[25],servo_param1[26],servo_param1[27],serv
o_param1[5],servo_param1[6],servo_param1[7],servo_param1[15],servo_param1[16],servo_para
m1[17]);

    sleep(1.0);


fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param2[10],servo_para
m2[11],servo_param2[12],servo_param2[0],servo_param2[1],servo_param2[2],servo_param2[20],
servo_param2[21],servo_param2[22],servo_param2[25],servo_param2[26],servo_param2[27],serv
o_param2[5],servo_param2[6],servo_param2[7],servo_param2[15],servo_param2[16],servo_para
m2[17]);

    sleep(1.0);


fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param3[10],servo_para
m3[11],servo_param3[12],servo_param3[0],servo_param3[1],servo_param3[2],servo_param3[20],
servo_param3[21],servo_param3[22],servo_param3[25],servo_param3[26],servo_param3[27],serv
o_param3[5],servo_param3[6],servo_param3[7],servo_param3[15],servo_param3[16],servo_para
m3[17]);

    sleep(1.0);


fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param4[10],servo_para
m4[11],servo_param4[12],servo_param4[0],servo_param4[1],servo_param4[2],servo_param4[20],
servo_param4[21],servo_param4[22],servo_param4[25],servo_param4[26],servo_param4[27],serv
o_param4[5],servo_param4[6],servo_param4[7],servo_param4[15],servo_param4[16],servo_para
m4[17]);

    sleep(1.0);
```

```
}
    fclose (pFile);
    printf("Puerto cerrado\n");
}
else
{
    printf("No he podido abrir el puerto\n");
    exit(-1);
}
}
  j++;
}

//Función principal donde se genera el subscriptor.
int main(int argc,char**argv)
{

ros::init(argc, argv,"listener");

ros::NodeHandle n;

ros::Subscriber sub = n.subscribe("/move_group/display_planned_path",100, hexapodCallback);

ros::spin();

return0;
}
```

6.7. Resultados

Una vez implementado el código, simplemente quedará construir el paquete con el *catkin_make*. Ahora ya se está en disposición de controlar el robot. Para ello se deberá conectar el robot al ordenador, ejecutar el *listener* y el *demo.launch*, y una vez esté todo arrancado lanzar el *move_control*.

La secuencia será la anterior, el simulador buscará una solución para las posiciones y la ejecutará en el simulador. Pero la diferencia es que mediante el *listener*, cuando se envíen esas soluciones, este las adaptará a la tarjeta y las enviará como se ha visto, en ese instante el hexápodo empezará a moverse en el bucle, pasando sus patas por las cuatro posiciones constantemente. El hexápodo caminará tal y como se le ha programado.

7. SEGUNDA SOLUCIÓN ADOPTADA: CÁLCULO DE LA CINEMÁTICA INVERSA

7.1. Planteamiento Inicial

Hasta el momento, el principal objetivo del proyecto se ha conseguido: mover el hexápodo mediante la aplicación MoveIt!. Aunque el resultado ha sido el esperado, el robot camina, es interesante ver si se puede realizar la misma tarea de otra manera que a su vez aporte nuevas ventajas.

A partir de aquí, se partirá de nuevo de cero en el trabajo con la idea principal de eliminar la parte de simulación. MoveIt! es una herramienta muy amplia y compleja que puede no ser del todo adecuada para este procedimiento. Aunque se haya conseguido hacer caminar al robot, puede que haya enfoques más simples y adecuados.

Debido a que únicamente hay tres servos por pata, el estudio de las ecuaciones que permiten calcular la cinemática inversa se presenta factible y relativamente sencillo frente a otros robots donde esto no se puede considerar hacerlo de forma manual y menos implementarlo. Cuando se conozca el modelo matemático que resuelve el problema, se creará un nodo que sustituya al simulador. Así será el nodo el que nos aporte la solución y desde ese mismo se enviará al robot.

Esta sustitución puede hacer que el tiempo computacional se reduzca prácticamente al mínimo cuando con el simulador se llegaba a hablar de minutos (algo que no es viable). Además al hacerlo todo en un nodo se reducen los envíos de información de un lugar a otro y se puede aumentar su sencillez y por lo tanto compresión.

Con este cambio podemos conseguir mejorar pero no cambiar el proceso. Se resuelve la cinemática para unos puntos y se ejecuta una secuencia de movimiento. Al haber hecho la parte previa se ha visto que el robot queda alejado del usuario, no hay interacción. Hasta ahora, todo lo realizado no está al alcance de todos. Por lo tanto se buscará generar una herramienta que acerque el uso del hexápodo a cualquier persona sin la necesidad de unos conocimientos de programación previos. Como se plantea en la introducción, los robots están entrando en el día a día de las personas y para ello esas personas deben de saber controlarlos. Con esta aplicación se buscará que sin la necesidad de entender todo el contenido que se ha desarrollado en el diseño de este trabajo, una persona común sea capaz de aprovechar ese trabajo y utilizar el robot desde tareas científicas hasta actividades tan simples como servir como juguete de niños.

Para conseguir el objetivo anterior se ha elegido formar unos cursores donde el usuario pueda elegir a que altura quiere que se levanten las patas cuando caminan y que paso quiere que tenga cada una de ellas en una secuencia de movimiento. Por lo tanto el hexápodo no solo caminará, sino que además lo hará como el usuario desee, la tarea del programador residirá en que esa información se recolecte y se trate de forma adecuada.

7.2. Obtención del Modelo Matemático de la Cinemática Inversa

Una vez concebido el guión a seguir en esta segunda parte se empezará a construir lo que podría ser un segundo proyecto. Lo primero será estudiar la cinemática inversa y ver que ecuaciones resuelven el problema y permiten conocer el ángulo de giro de cada servo para que el extremo de la pata alcance una posición previamente determinada (x,y,z).

En la imagen siguiente se muestra un esquema de una de las patas (en planta a la izquierda y la representación del plano que la atraviesa longitudinalmente a la derecha) situando el eje de referencia y marcando los ángulos que serán útiles en el estudio. Cada servo está representado por puntos, excepto el último que representa el extremo que queremos mover. A los ángulos solución (los que representan el giro de los servos) los denominaremos q_1, q_2 y q_3, en orden de lejanía del centro del sistema y se medirán respecto a la posición inicial donde el *femur* y la *tibia* están en horizontal y el *radius* en vertical.

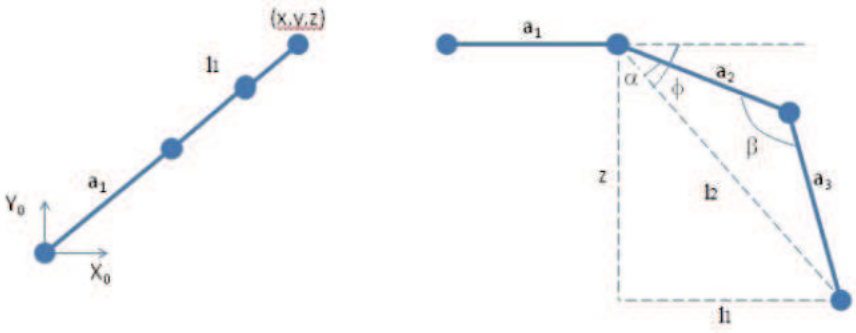

Figura 32. Esquema de las patas del hexápodo.

Obtener el primer ángulo es sencillo, únicamente se necesitan las coordenadas x e y. Esto se puede ver en la imagen de la izquierda. La ecuación resultaría tan simple como resolver una tangente:

$$q_1 = \tan^{-1}\left(\frac{y}{x}\right) \tag{1}$$

Para la obtención de los otros dos se necesitarán las medidas de las piezas (a_1 (*femur*), a_2 (*tibia*) y a_3 (*radius*)) además de dos distancias virtuales que se calculan a continuación:

$$l_1 = \sqrt{x^2 + y^2} - a_1 \tag{2}$$

$$l_2 = \sqrt{z^2 + l_1^2} \tag{3}$$

Una vez definidas esas medidas ya se tienen todos los valores necesarios para obtener los dos ángulos que faltan mediante la resolución de los triángulos. En este caso se utilizan los teoremas del coseno y posteriormente se suman y restan los ángulos obtenidos hasta obtener el deseado:

$$\alpha = \cos^{-1}\left(\frac{a_2^2 + l_2^2 - a_3^2}{2a_2 l_2}\right) \tag{4}$$

$$\beta = \cos^{-1}\left(\frac{a_2^2 + a_3^2 - l_2^2}{2a_2 a_3}\right) \tag{5}$$

$$\emptyset = \tan^{-1}\left(\frac{l_1}{-z}\right) \tag{6}$$

$$q_2 = \alpha + \emptyset - \frac{\pi}{2} \tag{7}$$

$$q_3 = \beta - \frac{\pi}{2} \tag{8}$$

Aquí ya se ve como la matemática que hay detrás del movimiento del hexápodo es muy sencilla lo que hará que la creación del nodo sea fácil. Para algo breve como esto (como ya se había supuesto), el simulador puede ser contraproducente, siendo más rápido realizarlo a mano.

La implementación de este modelo se realizará en un nodo denominado *server*. Pero como hemos mencionado, queremos compactar todas las tareas en este único nodo por lo que antes de crearlo hay que preparar las demás partes que tienen que darle información, como lo son los cursores dinámicos que permitirán manipular la altura y el paso. Estos cursores serán la entrada de las coordenadas de las ecuaciones matemáticas vistas.

7.3. Configuración de los Controladores para la Interacción Usuario-Robot

Para ello lo primero será crear el paquete donde vamos a trabajar. Se le llamará *dynamic_control*, y se establecerán como dependencias *roscpp*, *rospy*, *std_msgs* y *dynamic_reconfigure*. Ésta última es el paquete que nos permitirá generar los cursores en un archivo *.cfg*.

Ahora que ya hemos creado el paquete vamos a configurar los cursores. Para una mayor organización se creará un directorio dentro del paquete denominado *cfg* donde se situará el archivo. La idea de esta aplicación es que en pantalla aparezcan unos cursores donde se nos permita cambiar sus valores, para ello la herramienta *dynamic_reconfigure* es muy útil. Esta te permite crear distintos tipos de controles: listados, cursores de distintos tipos, variables booleanas, …

En este caso lo deseado son dos cursores de variables reales, que irán entre unos limites definidos por la estructura del robot, ya que no se puede permitir un paso donde las patas choquen entre sí, ni una altura donde las patas no lleguen.

Por lo tanto lo que se hará es crear un documento *.cfg* (denominado en este caso *Control.cfg*) donde primero se inicialice ROS y los parámetros generadores y luego ya se definen los controladores. Para ello se utiliza la siguiente función:

```
gen.add("<name>", <type>, <level>, "<description>", "<default>", "<min>", "<max>")
```

A la hora de elegir el tipo, la tipología podrá ser *int_t* (números enteros), *double_t* (números reales), *str_t* (cadena de caracteres) y *bool_t* (variable booleana). El nivel, previamente establecido como 0, es lo que le indica al sistema si el valor de dicha variable se ha modificado. Cuando el valor del nivel pase de 0 a 1 significará que la variable ha sufrido un cambio y se enviará un mensaje con los nuevos valores, a los nodos que estén suscritos a estos cursores activándose la función *callback* (vista anteriormente). Los últimos tres parámetros definen el valor por defecto del controlador y sus valores máximos y mínimos.

En el trabajo se han generado dos cursores únicamente, uno para el paso y otro para la altura. El archivo *.cfg* creado se puede ver a continuación:

```
#!/usr/bin/env python
PACKAGE = "dynamic_control"

from dynamic_reconfigure.parameter_generator_catkin import *
```

```
gen = ParameterGenerator()

gen.add("paso", double_t, 0, "El paso que avanza cada pata con una secuencia de movimiento", 0.07, 0, 0.08)
gen.add("altura", double_t, 0, "La altura a la que se eleva cada pata", -0.15, -0.15, -0.05 )

exit(gen.generate(PACKAGE, "dynamic_control", "Control"))
```

Una vez creado el archivo se le debe dotar de uso como ejecutable mediante el siguiente comando:

```
chmod a+x <cfg/relative_path_to_file1.cfg>
```

Por último se deberá añadir en el CMakeLists del paquete unas líneas (tienen que estar antes que el *catkin_package()* para que funcione correctamente) para construir la aplicación y que se generen las librerías que usaremos en el nodo. Las líneas comentadas son las siguientes:

```
generate_dynamic_reconfigure_options(
cfg/realtive_path_to_file1.cfg
cfg/relative_path_to_file2.cfg
…
)
```

Así se adjuntarán cada uno de los archivos que se hayan creado, en este caso *Controls.cfg*.

7.4. Implementación del Cálculo de la Cinematica Inversa y Envío de la Información

Una vez hecho todo esto ya podemos crear el nodo *server*. En este nodo lo que queremos es que se llame a una función *callback* cada vez que el *level* de alguno de los controladores cambie, y dentro de esa función realizar las operaciones pertinentes.

Lo primero son las librerías donde se incluirán algunas como *math.h* para las operaciones trigonométricas, y dos muy importantes *server.h* y *ControlConfig.h*. La primera de ellas permitirá llamar a la función *callback* mientras que la segunda es donde se almacena la información del tipo de controles que se han generado, para que el nodo pueda entenderlo. Similar a cuando se necesita un mensaje y se debe incluir al principio del código.

A continuación se ve la función *callback*. Lo primero es la generación de las variables tanto las necesarias para transformar de ángulos a valores de la tarjeta, como las utilizadas en la resolución del problema cinemático. Una vez definidas llega un paso importante: fijar los tres puntos que configuran la trayectoria de las patas. En este caso algunos de ellos se obtienen a partir de los datos que se pasen por los controladores (como son la altura o el paso medio) y es aquí donde el usuario interviene. Una vez se han definido los tres puntos se procede a calcular la cinemática inversa de cada uno de ellos con el modelo matemático obtenido.

Cuando ya se ha obtenido la solución para la trayectoria de una pata se le asigna a las demás una trayectoria equivalente con los cambios de signo necesarios. Y finalmente se realiza la misma operación que en el código del *listener* anterior, se generan tres posiciones y se le envía a la tarjeta. En este caso mientras tres de esas patas se mueven a lo largo de las tres posiciones, las otras tres se deben de quedar en la posición inicial. Esto se conseguirá cuando al enviar el telegrama, la variable que se use para cada servo donde se ha almacenado la posición, no sea la misma. En los que estén en reposo será *servo_param_3* y los demás la que toque.

Por último está la función principal donde la única tarea es inicializar el nodo y hacer que se llame a la función *callback* cada vez que haya un cambio en los cursores.

El código completo es el presente a continuación:

```
#include "ros/ros.h"
#include <dynamic_reconfigure/server.h>
#include <dynamic_control/ControlConfig.h>
#include <fstream>
#include <stdio.h>
#include <iostream>
#include <vector>
#include <string>
#include <math.h>

//Función callback donde se realiza el procesado y envío de datos.
void callback(dynamic_control::ControlConfig &config, uint32_t level)
{

//Definición de la variable fichero, el contador para bucles y variables para la recta de ángulos a valores de la tarjeta.
FILE* pFile;
int i;
float max_servo=2500, min_servo=500, max_angle=3.1416, min_angle=-3.1416, constant;
constant=(max_servo)-(((max_servo-min_servo)/(max_angle-min_angle))*max_angle);

//Definición de las variables posición y los ángulos del cálculo cinemático.
float x[3],y[3],z[3],serv_1[3],serv_2[3],serv_3[3],long_1[3],long_2[3],alpha[3],beta[3],phi[3];

//Definición de las variables del procesado de datos (similares a las del listener).
float angles_1 [18],angles_2 [18],angles_3 [18];
int servo_param_1[18],servo_param_2[18],servo_param_3[18];
float signes []={1,1,-1,1,-1,1,1,1,-1,1,-1,1,1,1,-1,1,1,-1};
int calibration []={75,0,-50,25,50,-50,40,-25,-50,-25,125,100,-75,-50,0,75,-50,-75};

//Asignación de valores a las tres posiciones de la trayectoria cogiendo algunos de los controladores.
   x[0]=0.15;
y[0]=config.paso/2;
   z[0]=config.altura;
   x[1]=0.15;
   y[1]=config.paso;
   z[1]=-0.15;
x[2]=0.15;
   y[2]=0.0;
```

```
   z[2]=-0.15;

//Cálculo de la cinemática inversa
for(i=0;i<3;i++)
{
serv_1[i]=atan(y[i]/x[i]);
long_1[i]=sqrt(pow(x[i],2)+pow(y[i],2))-0.05;
    long_2[i]=sqrt(pow(z[i],2)+pow(long_1[i],2));
    alpha[i]=acos((pow(0.1,2)+pow(long_2[i],2)-pow(0.12,2))/(2*0.1*long_2[i]));
    beta[i]=acos((pow(0.1,2)+pow(0.12,2)-pow(long_2[i],2))/(2*0.1*0.12));
phi[i]=atan(long_1[i]/(-z[i]));
serv_2[i]=-3.1416/2+phi[i]+alpha[i];
serv_3[i]=beta[i]-3.1416/2;
}

//Bucles para asignar a cada pata la cinemática inversa de cada posición.
for(i=0;i<9;i=i+3)
{
angles_1[i]=serv_1[0];
angles_1[i+1]=serv_2[0];
angles_1[i+2]=serv_3[0];
}
for(i=0;i<9;i=i+3)
{
angles_2[i]=serv_1[1];
angles_2[i+1]=serv_2[1];
angles_2[i+2]=serv_3[1];
}
for(i=0;i<9;i=i+3)
{
angles_3[i]=serv_1[2];
angles_3[i+1]=serv_2[2];
angles_3[i+2]=serv_3[2];
}
for(i=9;i<18;i=i+3)
{
angles_1[i]=-serv_1[0];
angles_1[i+1]=-serv_2[0];
angles_1[i+2]=-serv_3[0];
}
for(i=9;i<18;i=i+3)
{
angles_2[i]=-serv_1[1];
angles_2[i+1]=-serv_2[1];
angles_2[i+2]=-serv_3[1];
}
for(i=9;i<18;i=i+3)
{
angles_3[i]=-serv_1[2];
angles_3[i+1]=-serv_2[2];
angles_3[i+2]=-serv_3[2];
}

//Obtención de los valores a enviar a la tarjeta
for(i=0;i<18;i++)
```

```
{
servo_param_1[i]=(angles_1[i]*signes[i]*((max_servo-min_servo)/(max_angle-
min_angle))+constant);
servo_param_2[i]=(angles_2[i]*signes[i]*((max_servo-min_servo)/(max_angle-
min_angle))+constant);
servo_param_3[i]=(angles_3[i]*signes[i]*((max_servo-min_servo)/(max_angle-
min_angle))+constant);
}
for(i=0;i<18;i++)
{
servo_param_1[i]=servo_param_1[i]+calibration[i];
servo_param_2[i]=servo_param_2[i]+calibration[i];
servo_param_3[i]=servo_param_3[i]+calibration[i];
}

//Conexión con la tarjeta y envío del telegrama.
pFile=                                                    fopen("/dev/serial/by-id/usb-
TOROBOT.com_mini_USB_servo_controller_8D8F138C5557-if00","w");
if(pFile!=NULL)
{
    printf("Puerto abierto\n");
for(i=0;i<8;i++)
{

fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param_1[0],servo_para
m_1[1],servo_param_1[2],servo_param_3[3],servo_param_3[4],servo_param_3[5],servo_param_1
[6],servo_param_1[7],servo_param_1[8],servo_param_3[9],servo_param_3[10],servo_param_3[11
],servo_param_1[12],servo_param_1[13],servo_param_1[14],servo_param_3[15],servo_param_3[
16],servo_param_3[17]);
     sleep(1);


fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param_2[0],servo_para
m_2[1],servo_param_2[2],servo_param_3[3],servo_param_3[4],servo_param_3[5],servo_param_2
[6],servo_param_2[7],servo_param_2[8],servo_param_3[9],servo_param_3[10],servo_param_3[11
],servo_param_2[12],servo_param_2[13],servo_param_2[14],servo_param_3[15],servo_param_3[
16],servo_param_3[17]);
     sleep(1);


fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param_3[0],servo_para
m_3[1],servo_param_3[2],servo_param_1[3],servo_param_1[4],servo_param_1[5],servo_param_3
[6],servo_param_3[7],servo_param_3[8],servo_param_1[9],servo_param_1[10],servo_param_1[11
],servo_param_3[12],servo_param_3[13],servo_param_3[14],servo_param_1[15],servo_param_1[
16],servo_param_1[17]);
     sleep(1);


fprintf(pFile,"#1P%d#2P%d#3P%d#6P%d#7P%d#8P%d#13P%d#14P%d#15P%d#18P%d#19P%d
#20P%d#25P%d#26P%d#27P%d#30P%d#31P%d#32P%dT100\r\n",servo_param_3[0],servo_para
m_3[1],servo_param_3[2],servo_param_2[3],servo_param_2[4],servo_param_2[5],servo_param_3
[6],servo_param_3[7],servo_param_3[8],servo_param_2[9],servo_param_2[10],servo_param_2[11
```

```
],servo_param_3[12],servo_param_3[13],servo_param_3[14],servo_param_2[15],servo_param_2[
16],servo_param_2[17]);
      sleep(1);

}
     fclose (pFile);
     printf("Puerto cerrado\n");
}
else
{
     printf("No he podido abrir el puerto\n");
exit(-1);
}
}

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

ros::init(argc, argv,"server");

   dynamic_reconfigure::Server<dynamic_control::ControlConfig> server;
dynamic_reconfigure::Server<dynamic_control::ControlConfig>::CallbackType f;

f= boost::bind(&callback, _1, _2);
server.setCallback(f);

   ROS_INFO("Spinning node");
ros::spin();
return0;
}
```

Ahora que ya está escrito el código queda incluirlo en el *CMakeLists* para que se cree el ejecutable al construir el paquete, y proceder a dicha construcción con el *catkin_make*.

Cuando ya se ha construido todo correctamente ya se puede ejecutar el nodo *server*. Si se lanza se verá que al robot le llega una señal y este empieza a caminar, se ha conseguido el objetivo, el hexápodo avanza.

7.5. Uso de los Controladores

Por defecto en la primera iteración se le pasan los valores *default* de los controladores. Para utilizar los cursores lo que se debe hacer es ejecutar el siguiente comando para abrir la aplicación:

```
rosrun rqt_reconfigure rqt_reconfigure
```

Aparecerá una ventana donde se podrá elegir el nodo del cual se quieren ver los controles, en este caso se escogerá el nodo *server*. Ahora se podrá ir variando el valor de los cursores y por lo tanto la trayectoria de las patas en la secuencia de movimiento. Al cambiarlos se podrá apreciar como el hexápodo va cambiando la manera de caminar en la realidad de la misma forma que nosotros le indicamos en el ordenador.

Figura 33. Ventana de los controladores en su posición inicial.

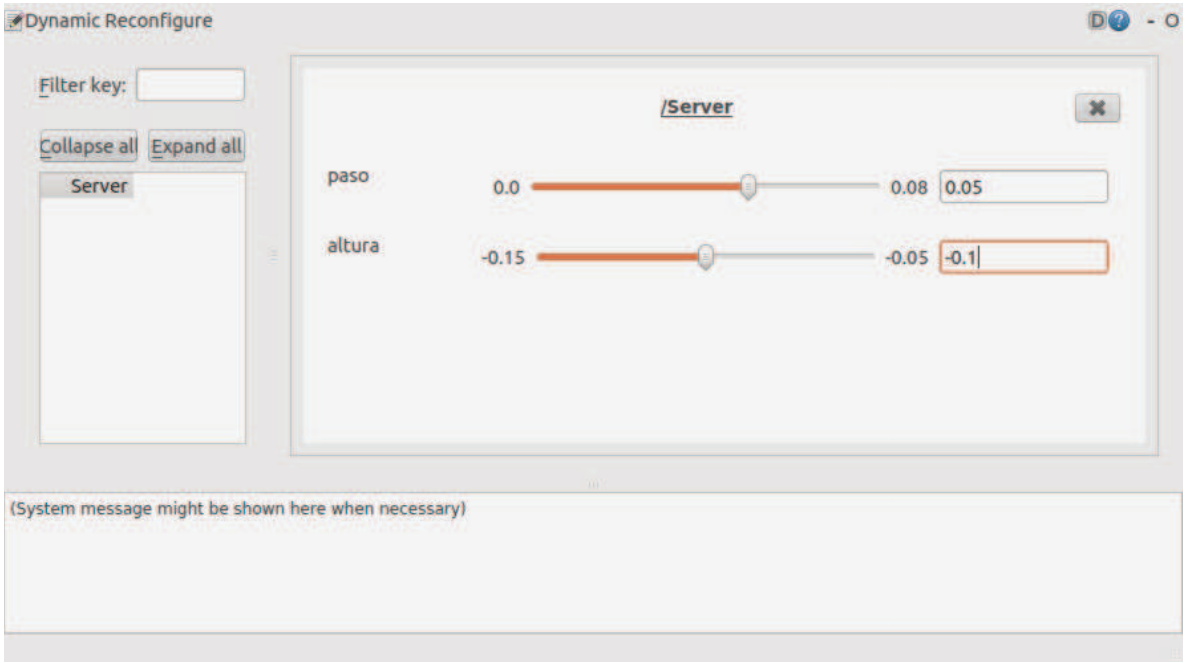

Figura 34. Ventana de los controladores en una posición manipulada por el usuario.

7.6. Resultados

El proceso está cerrado, se ha conseguido generar una aplicación que no solo permite que el hexápodo camine, sino que además ofrece la posibilidad de que cualquier persona le indique como hacerlo.

8. SOLUCIONES ALTERNATIVAS

A parte de las dos soluciones vistas para el problema planteado existen otras posibles soluciones que podrían dar resultados igualmente válidos.

Una de ellas sería mediante la utilización del simulador Gazebo. En este caso la solución es muy similar pero se utilizaría otra aplicación para llevar a cabo la configuración de los paquetes.

Como alternativa más similar a la segunda solución adoptada, pero sustituyendo ROS por Arduino, se podría resolver la cinemática inversa y generarse los telegramas en un código que se almacenaría en una tarjeta de Arduino. Todo esto para posteriormente ejecutarlo y a través de uno de los puertos de la tarjeta enviar los telegramas a la tarjeta Torobot. Además, para añadirle una parte de control externo se podrían conectar cuatro pulsadores a los pines de la tarjeta Arduino de modo que dependiendo del pulsador que se accione se active uno de esos pines y el código haga una secuencia diferente. Así por ejemplo, podríamos asociar cada uno de los pulsadores al movimiento en las cuatro direcciones principales, y dependiendo de que pulsador se active se resuelva la cinemática para posiciones que hacen que el hexápodo avance en una dirección de las cuatro.

9. CONCLUSIÓN

El propósito del proyecto era conseguir diseñar una programación para dotar de movimiento al hexápodo. Este objetivo se ha conseguido, mediante el control de la herramienta ROS, de dos formas completamente distintas. A continuación se concluirá el proyecto realizando una comparación entre ambas.

La primera de ellas utilizando la aplicación MoveIt! para simular, en un modelo virtual diseñado, el movimiento del robot y una vez simulado enviarlo a través de un puerto USB.

La segunda se ha resuelto utilizando la cinemática inversa manualmente y se a implementado directamente en un nodo al cual se le ha unido una aplicación que permite a cualquier usuario su interacción con el proceso. Así desde unos cursores se puede elegir la altura a la que se quiere que lleguen las patas al caminar, y el paso al que se desee que estas avancen. Este aspecto es importante ya que acerca el proceso al usuario medio.

Una vez vistas y desarrolladas las dos posibilidades cabe resaltar las ventajas e inconvenientes encontradas en cada caso.

El principal problema de la primera opción es el tiempo computacional. MoveIt! es una herramienta muy potente pero que tras la realización del trabajo se ha visto que es demasiado compleja para un proceso simple.

La aplicación está diseñada para poder generar escenarios, evitar obstáculos o introducir trayectorias muy complejas que sería imposible resolver a mano. Además ofrece una interacción continua con el robot mediante los sensores que este pueda tener generando mapas virtuales de lo que el robot “ve” y pudiendo interactuar con el entorno. En este trabajo el robot carece de sensores, y el movimiento que se le programa es independiente del entorno (se considera que se le ofrece un entorno propicio), por ello la segunda opción es más asequible y sencilla. El proyecto presentado tiene como objetivo que el robot camine hacia delante sin importar la presencia de obstáculos y por

tanto el problema matemático es breve. Es en consecuencia más operativo que exista un nodo que resuelva la cinemática inversa antes que calcularlo mediante un simulador que tiene en cuenta muchas variables y cuyo algoritmo es genérico y por lo tanto más complejo.

A parte de esto, MoveIt! está diseñado para brazos robóticos o manipuladores, que como mucho presentan dos extremos. En el trabajo presentado se le ha introducido un modelo de seis patas. MoveIt! resuelve la cinemática colocando un extremo en una posición y luego el siguiente partiendo del primero, si uno falla vuelve a empezar lo que para seis extremos puede ser muy largo.

Todas estas razones hacen que si se realiza el control del hexápodo con MoveIt! se tarde unos minutos en encontrar la solución, algo que con el nodo es casi instantáneo. Por lo tanto si la tarea es concreta e independiente de agentes externos, la resolución mediante un único nodo donde se desarrolle todo el proceso es más adecuado. Sin embargo, queda claro que en el momento en el que se desee que el robot esquive obstáculos mientras camine, o realice trayectorias que dependan de su entorno se deberá utilizar el simulador, aunque los tiempos de cálculo sean extremadamente largos.

Es importante resaltar que con ambos métodos se ha conseguido el objetivo inicial propuesto. Con MoveIt! se ha alcanzado una aplicación útil para el propósito, pero poco eficiente, además de resultar más costoso. Sin embargo, el trabajo realizado es muy útil para posteriores escalados más complejos del proceso. Lo que es el diseño de los componentes para utilizar MoveIt!, tanto para procesos sencillos como complejos, está prácticamente generado.

Por otra parte la segunda opción tratada, es más sencilla y limitada. Para realizar otras tareas se debería cambiar el código, cosa que podría resultar muy costosa dependiendo de la tarea. Sin embargo para este trabajo podemos concluir que sería el método más adecuado. Además la introducción de los controladores amplían el público al que va destinado, al fin y al cabo el propósito principal de una parte de la robótica en la industria es hacer robots (siempre complejos y difíciles de crear) cercanos a las personas y fáciles de utilizar.

10. BIBLIOGRAFÍA

[1] John McKerrow, Phillip. (1991). *Introduction to Robotics*.Malaysia.

[2]Torres, Fernando y otros. (2002). *Robots y Sistemas Sensoriales*.Madrid.

[3]Barrientos, Antonio. (1997). *Fundamentos de Robótica*.Madrid.

[4]Rentería, Arantxa. (2000). *Robótica Industrial*.Madrid.

[5] Angulo Usategui, José Mª. (2005). *Introducción a la Robótica*. Madrid.

[6] Reyes Cortés, Fernando. (2011). *Robótica: Control de Manipuladores*. Madrid.

[7]*http://www.heliumfrog.com/hf08robot/hf08blog.html*

[8]*http://www.ros.org*

[9]*http://wiki.ros.org/hydro/Installation*

MEMORIA:

ANEJO

CÓDIGO COMPLETO DEL MODELO DEL ROBOT

```
<robotname="hexapod">

<linkname="body_central">
        <visual>
        <originxyz="0 0 0.065"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.2 0.1 0.13"/>
        </geometry>
        <materialname="white">
        <colorrgba="1 1 1 1"/>
        </material>
        </visual>
</link>

<linkname="body_auxiliar">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.08 0.13 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_structure"type="fixed">
<originxyz="0 0 0.0975"/>
<parentlink="body_central"/>
<childlink="body_auxiliar"/>
</joint>


<linkname="frontal_right_femur">
        <visual>
        <originxyz="0 0.0175 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.05 0.035 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_to_frontal_right"type="revolute">
        <axisxyz="0 0 1"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0.1 0.05 0.0975"/>
        <parentlink="body_central"/>
        <childlink="frontal_right_femur"/>
</joint>
```

```
<linkname="frontal_left_femur">
        <visual>
        <originxyz="0 -0.0175 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.05 0.035 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_to_frontal_left"type="revolute">
        <axisxyz="0 0 1"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0.1 -0.05 0.0975"/>
        <parentlink="body_central"/>
        <childlink="frontal_left_femur"/>
</joint>

<linkname="central_right_femur">
        <visual>
        <originxyz="0 0.0175 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.05 0.035 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_to_central_right"type="revolute">
        <axisxyz="0 0 1"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.065 0"/>
        <parentlink="body_auxiliar"/>
        <childlink="central_right_femur"/>
</joint>

<linkname="central_left_femur">
        <visual>
        <originxyz="0 -0.0175 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.05 0.035 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_to_central_left"type="revolute">
        <axisxyz="0 0 1"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 -0.065 0"/>
        <parentlink="body_auxiliar"/>
        <childlink="central_left_femur"/>
```

```
</joint>

<linkname="ass_right_femur">
        <visual>
        <originxyz="0 0.0175 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.05 0.035 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_to_ass_right"type="revolute">
        <axisxyz="0 0 1"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="-0.1 0.05 0.0975"/>
        <parentlink="body_central"/>
        <childlink="ass_right_femur"/>
</joint>

<linkname="ass_left_femur">
        <visual>
        <originxyz="0 -0.0175 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.05 0.035 0.065"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="body_to_ass_left"type="revolute">
        <axisxyz="0 0 1"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="-0.1 -0.05 0.0975"/>
        <parentlink="body_central"/>
        <childlink="ass_left_femur"/>
</joint>

<linkname="frontal_right_tibia">
        <visual>
        <originxyz="0 0.0325 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.07 0.065 0.065"rpy="0 0 0"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_right_femur_to_tibia"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.035 0"/>
```

```
        <parentlink="frontal_right_femur"/>
        <childlink="frontal_right_tibia"/>
</joint>

<linkname="frontal_left_tibia">
        <visual>
        <originxyz="0 -0.0325 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.07 0.065 0.065"rpy="0 0 0"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_left_femur_to_tibia"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 -0.035 0"/>
        <parentlink="frontal_left_femur"/>
        <childlink="frontal_left_tibia"/>
</joint>

<linkname="central_right_tibia">
        <visual>
        <originxyz="0 0.0325 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.07 0.065 0.065"rpy="0 0 0"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="central_right_femur_to_tibia"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.035 0"/>
        <parentlink="central_right_femur"/>
        <childlink="central_right_tibia"/>
</joint>

<linkname="central_left_tibia">
        <visual>
        <originxyz="0 -0.0325 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.07 0.065 0.065"rpy="0 0 0"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="central_left_femur_to_tibia"type="revolute">
        <axisxyz="1 0 0"/>
```

```
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 -0.035 0"/>
        <parentlink="central_left_femur"/>
        <childlink="central_left_tibia"/>
</joint>

<linkname="ass_right_tibia">
        <visual>
        <originxyz="0 0.0325 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.07 0.065 0.065"rpy="0 0 0"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="ass_right_femur_to_tibia"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.035 0"/>
        <parentlink="ass_right_femur"/>
        <childlink="ass_right_tibia"/>
</joint>

<linkname="ass_left_tibia">
        <visual>
        <originxyz="0 -0.0325 0"rpy="0 0 0"/>
        <geometry>
        <boxsize="0.07 0.065 0.065"rpy="0 0 0"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="ass_left_femur_to_tibia"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 -0.035 0"/>
        <parentlink="ass_left_femur"/>
        <childlink="ass_left_tibia"/>
</joint>

<linkname="frontal_right_radius">
        <visual>
        <originxyz="0 0.015 -0.0325"rpy="0 0 0"/>
        <geometry>
        <cylinderlength="0.13"radius="0.015"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>
```

```
<jointname="frontal_right_tibia_to_radius"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.065 0"/>
        <parentlink="frontal_right_tibia"/>
        <childlink="frontal_right_radius"/>
</joint>

<linkname="frontal_left_radius">
        <visual>
        <originxyz="0 -0.015 -0.0325"rpy="0 0 0"/>
        <geometry>
        <cylinderlength="0.13"radius="0.015"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_left_tibia_to_radius"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 -0.065 0"/>
        <parentlink="frontal_left_tibia"/>
        <childlink="frontal_left_radius"/>
</joint>

<linkname="central_right_radius">
        <visual>
        <originxyz="0 0.015 -0.0325"rpy="0 0 0"/>
        <geometry>
        <cylinderlength="0.13"radius="0.015"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="central_right_tibia_to_radius"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.065 0"/>
        <parentlink="central_right_tibia"/>
        <childlink="central_right_radius"/>
</joint>

<linkname="central_left_radius">
        <visual>
        <originxyz="0 -0.015 -0.0325"rpy="0 0 0"/>
        <geometry>
        <cylinderlength="0.13"radius="0.015"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
```

```
</link>

<jointname="central_left_tibia_to_radius"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 -0.065 0"/>
        <parentlink="central_left_tibia"/>
        <childlink="central_left_radius"/>
</joint>

<linkname="ass_right_radius">
        <visual>
        <originxyz="0 0.015 -0.0325"rpy="0 0 0"/>
        <geometry>
        <cylinderlength="0.13"radius="0.015"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="ass_right_tibia_to_radius"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 0.065 0"/>
        <parentlink="ass_right_tibia"/>
        <childlink="ass_right_radius"/>
</joint>

<linkname="ass_left_radius">
        <visual>
        <originxyz="0 -0.015 -0.0325"rpy="0 0 0"/>
        <geometry>
        <cylinderlength="0.13"radius="0.015"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="ass_left_tibia_to_radius"type="revolute">
        <axisxyz="1 0 0"/>
        <limiteffort="1000.0"lower="-1.57"upper="1.57"velocity="0.5"/>
        <originxyz="0 -0.065 0"/>
        <parentlink="ass_left_tibia"/>
        <childlink="ass_left_radius"/>
</joint>

<linkname="frontal_right_knee">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
```

```
        </material>
        </visual>
</link>

<jointname="frontal_right_radius_to_knee"type="continuous">
        <axisxyz="1 0 0"/>
        <originxyz="0 0.015 -0.0975"/>
        <parentlink="frontal_right_radius"/>
        <childlink="frontal_right_knee"/>
</joint>

<linkname="frontal_right_feet">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_right_knee_to_feet"type="continuous">
        <axisxyz="0 0 1"/>
        <originxyz="0 0 0"/>
        <parentlink="frontal_right_knee"/>
        <childlink="frontal_right_feet"/>
</joint>

<linkname="frontal_left_knee">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="frontal_left_radius_to_knee"type="continuous">
        <axisxyz="1 0 0"/>
        <originxyz="0 -0.015 -0.0975"/>
        <parentlink="frontal_left_radius"/>
        <childlink="frontal_left_knee"/>
</joint>

<linkname="frontal_left_feet">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
```

```
        </visual>
</link>

<jointname="frontal_left_knee_to_feet"type="continuous">
        <axisxyz="0 0 1"/>
        <originxyz="0 0 0"/>
        <parentlink="frontal_left_knee"/>
        <childlink="frontal_left_feet"/>
</joint>

<linkname="central_right_knee">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="central_right_radius_to_knee"type="continuous">
        <axisxyz="1 0 0"/>
        <originxyz="0 0.015 -0.0975"/>
        <parentlink="central_right_radius"/>
        <childlink="central_right_knee"/>
</joint>

<linkname="central_right_feet">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="central_right_knee_to_feet"type="continuous">
        <axisxyz="0 0 1"/>
        <originxyz="0 0 0"/>
        <parentlink="central_right_knee"/>
        <childlink="central_right_feet"/>
</joint>

<linkname="central_left_knee">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
```

```
</link>

<jointname="central_left_radius_to_knee"type="continuous">
        <axisxyz="1 0 0"/>
        <originxyz="0 -0.015 -0.0975"/>
        <parentlink="central_left_radius"/>
        <childlink="central_left_knee"/>
</joint>

<linkname="central_left_feet">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="central_left_knee_to_feet"type="continuous">
        <axisxyz="0 0 1"/>
        <originxyz="0 0 0"/>
        <parentlink="central_left_knee"/>
        <childlink="central_left_feet"/>
</joint>

<linkname="ass_right_knee">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="ass_right_radius_to_knee"type="continuous">
        <axisxyz="1 0 0"/>
        <originxyz="0 0.015 -0.0975"/>
        <parentlink="ass_right_radius"/>
        <childlink="ass_right_knee"/>
</joint>

<linkname="ass_right_feet">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>
```

```
<jointname="ass_right_knee_to_feet"type="continuous">
        <axisxyz="0 0 1"/>
        <originxyz="0 0 0"/>
        <parentlink="ass_right_knee"/>
        <childlink="ass_right_feet"/>
</joint>

<linkname="ass_left_knee">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="ass_left_radius_to_knee"type="continuous">
        <axisxyz="1 0 0"/>
        <originxyz="0 -0.015 -0.0975"/>
        <parentlink="ass_left_radius"/>
        <childlink="ass_left_knee"/>
</joint>

<linkname="ass_left_feet">
        <visual>
        <originxyz="0 0 0"rpy="0 0 0"/>
        <geometry>
        <sphereradius="0.02"/>
        </geometry>
        <materialname="white">
        </material>
        </visual>
</link>

<jointname="ass_left_knee_to_feet"type="continuous">
        <axisxyz="0 0 1"/>
        <originxyz="0 0 0"/>
        <parentlink="ass_left_knee"/>
        <childlink="ass_left_feet"/>
</joint>

</robot>
```

CÓDIGO COMPLETO DEL ARCHIVO .SRDF

```
<?xml version="1.0" ?>
<!--This does not replace URDF, and is not an extension of URDF.
    This is a format for representing semantic information about the robot structure.
    A URDF file must exist for this robot as well, where the joints and the links that are referenced are defined
-->
<robot name="hexapod">
<!--GROUPS: Representation of a set of joints and links. This can be useful for specifying DOF to plan for, defining arms, end effectors, etc-->
<!--LINKS: When a link is specified, the parent joint of that link (if it exists) is automatically included-->
<!--JOINTS: When a joint is specified, the child link of that joint (which will always exist) is automatically included-->
<!--CHAINS: When a chain is specified, all the links along the chain (including endpoints) are included in the group. Additionally, all the joints that are parents to included links are also included. This means that joints along the chain and the parent joint of the base link are included in the group-->
<!--SUBGROUPS: Groups can also be formed by referencing to already defined group names-->
<group name="base">
<joint name="world_joint" />
</group>
<group name="body">
<joint name="body_structure" />
</group>
<group name="frontal_right_leg">
<chain base_link="body_central" tip_link="frontal_right_feet" />
</group>
<group name="frontal_left_leg">
<chain base_link="body_central" tip_link="frontal_left_feet" />
</group>
<group name="central_right_leg">
<chain base_link="body_auxiliar" tip_link="central_right_feet" />
</group>
<group name="central_left_leg">
<chain base_link="body_auxiliar" tip_link="central_left_feet" />
</group>
<group name="ass_right_leg">
<chain base_link="body_central" tip_link="ass_right_feet" />
</group>
<group name="ass_left_leg">
<chain base_link="body_central" tip_link="ass_left_feet" />
</group>
<group name="legs">
<group name="frontal_right_leg" />
<group name="frontal_left_leg" />
<group name="central_right_leg" />
<group name="central_left_leg" />
<group name="ass_right_leg" />
<group name="ass_left_leg" />
</group>
<group name="hexapod">
```

<group name="base" />
<group name="body" />
<group name="legs" />
</group>
<!--VIRTUAL JOINT: Purpose: this element defines a virtual joint between a robot link and an external frame of reference (considered fixed with respect to the robot)-->
<virtual_joint name="world_joint" type="floating" parent_frame="world" child_link="body_central" />
<!--PASSIVE JOINT: Purpose: this element is used to mark joints that are not actuated-->
<passive_joint name="world_joint" />
<!--DISABLE COLLISIONS: By default it is assumed that any link of the robot could potentially come into collision with any other link in the robot. This tag disables collision checking between a specified pair of links. -->
<disable_collisions link1="ass_left_feet" link2="ass_left_knee" reason="Adjacent" />
<disable_collisions link1="ass_left_feet" link2="ass_left_radius" reason="Default" />
<disable_collisions link1="ass_left_feet" link2="ass_left_tibia" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="ass_right_femur" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="ass_right_radius" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="central_right_feet" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="central_right_femur" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="central_right_knee" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="central_right_radius" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="ass_left_feet" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="ass_left_tibia" reason="Adjacent" />
<disable_collisions link1="ass_left_femur" link2="ass_right_feet" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="ass_right_femur" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="ass_right_knee" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="ass_right_radius" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="ass_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="body_auxiliar" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="body_central" reason="Adjacent" />
<disable_collisions link1="ass_left_femur" link2="central_left_femur" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="central_right_feet" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="central_right_femur" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="central_right_knee" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="central_right_radius" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="ass_left_femur" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="ass_left_radius" reason="Adjacent" />
<disable_collisions link1="ass_left_knee" link2="ass_left_tibia" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="ass_right_femur" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="ass_right_radius" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="central_right_feet" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="central_right_femur" reason="Never" />

<disable_collisions link1="ass_left_knee" link2="central_right_knee" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="central_right_radius" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="ass_left_knee" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="ass_left_tibia" reason="Adjacent" />
<disable_collisions link1="ass_left_radius" link2="ass_right_feet" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="ass_right_femur" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="ass_right_knee" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="ass_right_radius" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="ass_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="central_right_feet" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="central_right_femur" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="central_right_knee" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="central_right_radius" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="ass_left_radius" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="ass_right_femur" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="ass_right_radius" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="ass_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="central_right_feet" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="central_right_femur" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="central_right_knee" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="central_right_radius" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="ass_left_tibia" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="ass_right_knee" reason="Adjacent" />
<disable_collisions link1="ass_right_feet" link2="ass_right_radius" reason="Default" />
<disable_collisions link1="ass_right_feet" link2="ass_right_tibia" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="central_left_feet" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="central_left_femur" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="central_left_knee" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="central_left_radius" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="central_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="ass_right_feet" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="ass_right_tibia" reason="Adjacent" />
<disable_collisions link1="ass_right_femur" link2="body_auxiliar" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="body_central" reason="Adjacent" />
<disable_collisions link1="ass_right_femur" link2="central_left_feet" reason="Never" />

<disable_collisions link1="ass_right_femur" link2="central_left_femur" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="central_left_knee" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="central_left_radius" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="central_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="central_right_femur" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="ass_right_femur" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="ass_right_radius" reason="Adjacent" />
<disable_collisions link1="ass_right_knee" link2="ass_right_tibia" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="central_left_feet" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="central_left_femur" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="central_left_knee" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="central_left_radius" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="central_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="ass_right_knee" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="ass_right_tibia" reason="Adjacent" />
<disable_collisions link1="ass_right_radius" link2="central_left_feet" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="central_left_femur" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="central_left_knee" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="central_left_radius" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="central_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="ass_right_radius" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="central_left_feet" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="central_left_femur" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="central_left_knee" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="central_left_radius" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="central_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="ass_right_tibia" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="body_auxiliar" link2="body_central" reason="Adjacent" />
<disable_collisions link1="body_auxiliar" link2="central_left_femur" reason="Adjacent" />
<disable_collisions link1="body_auxiliar" link2="central_right_femur" reason="Adjacent" />
<disable_collisions link1="body_auxiliar" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="body_auxiliar" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="body_central" link2="frontal_left_femur" reason="Adjacent" />
<disable_collisions link1="body_central" link2="frontal_right_femur" reason="Adjacent" />
<disable_collisions link1="central_left_feet" link2="central_left_knee" reason="Adjacent" />
<disable_collisions link1="central_left_feet" link2="central_left_radius" reason="Default" />

<disable_collisions link1="central_left_feet" link2="central_left_tibia" reason="Never" />
<disable_collisions link1="central_left_feet" link2="central_right_femur" reason="Never" />
<disable_collisions link1="central_left_feet" link2="central_right_radius" reason="Never" />
<disable_collisions link1="central_left_feet" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="central_left_feet" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="central_left_feet" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="central_left_feet" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="central_left_feet" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="central_left_feet" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="central_left_femur" link2="central_left_tibia" reason="Adjacent" />
<disable_collisions link1="central_left_femur" link2="central_right_feet" reason="Never" />
<disable_collisions link1="central_left_femur" link2="central_right_femur" reason="Never" />
<disable_collisions link1="central_left_femur" link2="central_right_knee" reason="Never" />
<disable_collisions link1="central_left_femur" link2="central_right_radius" reason="Never" />
<disable_collisions link1="central_left_femur" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="central_left_femur" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="central_left_femur" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="central_left_femur" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="central_left_femur" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="central_left_femur" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="central_left_femur" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="central_left_knee" link2="central_left_radius" reason="Adjacent" />
<disable_collisions link1="central_left_knee" link2="central_left_tibia" reason="Never" />
<disable_collisions link1="central_left_knee" link2="central_right_femur" reason="Never" />
<disable_collisions link1="central_left_knee" link2="central_right_radius" reason="Never" />
<disable_collisions link1="central_left_knee" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="central_left_knee" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="central_left_knee" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="central_left_knee" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="central_left_knee" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="central_left_knee" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="central_left_radius" link2="central_left_tibia" reason="Adjacent" />
<disable_collisions link1="central_left_radius" link2="central_right_femur" reason="Never" />
<disable_collisions link1="central_left_radius" link2="central_right_radius" reason="Never" />
<disable_collisions link1="central_left_radius" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="central_left_radius" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="central_left_radius" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="central_left_radius" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="central_left_radius" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="central_left_radius" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="central_right_feet" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="central_right_femur" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="central_right_knee" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="central_right_radius" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="central_left_tibia" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="central_right_feet" link2="central_right_knee" reason="Adjacent" />
<disable_collisions link1="central_right_feet" link2="central_right_radius" reason="Default" />
<disable_collisions link1="central_right_feet" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="central_right_feet" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="central_right_feet" link2="frontal_left_femur" reason="Never" />

<disable_collisions link1="central_right_feet" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="central_right_feet" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="central_right_feet" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="central_right_femur" link2="central_right_tibia" reason="Adjacent" />
<disable_collisions link1="central_right_femur" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="central_right_femur" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="central_right_femur" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="central_right_femur" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="central_right_femur" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="central_right_femur" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="central_right_knee" link2="central_right_radius" reason="Adjacent" />
<disable_collisions link1="central_right_knee" link2="central_right_tibia" reason="Never" />
<disable_collisions link1="central_right_knee" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="central_right_knee" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="central_right_knee" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="central_right_knee" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="central_right_knee" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="central_right_radius" link2="central_right_tibia" reason="Adjacent" />
<disable_collisions link1="central_right_radius" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="central_right_radius" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="central_right_radius" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="central_right_radius" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="central_right_radius" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="central_right_tibia" link2="frontal_left_feet" reason="Never" />
<disable_collisions link1="central_right_tibia" link2="frontal_left_femur" reason="Never" />
<disable_collisions link1="central_right_tibia" link2="frontal_left_knee" reason="Never" />
<disable_collisions link1="central_right_tibia" link2="frontal_left_radius" reason="Never" />
<disable_collisions link1="central_right_tibia" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="frontal_left_feet" link2="frontal_left_knee" reason="Adjacent" />
<disable_collisions link1="frontal_left_feet" link2="frontal_left_radius" reason="Default" />
<disable_collisions link1="frontal_left_feet" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="frontal_left_feet" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="frontal_left_femur" link2="frontal_left_tibia" reason="Adjacent" />
<disable_collisions link1="frontal_left_femur" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="frontal_left_femur" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="frontal_left_femur" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="frontal_left_femur" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="frontal_left_femur" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="frontal_left_knee" link2="frontal_left_radius" reason="Adjacent" />
<disable_collisions link1="frontal_left_knee" link2="frontal_left_tibia" reason="Never" />
<disable_collisions link1="frontal_left_knee" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="frontal_left_radius" link2="frontal_left_tibia" reason="Adjacent" />
<disable_collisions link1="frontal_left_radius" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="frontal_left_radius" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="frontal_left_radius" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="frontal_left_tibia" link2="frontal_right_feet" reason="Never" />
<disable_collisions link1="frontal_left_tibia" link2="frontal_right_femur" reason="Never" />
<disable_collisions link1="frontal_left_tibia" link2="frontal_right_knee" reason="Never" />
<disable_collisions link1="frontal_left_tibia" link2="frontal_right_radius" reason="Never" />
<disable_collisions link1="frontal_left_tibia" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="frontal_right_feet" link2="frontal_right_knee" reason="Adjacent" />
<disable_collisions link1="frontal_right_feet" link2="frontal_right_radius" reason="Default" />
<disable_collisions link1="frontal_right_feet" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="frontal_right_femur" link2="frontal_right_tibia" reason="Adjacent" />
<disable_collisions link1="frontal_right_knee" link2="frontal_right_radius" reason="Adjacent" />

<disable_collisions link1="frontal_right_knee" link2="frontal_right_tibia" reason="Never" />
<disable_collisions link1="frontal_right_radius" link2="frontal_right_tibia" reason="Adjacent" />
</robot>

PRESUPUESTO

Presupuesto

ÍNDICE

ÍNDICE DE TABLAS

Presupuesto

1. INTRODUCCIÓN

En el presente documento se va a obtener el presupuesto que tendría el diseño, monitorización y control de un hexápodo con ROS en el ámbito laboral actual. Para ello se dividirá el trabajo en distintas unidades de obra, obteniendo el precio unitario de cada una de ellas. Una vez obtenido el coste de las tareas se agrupará en un único presupuesto y se le aplicarán los impuestos pertinentes.

2. UNIDADES DE OBRA

Las unidades de obra en las que se va a dividir el proyecto son las distintas tareas que se han tenido que realizar para su desarrollo. En este caso:

- Construcción física del hexápodo.
- Aprendizaje del sistema ROS.
- Aprendizaje de la aplicación MoveIt!
- Implementación de la solución en MoveIt!
- Cálculo e implementación de la cinemática inversa.

3. PRECIOS UNITARIOS

Una vez definidas las unidades de obra se obtendrá el precio de cada una de ellas.

3.1. Construcción Física del Hexápodo

En esta unidad de obra se incluyen todas las tareas que tienen como objetivo transformar la idea y el diseño del robot en un producto físico. Por lo tanto habrá que tener en cuenta la impresión de las piezas y el montaje posterior. Además se incluirán los gastos de compra de material como la tarjeta o los servos. A todo esto se le incluirá la mano de obra que consistirá en un Ingeniero Industrial trabajando durante 15 horas.

Con estos datos se puede obtener el precio unitario de dicha unidad de obra:

Producto	Unidades	Precio (€/ud)	Importe (€)
Filamento ABS 3mm-1kg-BLANCO	1	18,5	18,5
Servo TowerPro SG-5010	18	12	216
Torobot 32 servo controller	1	28,46	28,46
Ingeniero Industrial	15	30	450
PRECIO UNITARIO			**712,96**

Tabla 1. Precio unitario de la construcción física del hexápodo.

3.2. Aprendizaje del sistema ROS

Aquí se considera el coste de todo el proceso de aprendizaje de la herramienta ROS. En esta unidad de obra se incluirá como coste material la compra de un ordenador potente para no tener problemas a la hora de ejecutar los distintos programas. Aunque el ordenador se utilice en las siguientes unidades de obra únicamente se incluirá en esta. Por otro lado, aunque el programa y la realización de los tutoriales son gratuitos, se deberán de incluir las horas trabajadas por el proyectista para aprender a utilizar el sistema, que en este caso son aproximadamente unas 80.

Para obtener el coste de un equipo material como el ordenador donde su utilidad se extenderá más halla del proyecto se debe calcular la parte de su precio que se utiliza en este proyecto con la siguiente expresión (se considera el tiempo de amortización de 5 años y 3 meses el de utilización):

$$\text{Importe } (€) = \text{Coste } (€) * \frac{\text{Tiempo de utilización (h)}}{\text{Tiempo de amortización (h)}} = 961{,}26 * \frac{2.160}{43.800} = 47{,}4€ \qquad (9)$$

Así el precio unitario de la unidad de obra resultaría el siguiente:

Producto	Unidades	Precio (€/ud)	Importe (€)
MacBook Pro 13-inch	1	47,4	47,4
Ingeniero Industrial	80	30	2.400
PRECIO UNITARIO			**2.447,4**

Tabla 2. Precio unitario del aprendizaje del sistema ROS.

3.3. Aprendizaje de la Aplicación MoveIt!

En este caso no aparece ningún coste material, la realización de esta tarea solamente necesita la mano de obra de un ingeniero mientras realiza el tutorial y aprende a utilizar la aplicación MoveIt! para aplicarlo en la primera parte del proyecto. En este caso el tiempo de trabajo se estima en 60 horas dando lugar a un precio unitario:

Producto	Unidades	Precio (€/ud)	Importe (€)
Ingeniero Industrial	60	30	1.800
PRECIO UNITARIO			**1.800**

Tabla 3. Precio unitario del aprendizaje de la aplicación MoveIt!

3.4. Implementación de la solución en MoveIt!

Al igual que en la unidad de obra anterior esta unidad no incluye ningún coste material nuevo, únicamente 100 horas de trabajo de un Ingeniero Industrial. En este caso se incluyen las tareas desarrolladas en la primera de las alternativas presentada en la memoria: diseño de un modelo, preparación y utilización del simulador y obtención, tratado y envío de los datos.

Producto	Unidades	Precio (€/ud)	Importe (€)
Ingeniero Industrial	100	30	3.000
PRECIO UNITARIO			**3.000**

Tabla 4. Precio unitario de la implementación de la solución en MoveIt!

3.5. Cálculo e implementación de la cinemática inversa.

Por último la unidad de obra donde se incluyen las tareas relacionadas con la segunda alternativa de la memoria. Se incluirán las tareas de obtención del modelo matemático para resolver la cinemática inversa, la generación de los cursores para el control del robot y la generación de un ejecutable que agrupe las dos tareas anteriores y consiga mover el hexápodo. Para todo esto se destinan 75 horas de trabajo por parte de un Ingeniero Industrial sin ningún coste material añadido al ser todo tareas de programación e intelectuales.

Por lo tanto el precio unitario sería:

Producto	Unidades	Precio (€/ud)	Importe (€)
Ingeniero Industrial	75	30	2.250
PRECIO UNITARIO			**2.250**

Tabla 5. Precio unitario del cálculo e implementación de la cinemática inversa.

4. PRESUPUESTO TOTAL

Una vez obtenidos los precios unitarios de las distintas unidades de obra se puede obtener el presupuesto total. Aunque primero añadir que la razón de no incluir costes indirectos es debido a que en este proyecto no ha hecho falta ningún trabajo administrativo.

A continuación se obtiene el Presupuesto de Ejecución del material sumando el coste de cada una de las unidades de obra.

Unidad de obra	Precio unitario (€)
Construcción física del hexápodo	712,96
Aprendizaje del sistema ROS	2.447,4
Aprendizaje de la aplicación MoveIt!	1.800
Implementación de la solución en MoveIt!	3.000
Cálculo e implementación de la cinemática inversa	2.250
PRESUPUESTO DE EJECUCIÓN DEL MATERIAL	**10.210,36**

Tabla 6. Presupuesto de ejecución del material.

A este presupuesto para obtener el Presupuesto Total se le aplican dos porcentajes: un 13% extra por Gastos Generales y un 6% por Beneficio Industrial.

Unidad de obra	Precio unitario (€)
Presupuesto de ejecución del material	10.210,36
Gastos generales	1.327,34
Beneficio industrial	612,62
PRESUPUESTO TOTAL	**12.150,32**

Tabla 7. Presupuesto total.

Una vez se ha calculado el Presupuesto Total ya se puede obtener el Presupuesto de Ejecución por Contrata aplicándole el Impuesto sobre el Valor Añadido (I.V.A.) de un 21%.

Unidad de obra	Precio unitario (€)
Presupuesto total	12.150,32
I.V.A.	2.551,56
PRESUPUESTO DE EJECUCIÓN POR CONTRATA	**14.701,88**

Tabla 8. Presupuesto de ejecución por contrata.

Por lo tanto el coste del desarrollo de este proyecto ascendería a 14.701,88€.