UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

SIMULACIÓN, IMPLEMENTACIÓN Y CONTROL DE UN ROBOT HEXÁPODO

MEMORIA

REALIZADO POR **David Laseca Pérez**

TUTORIZADO POR **Vicente Fermín Casanova Calvo**

TRABAJO FINAL DEL GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

CURSO ACADÉMICO 2020/2021

Agradecimientos

Me gustaría agradecer a mi tutor por su tiempo y su dedicación en la realización de este proyecto. A mi familia, por su cariño y apoyo, y a mi perro, Willy, por siempre estar ahí para mí. Finalmente, gracias a mis amigos, con los que tanto he compartido durante estos años.

Resumen

Desde sus inicios, la robótica ha sido una disciplina que ha demostrado ser de gran utilidad en una gran variedad de aplicaciones y despertado mucha curiosidad. El uso de robots permite desempeñar toda clase de actividades tradicionalmente realizadas por personas mejorando las condiciones de seguridad y eficiencia de las mismas.

El presente proyecto plantea el desarrollo de un robot caminante de seis extremidades o robot hexápodo, con el fin de obtener un modelo de robot capaz de funcionar de manera teleoperada, es decir bajo el control de un ser humano, en un entorno controlado.

A lo largo de este documento se abordarán distintos aspectos de la robótica y se justificará la elección de este modelo de robot. En primer lugar será necesario realizar un estudio del modelo del robot para analizar sus movimientos y obtener un sistema teórico para gestionar sus desplazamientos sobre el plano. Una vez completado dicho sistema, se implementará junto a la estructura del robot en el entorno de simulación de Simulink Simscape Multibody, para analizar la corrección y eficiencia de los movimientos. Finalmente, tras obtener un resultado satisfactorio en la simulación, se procederá a implementar el modelo del robot en la realidad. Para ello se emplearán distintos componentes electrónicos y un conjunto de piezas impresas en 3D. También se desarrollará un programa y una aplicación móvil que permita controlar el robot desde un teléfono móvil.

Abstract

Since its inception, robotics has been a discipline that has proven to be very useful in a wide variety of applications and aroused a lot of curiosity. The use of robots allows to carry out all kinds of activities traditionally done by people, improving their safety and efficiency conditions.

The present project proposes the development of a six-limbed walking robot or hexapod robot, in order to obtain a robot model capable of working under the control of a human being, in a controlled environment using a device to drive it.

Throughout this document, different aspects of robotics will be addressed and the choice of this robot model will be justified. In the first place, it will be necessary to carry out a study of the robot model to analyze its movements and obtain a theoretical system to manage its movements on the plane. Once this system is completed, it will be implemented together with the robot structure in the Simulink Simscape Multibody simulation environment, to analyze the correct functioning and efficiency of the movements. Finally, after obtaining a satisfactory result in the simulation, the robot model will be implemented in reality. For this, different electronic components and a set of 3D printed parts will be used. A program and a mobile application will also be developed to control the robot from a mobile phone.

Resum

Des dels seus inicis, la robòtica ha sigut una disciplina que ha demostrat ser-ne de gran utilitat en una gran varietat d'aplicacions, a banda de despertar-ne molta curiositat. L'ús de robots permet realitzar tota mena d'activitats tradicionalment realitzades per persones, millorant les condicions de seguretat i eficiència d'aquestes.

El present projecte disposa el desenvolupament d'un robot caminant de sis extremitats o robot hexàpode, amb la fi d'obtindre un model de robot capaç de funcionar de manera teleoperada, és a dir, baix el control d'un ser humà, en un entorn controlat.

Al llarg d'aquest document s'abordaran diversos aspectes de la robòtica i es justificarà l'elecció d'aquest model de robot. En primer lloc, serà necessari realitzar un estudi del model del robot per analitzar els seus moviments i obtindre un sistema teòric per a gestionar els desplaçaments sobre el pla. Una vegada completat dit sistema, s'implementarà junt l'estructura del robot a l'entorn de simulació de Simulink Simscape Multibody, per analitzar la correcció i eficiència dels moviments. Finalment, després d'obtindre un resultat satisfactori en la simulació, es continuarà implementant el model del robot en la realitat. Per tal d'aconseguir-lo, s'utilitzaran diferents components electrònics i un conjunt de peces impreses en 3D. També es desenvoluparà un programa i una aplicació mòbil que permetrà controlar el robot des d'un telèfon mòbil.

Índice general

Índice de figuras

Índice de tablas

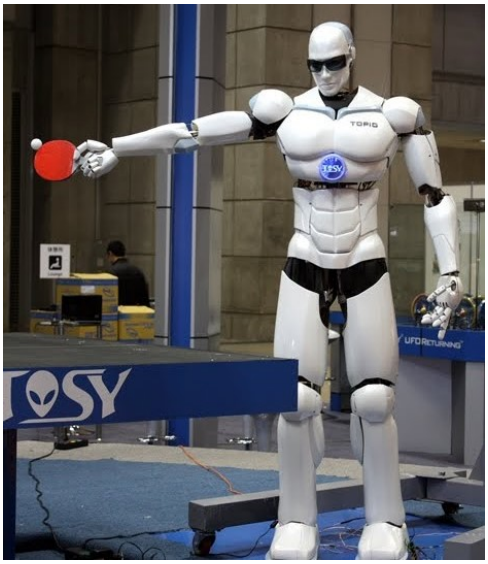
1

Introducción

Un robot es una máquina programable capaz de llevar a cabo complejas secuencias de acciones de manera automática para cumplir un objetivo. Los robots tienen su origen en la creación de máquinas que reemplacen y mejoren el rendimiento de un humano a la hora de realizar una determinada tarea, mientras que la robótica es el campo de la ingeniería especializado en el diseño, construcción y control de robots.

La robótica empezó a desarrollarse en profundidad en la segunda mitad del siglo XX, pese a que ya habían existido máquinas automáticas o semiautomáticas desde mucho tiempo antes. La humanidad siempre ha imaginado y tratado de crear herramientas que realicen tareas en su lugar. En un principio se trataban de artificios mecánicos con el objetivo de impresionar o entretener, pero con el paso del tiempo se empezaron a crear sistemas más complejos con aplicaciones industriales. A día de hoy, los robots se encuentran presentes en muy diversos ámbitos, predominan los robots diseñados para procesos industriales, pero existen robots que abarcan un amplio rango de disciplinas, desde la medicina hasta la documentación y exploración, en la Figura 1.1 se puede observar de manera sencilla la evolución que han sufrido los robots desde sus orígenes[11].

(a) Robot humanoide de Leonardo Da Vinci.



(b) TOPIO, Robot diseñado para jugar Ping Pong.

Figura 1.1: Comparativa que muestra la evolución de los robots.

Existen muchas clasificaciones para los robots, que pueden hacer referencia al modo de controlarlo, a su aspecto o a su funcionalidad. La idea de robot ha evolucionado en gran medida desde sus orígenes, y ahora mismo nos encontramos en un mundo lleno de robots, lo que se ha convertido en una tendencia que no parece decrecer.

1.1. Motivación

Uno de los campos de la robótica que se encuentra en especial auge es el de los robots móviles, que hace referencia a aquellos robots que poseen la capacidad de moverse por su entorno y no se encuentran fijos en un lugar. Esta clase de robots suele asemejarse a vehículos con ruedas y sus principales aplicaciones son el transporte de objetos o carga y el entretenimiento.

Dentro de este campo se encuentran también los robots caminantes, que son aquellos robots móviles que se desplazan por el medio empleando patas para ello. Existen múltiples configuraciones para esta clase de robot, pero destacan por su capacidad de reaccionar ante situaciones adversas y de moverse en terrenos accidentados o con desniveles.

En muchos casos los robots caminantes se inspiran en la biología, este detalle los hace muy interesantes de cara al público general y les ofrece mucha versatilidad en cuanto a sus

movimientos. Puesto que se trata de un campo con mucho futuro por delante, y que aún se encuentra en desarrollo, se ha escogido este tema como el eje principal del proyecto.

1.2. Objetivos del proyecto

En este trabajo se ha planteado el control de un robot hexápodo, un robot caminante de seis extremidades, que presenta la gran ventaja de poder mantener en todo momento el extremo de tres de ellas apoyadas, lo cual aumenta en gran medida su estabilidad. A diferencia de otros robots caminantes con menos patas, no se requiere que el robot se equilibre a lo largo del movimiento, por lo que los cálculos necesarios para desarrollar los sistemas que le permitan avanzar y girar son mucho más sencillos.

No se ha considerado la etapa de diseño del proyecto, sino que se han obtenido los modelos 3D de las piezas del robot en Internet, en concreto de una versión de código abierto llamada Antdroid[12]. Este proyecto se ha centrado en el desarrollo del funcionamiento de los robots hexápodos, por lo que el diseño no forma parte del mismo, y se podría aplicar gran parte de los resultados del proyecto a otro modelo si se modifican los parámetros adecuados. Se planteado los siguientes objetivos para llevar a cabo en el proyecto:

- Desarrollo de los movimientos de las extremidades del robot mediante la cinemática directa e inversa de las mismas.
- Combinación de dichos movimientos para la generación de secuencias que provoquen el desplazamiento del robot.
- Simulación empleando Simulink Simscape Multibody del cuerpo y los movimientos del hexápodo.
- Estudio de los componentes y electrónica necesarios para la implementación del robot.
- Montaje del robot.
- Programación de la cinemática y los movimientos del robot empleando un microcontrolador Wroom ESP32.
- Manejo del robot de forma teleoperada empleando un teléfono móvil y el módulo Bluetooth Low Energy del microcontrolador.

2

Marco teórico

Existe una gran cantidad de robots diseñados específicamente para trabajar en toda clase de ámbitos y lugares. A continuación se listan algunos de los que son más usados hoy en día[13].

2.1. Tipos de robots

2.1.1. Robots industriales

Son los más comunes y extendidos, en su mayoría se trata de robots cartesianos capaces de soportar una gran carga de trabajo o de robots manipuladores, que son brazos robóticos articulados con una base fija. Son usados en una gran variedad de aplicaciones dentro de los procesos de manufacturación de bienes, como puede ser la soldadura, la pintura o el montaje. Se trata de robots muy rápidos y precisos, capaces de realizar una amplia gama de tareas y ofrecen un gran ahorro en los procesos productivos. De acuerdo con la Federación Internacional de Robótica (International Federation of Robotics, IRF), habría aproximadamente 2.6 millones de robots industriales en funcionamiento para 2019. En la Figura 2.1 se puede observar un brazo robótico industrial de la serie KUKA KR titan.

Figura 2.1: Robot KUKA KR 1000 titan[1].

2.1.2. Robots domésticos

Se trata de robots de tamaño moderado creados para llevar a cabo tareas domésticas, de este modo han surgido robots aspiradoras y robots limpiadores de piscinas entre otros. Se consideran dentro de este campo otros robots de mayores dimensiones creados para la limpieza de espacios más grandes, como locales comerciales. En la Figura 2.2 se muestra un robot de esta clase.

Figura 2.2: Robot limpiador de uso doméstico[2].

2.1.3. Robots militares y de respuesta a emergencias

Los robots cumplen una importante labor en los ámbitos militar y de respuesta a emergencias, donde pueden surgir situaciones imprevisibles y que implican una respuesta rápida. Los militares emplean robots para exploración y reconocimiento aéreo, y los equipos de emergencias usan robots aéreos, terrestres y submarinos para la evaluación de zonas y la búsqueda de supervivientes, además de robots de rescate. Esta clase de robots son capaces de aguantar condiciones muy duras y de atravesar terrenos accidentados, por lo que también se usan en el campo de la exploración incluso en otros planetas, como con el vehículo explorador Curiosity en Marte, mostrado en la Figura 2.3[3].

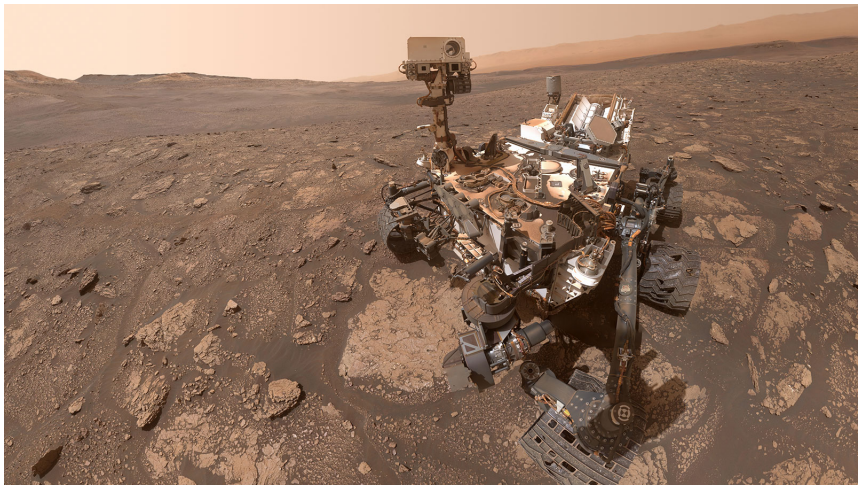


Figura 2.3: Robot explorador Curiosity, de la NASA[3].

2.1.4. Robots médicos

Los robots médicos mas extendidos son los robots quirúrgicos, aunque también se usan robots para transportar materiales y equipos en hospitales. Se emplean para cirugías asistidas por robot, donde el cirujano controla el robot, normalmente a través de un ordenador. Estas máquinas proporcionan la posibilidad de ejecutar procedimientos médicos muy delicados y precisos, reduciendo así los riesgos que estas prácticas conllevan. Un ejemplo de esta clase de robot se observa en la Figura 2.4.

Figura 2.4: Robot quirúrgico Da Vinci XI[4].

2.1.5. Robots de seguridad

Son robots especializados en la vigilancia y reconocimiento, en muchos casos se dedican a patrullar alrededor de una ruta establecida y reportar situaciones anormales. Pueden incorporar desde sistemas de vigilancia con reconocimiento facial hasta sensores para detectar temperaturas e incluso gases fuera de lo habitual. También se emplean para tareas que se consideran demasiado peligrosas para un ser humano, una de ellas es la desactivación de bombas, donde se encuentra el robot de la Figura 2.5

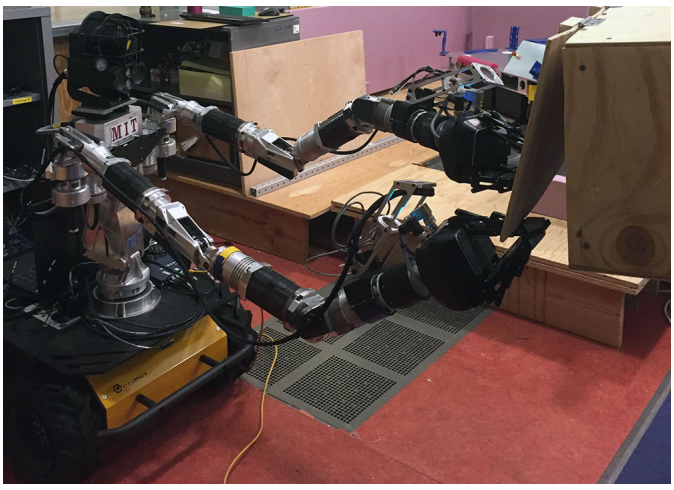


Figura 2.5: Prototipo de robot desactivador de bombas del MIT[5].

2.1.6. Robots colaborativos

Un robot colaborativo es aquel que puede interactuar con humanos de manera segura y efectiva mientras lleva a cabo una tarea. Estos robots son diseñados con sensores específicos y programados de forma que se detienen o limitan en gran medida su movimiento cuando detectan la presencia de un humano cerca. Estos robots aún tienen un largo camino por recorrer, pero desde su aparición se han vuelto más y más comunes con el tiempo[6]. Otra característica de estos robots es que pueden ser guiados por un humano para aprender a realizar una tarea, como es el caso de la Figura 2.6.

Figura 2.6: Robot colaborativo de Omron[6].

2.2. Robots móviles

Los robots móviles son aquellos que tienen la capacidad de desplazarse en un entorno, es decir, no se encuentran fijados en un espacio, como es el caso de la mayoría de brazos robóticos industriales. Estos robots pueden ser autónomos, lo que implica que son capaces de trazar recorridos a lo largo de un entorno sin la necesidad de ser guiados. Por otra parte, una gran cantidad de estos robots emplean mecanismos que les permiten ser manejados directamente por humanos o recorrer rutas preestablecidas[14].

Estos robots son cada vez más habituales en entornos industriales, comerciales e incluso domésticos. Durante muchos años se han estado empleando robots móviles para transportar material en hospitales y almacenes. También se trata de un campo de investigación muy impor-

tante y muchas universidades tienen laboratorios centrados en este campo.

Esta clase de robots se componen de un núcleo o controlador, sensores, actuadores y fuente de energía. El controlador suele ser un microprocesador, un sistema embebido o un ordenador. Por otra parte, los sensores y actuadores dependen en gran medida de la aplicación. Entre los sensores son comunes los de distancia y táctiles para la evitación de obstáculos, y también las cámaras y los sensores ópticos para reconocimiento de características del entorno. Los actuadores incluyen principalmente los motores que pueden llevar a cabo los desplazamientos del robot, como pueden ser los que mueven las ruedas o patas de un robot terrestre o las hélices de un cuadricóptero, pero en ocasiones también existen otros actuadores con los que el robot puede interactuar con el medio.

En muchas ocasiones los robots móviles son clasificados por el terreno por el que pueden moverse, se encuentran así:

- Robots terrestres, son los llamados UGVs (Unmanned Ground Vehicles) por sus siglas en inglés. La mayoría usan ruedas u orugas, pero también incluyen a los robots caminantes.
- Robots de transporte, que son diseñados para mover materiales y suministros en un entorno de trabajo.
- Robots aéreos, que son llamados habitualmente UAVs (Unmmaned Aerial Vehicles).
- Robots submarinos, es común referirse a ellos como AUVs (Autonomous Underwater Vehicle).
- Robots polares, aquellos especializados en recorrer zonas heladas y también los empleados para explorar las grietas de los glaciares.

También es habitual realizar una clasificación atendiendo a su medio de locomoción, de este modo se realiza la distinción entre robots con ruedas y robots con patas o caminantes.

2.3. Robots caminantes

Los robots caminantes emplean extremidades articuladas como medio de locomoción. Esto los hace mucho más versátiles frente a los robots con ruedas, ya que pueden moverse por toda

clase de terrenos. Por otra parte, estos robots son mucho más complejos y requieren de un mayor consumo energético. En muchos casos los robots caminantes imitan a animales como humanos o insectos en un ejemplo de biomímesis[15].

La imitación del movimiento de los animales requiere un estudio del paso o andadura del animal, que se puede llevar a cabo analizando el patrón de movimiento de sus extremidades durante la locomoción. Este paso puede implementarse en el robot a través del control de sus articulaciones junto con sensores que determinen la inclinación del cuerpo y la situación de apoyo de las patas. Finalmente, para completar el paso debe combinarse lo anterior con algoritmos de planificación de dirección y velocidad de movimiento.

Otra característica de los robots caminantes es que su centro de gravedad debe ser soportado por el cuerpo, de manera estática o dinámica, para mantener el movimiento. El soporte estático se produce cuando se asegura que el centro de gravedad se encuentra dentro del patrón formado por las patas en contacto con el suelo. En contraposición, el soporte dinámico se implementa manteniendo conocida la localización del centro de gravedad del robot y la trayectoria que ha descrito, de modo que se pueda reposicionar por las fuerzas ejercidas por sus patas.

Del mismo modo que otros tipos de robots, existen clasificaciones para los robots caminantes, siendo una de las más comunes la clasificación atendiendo al número de extremidades de los mismos[16].

2.3.1. Robots saltadores

Se trata de robots con una única extremidad, por lo que son tremendamente inestables. Para desplazarse emplean una dinámica basada en pequeños saltos mientras se mantiene el equilibrio del conjunto. Un ejemplo bastante conocido es el robot SALTO de Berkeley[7], mostrado en la Figura 2.7.

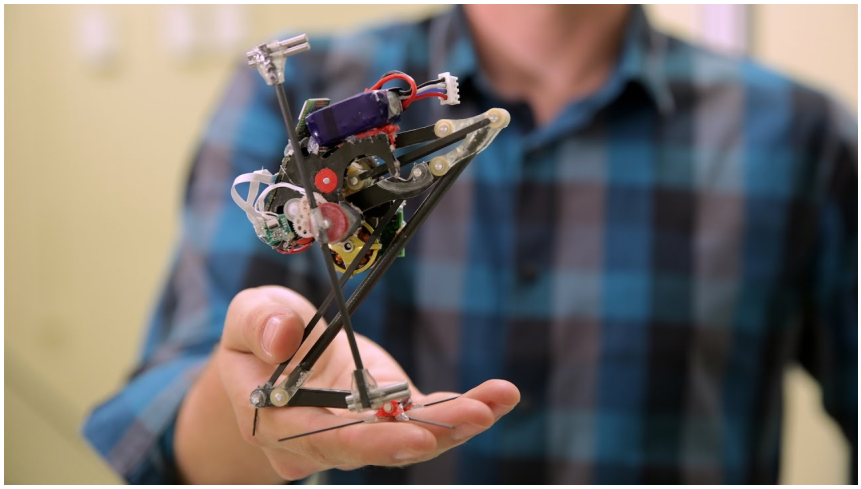


Figura 2.7: Robot saltador SALTO, de Berkeley[7].

2.3.2. Robots humanoides

Son robots que emplean un movimiento bípedo y su estructura se asemeja a la humana. Requieren de un complejo sistema de control de movimientos y estabilidad para desplazarse, ya que esta configuración implica la necesidad de mantener el equilibrio a la vez que se realiza toda clase de movimientos e incluso cuando el robot se encuentra detenido.

En algunos casos, especialmente para juguetes, se ha solucionado este problema con un diseño con unos pies muy amplios, que proveen de una gran estabilidad, aunque reducen la movilidad. Otros modelos más avanzados emplean varios sensores como acelerómetros o giroscopios para poder crear un sistema realimentado que podría compararse al equilibrio de los seres humanos. Algunos robots incluidos como robots humanoides son Atlas, representado en la Figura 2.8, de Boston Dynamics; ASIMO, de Honda; y TOPIO, de TOSY.

Figura 2.8: Robot humanoide Atlas, de Boston Dynamics[8].

2.3.3. Robots cuadrúpedos

Los robots cuadrúpedos son aquellos que, como su nombre indica, emplean cuatro extremidades para la locomoción. Disfrutan de una mayor estabilidad respecto a los robots bípedos, especialmente durante el movimiento, pero también de la posición más baja del centro de gravedad del robot.

Estos robots presentan principalmente dos métodos para desplazarse, a bajas velocidades pueden mover únicamente una de sus patas cada vez de modo que se forme una estructura de trípode estable con el resto. Para moverse a mayores velocidades se requieren movimientos que implican situaciones donde únicamente dos patas o menos se encontraran apoyadas, lo que dificulta en gran medida el diseño y programación de los pasos.

Existen muchos robots cuadrúpedos que se han dado a conocer por diferentes empresas, como BigDog, Cheetah o Spot, de Boston Dynamics[8]; la serie TITAN del Tokyo Institute of Technology; y Mini cheetah, del MIT[17]. Algunos de ellos se muestran en la Figura 2.9.

(a) Robot BIGDOG, de Boston Dynamics[8].

(b) Robot MiniCheetah, del MIT[17].

Figura 2.9: Robots cuadrúpedos.

2.3.4. Robots hexápodos

Esta clase de robots surgen debido a la motivación de obtener una mayor estabilidad. Por su diseño, normalmente imitan la locomoción de los insectos. Los robots hexápodos presentan dos tipos de paso principalmente, que son los más comunes entre los insectos:

- El paso de tipo ola, en el cual el robot mueve sus patas en parejas, describiendo una especie de ola al final de la serie de movimientos de sus tres pares de extremidades. Es el paso más lento.
- El paso de tipo trípode, en este paso tres patas avanzan cada vez, mientras que las tres patas restantes permanecen apoyadas, dando lugar así a una estructura similar a un trípode. Este paso es ligeramente más rápido que el paso de tipo ola.

Existe un tercer tipo de paso para esta clase de configuración de las extremidades, pero no se ha dado de manera inherente en la naturaleza y no es muy común en la robótica tampoco. Este movimiento se podría describir como la extrapolación del paso realizado por animales como perros o guepardos al caso de seis extremidades, y parece ser más rápido que los anteriores[18].

Dentro de los robots hexápodos destacan la serie de robots de juguete Hexbug, y el robot MorpHex, de Zenta Robotic Creations[9], representado en la Figura 2.10, entre otros.

Figura 2.10: MorpHex, de Zenta Robotic Creations[9].

2.3.5. Robots de ocho patas

Estos robots han sido inspirados por arañas y otros arácnidos, y ofrecen la mayor estabilidad entre los robots caminantes enumerados. No son muy comunes, ya que los robots hexápodos funcionan de manera muy similar y son más sencillos y económicos generalmente. Dentro de esta clase de robots destaca Dante, un robot diseñado para la exploración de un volcán desarrollado por la Universidad Carnegie Mellon[16].

2.3.6. Robots híbridos

Existen robots que usan una combinación de patas y ruedas. Esta combinación permite aprovechar parte de las ventajas de ambas posibilidades, la velocidad y eficiencia energética de las ruedas al igual que la movilidad de las extremidades articuladas, como es el ejemplo del robot Handle, mostrado en la Figura 2.11, de Boston Dynamics[10].

Figura 2.11: Robot Handle, de Boston Dynamics[8][10].

2.4. Análisis del robot empleado

Los robots hexápodos presentan una gran estabilidad, como se ha explicado anteriormente. Esto se debe a la cantidad de patas que presentan, en concreto debido a que pueden mantener su cuerpo completamente estabilizado mientras se desplazan, al tener la cantidad de patas suficiente como para formar un triángulo sobre el suelo con las patas apoyadas en todo momento, de modo que el centro de gravedad del robot se encontraría dentro de dicho triángulo y el robot estaría estable. Se puede apreciar este hecho en las imágenes de la Figura 2.12, donde se representan dos configuraciones opuestas de la secuencia que el robot realiza para avanzar, y se muestra que el centro de gravedad del robot se encuentra dentro de los triángulos formados por las patas. Debido a la complejidad presente en los robots con menos patas y a la necesidad de sensores y estructuras de control más complejas para poder diseñar los movimientos necesarios en dichos robots, se ha optado por el modelo de robot hexápodo.

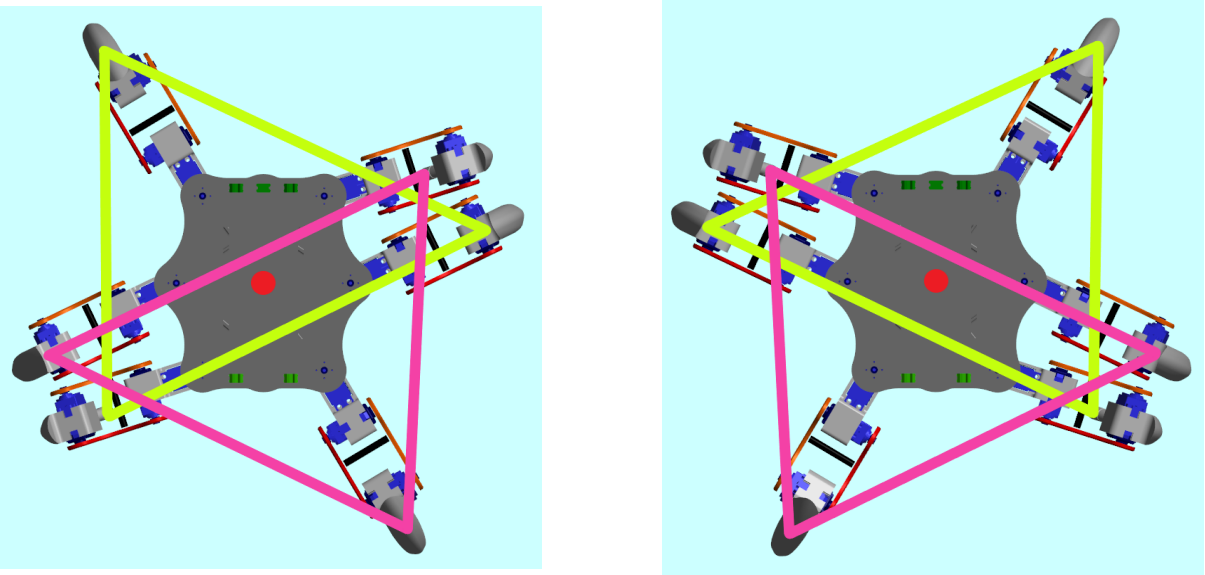


Figura 2.12: Localización del centro de gravedad del robot en relación a la posición del extremo de sus patas.

Para el proyecto se ha decidido emplear un diseño de robot hexápodo ya existente en lugar de crear uno propio, dado que la etapa de diseño no forma parte de los objetivos a desarrollar. Resulta sencillo encontrar realizando una búsqueda en Internet una gran variedad de diseños disponibles para obtener e imprimir sus piezas en 3D, de modo que se han de fijar ciertas características que el diseño debe tener.

En primer lugar, se han de descartar todos los modelos que carecen de tres grados de libertad por cada pata, dado que esta condición es estrictamente necesaria para poder llevar a cabo movimientos que permitan el avance del robot sin el deslizamiento de las patas. Una extremidad con dos grados de libertad, lo que se puede considerar como un brazo robótico con únicamente dos articulaciones no puede realizar trayectorias definidas en un espacio tridimensional. Por este motivo se necesita un modelo que disponga de tres articulaciones por cada pata, de modo que cada pata posea tres grados de libertad espaciales (X, Y, Z) de manera independiente. Este aspecto permitirá desarrollar a partir de la cinemática directa e inversa del modelo, las trayectorias de las extremidades que resultarán en el avance o giro del robot.

Otro factor a considerar en el diseño es el tamaño, se desea un diseño de tamaño moderado y robusto, dado que una de sus posibles aplicaciones sería la exploración en terrenos accidentados. También por este motivo se ha escogido un modelo cuyas patas tengan un extremo de longitud

considerable, esto aportará la posibilidad de levantar o bajar el cuerpo del robot para que se adapte a distintas situaciones sin que el cuerpo pueda colisionar con el suelo y sin necesitar cambios amplios en la posición de las articulaciones.

Un aspecto de gran importancia que se debe tener en cuenta en esta clase de robots es el cálculo referente a la capacidad de los motores de las articulaciones para aguantar el peso del robot. Este cálculo determinará el par mínimo que deben poseer los motores para poder mover adecuadamente al robot.

3

Cinemática y descripción de los movimientos

La cinemática se define como un subcampo de la física, encargado de describir el movimiento de puntos, cuerpos y conjuntos de objetos sin considerar las fuerzas que causan dichos movimientos [19].

Se trata de una disciplina de gran utilidad en el ámbito de la robótica, dado que permite estudiar la movilidad de los robots en función del estado de sus articulaciones. Para llevar a cabo este estudio en el modelo de robot hexápodo propuesto se debe crear un sistema de referencia que actuará de raíz para localizar el resto de sistemas que se emplearán. Es importante destacar que se usarán sistemas dextrógiros, es decir, el producto vectorial de dos de sus componentes producirá como resultado la tercera de ellos. Esta clase de sistemas son los generados por la conocida regla de la mano derecha Figura 3.1, y también son los opuestos a los sistemas levógiros.

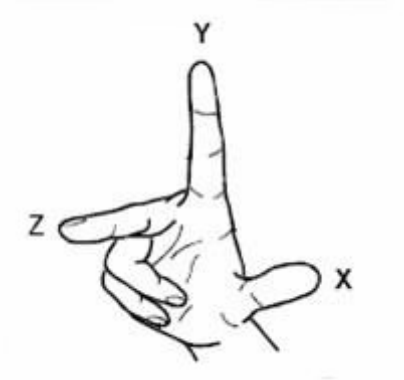


Figura 3.1: Sistema dextrógiro según la regla de la mano derecha.

Dado que existe cierta libertad en la elección de la posición del sistema de referencia se

determinará su ubicación y orientación atendiendo a los siguientes criterios: el sistema raíz se encontrará en el centro geométrico de la cubierta superior del robot, ya que es donde se encuentran ancladas las articulaciones que la unen con las patas; se tomará el eje X en la dirección de avance del robot; y se orientará el eje Z en la dirección perpendicular a la base del robot y con sentido vertical. Una vez determinados los ejes X y Z, el eje Y se determina de manera automática para se cumpla la propiedad de sistema dextrógiro.

A partir del sistema central del robot, que se considerará el número 0, se han de determinar los sistemas de referencia origen de cada una de las patas. Estos sistemas se encontrarán a la misma altura que el sistema 0, es decir, sobre la cubierta del robot, pero desplazados sobre el eje de rotación de cada articulación. Estos sistemas serán numerados de modo que los números impares corresponderán a los sistemas del lado izquierdo del robot, y los números pares a los sistemas del lado derecho. Adicionalmente, se respetará que los sistemas tendrán orientado su eje X en la dirección de avance del robot, pero en este caso, cada sistema orientará su eje Z de modo que los giros producidos respecto a este sean positivos cuando la extremidad del robot de desplace en la dirección definida por el eje X. Esto provoca que los sistemas del lado derecho del robot coincidan en orientación con el sistema 0, mientras que los del lado izquierdo solo tendrán en común el eje X. La disposición de los sistemas de referencia del robot se puede apreciar en la Figura 3.2.

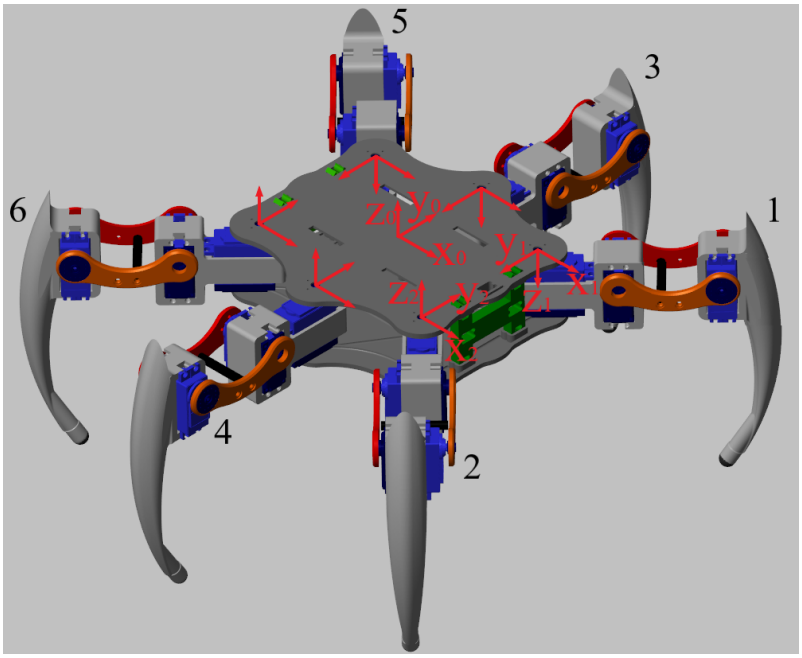


Figura 3.2: Sistemas de referencia sobre el cuerpo del robot. No se indican las componentes de todos ellos por ser idénticas para cada lado.

Una vez ubicados los sistemas de referencia de cada extremidad, es posible trabajar con la posición de los extremos de cada una de ellas respecto a su propio sistema, pero también es posible relacionar dichos puntos con el sistema origen del robot. En realidad, calcular la posición de cada extremo respecto al sistema de referencia origen solo es necesario para movimientos de giro, donde las patas deben trazar arcos alrededor del centro geométrico del robot, y para aplicaciones de equilibrio.

Dada la configuración de las extremidades, cada una de ellas puede ser tratada como un brazo robótico de 3 grados de libertad, esto se debe a que cada extremidad tiene 3 articulaciones que le proporcionan la capacidad de moverse por un espacio tridimensional. Además, cada uno de estos brazos será coplanar, dado que la articulación que une la base con el brazo determina un plano por el cual se mueve todo el resto del brazo.

3.1. Cinemática inversa

La cinemática inversa es el proceso matemático de calcular los valores de la posición de las articulaciones necesarios para localizar el final de una cadena cinemática, como es en este caso la extremidad del hexápodo, en una posición dada.

A continuación se muestran los cálculos correspondientes a una pata del lado izquierdo del robot, aunque los correspondientes a una pata del lado derecho son idénticos cambiando la orientación y signo de algunas componentes. Para resolver la cinemática se han empleado ciertos parámetros auxiliares, y se han denotado las posiciones angulares de cada articulación de la pata como q_1, q_2 y q_3, con la numeración correspondiente a la cadena cinemática de la extremidad, considerando el cuerpo del robot el origen de la cadena.

En la Figura 3.3 se muestra una imagen de la pata del robot vista desde abajo, donde se ha representado el sistema de referencia junto con el valor de q_1, y de un parámetro auxiliar r, que denota la distancia entre el origen y el extremo de la pata en el plano XY. La tupla (p_x, p_y, p_z) representa las coordenadas del punto en el que se encuentra el extremo de la pata respecto al sistema de referencia de la misma.

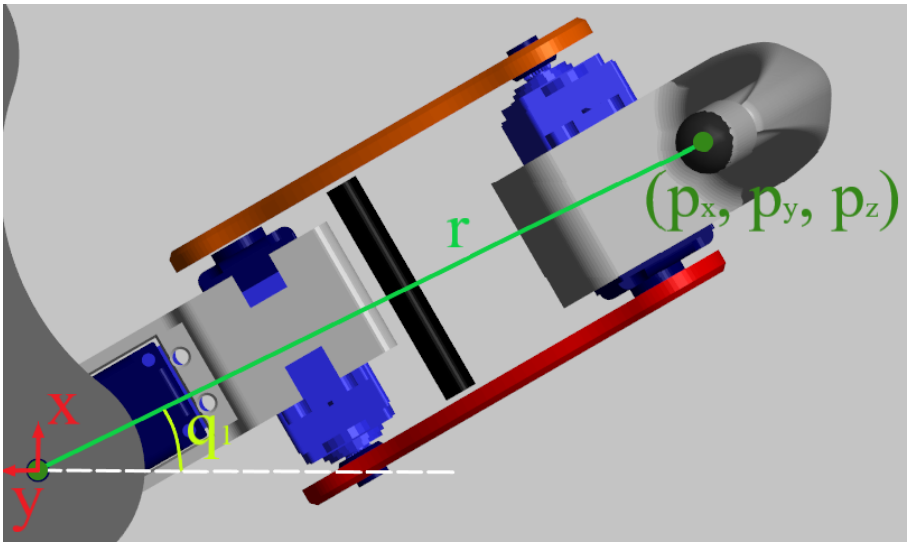


Figura 3.3: Descripción gráfica del valor asociado a q_1 en una pata izquierda.

En la Figura 3.4 se muestra la parte contenida en un plano de la extremidad del robot. En este caso la geometría es algo más compleja, al existir dos grados de libertad en dicho plano, y se requieren de una serie de parámetros adicionales para poder simplificar la resolución.

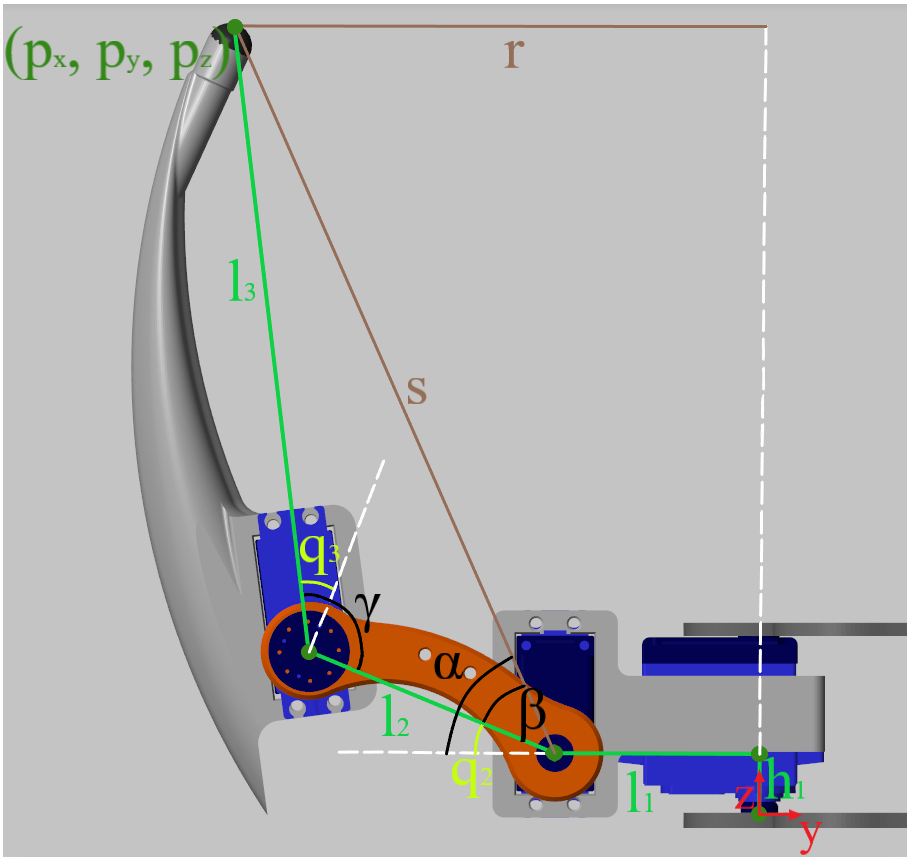


Figura 3.4: Descripción gráfica de los valores asociados a q_2 y q_3 en una pata izquierda.

Los parámetros h_1, l_1, l_2 y l_3 empleados corresponden a las dimensiones de los segmentos que representan en el robot real. El parámetro r coincide con el de mismo nombre de la Figura 3.3, aunque debe tenerse en cuenta que puede dar la sensación de ser más corto debido a la perspectiva. Finalmente, los parámetros s, α, β y γ hacen referencia a distancias y ángulos auxiliares que serán necesarios para obtener los valores de las articulaciones q_2 y q_3. Cabe mencionar que el sistema de referencia mostrado, es el que actúa como origen de una de las patas del lado izquierdo del robot, es decir, podría ser cualquiera de los sistemas 1, 3 o 5, según como han sido mostrados en la Figura 3.2.

Una vez definidos los parámetros adecuados se debe crear el sistema que permite calcular los valores angulares de las articulaciones en función del punto en el que se encuentre el extremo y de los parámetros del brazo. Para ello, se debe comenzar despejando los valores de r y q_1, obteniendo las expresiones de la Ecuación 3.1.

$$\begin{aligned} r &= \sqrt{p_x^2 + p_y^2} \\ q_1 &= arctan(\tfrac{p_x}{-p_y}) \end{aligned} \tag{3.1}$$

Tras obtener el valor de r, resulta posible despejar el resto de parámetros empleando el teorema del coseno para β y γ, como se puede apreciar en la Ecuación 3.2.

$$\begin{aligned} s &= \sqrt{(r-l_1)^2 + (p_z - h_1)^2} \\ \alpha &= arctan(\tfrac{p_z - h_1}{r - l_1}) \\ \beta &= arccos(\tfrac{s^2 + l_2^2 - l_3^2}{2 \cdot s \cdot l_2}) \\ \gamma &= arccos(\tfrac{l_2^2 + l_3^2 - s^2}{2 \cdot l_2 \cdot l_3}) \\ q_2 &= \alpha - \beta \\ q_3 &= \gamma - \tfrac{\pi}{2} \end{aligned} \tag{3.2}$$

De este modo es posible obtener una expresión de los ángulos que deben tomar las articula-

ciones del brazo para llevar el extremo al punto introducido previamente. Es necesario señalar que el cálculo representado es válido únicamente para las tres patas del lado izquierdo, las patas del lado derecho presentan un sistema de referencia con el eje Z en el sentido opuesto, lo que implica que un mismo punto en este eje será tomaría valor positivo para una pata del lado izquierdo y negativo para el derecho. Por este motivo los ángulos de las articulaciones del lado derecho han de determinarse con las ecuaciones señaladas en la Ecuación 3.3.

$$\begin{aligned} r &= \sqrt{p_x^2 + p_y^2} \\ q_1 &= arctan(\tfrac{p_x}{-p_y}) \\ s &= \sqrt{(r - l_1)^2 + (p_z - h_1)^2} \\ \alpha &= arctan(\tfrac{-p_z - h_1}{r - l_1}) \\ \beta &= arccos(\tfrac{s^2 + l_2^2 - l_3^2}{2 \cdot s \cdot l_2}) \\ \gamma &= arccos(\tfrac{l_2^2 + l_3^2 - s^2}{2 \cdot l_2 \cdot l_3}) \\ q_2 &= \alpha - \beta \\ q_3 &= \gamma - \tfrac{\pi}{2} \end{aligned} \tag{3.3}$$

A partir de las ecuaciones listadas anteriormente se puede obtener la configuración de las articulaciones necesaria para llevar el extremo de cada pata robot a un punto determinado.

3.2. Cinemática directa

La cinemática directa funciona de manera similar a la cinemática inversa, solo que en este caso las ecuaciones cinemáticas del robot se emplean para calcular la posición del efector final, en este caso el extremo de la pata, a partir de unos valores dados para las posiciones angulares de las articulaciones.

A la hora de resolver la cinemática directa de las extremidades robot hexápodo se presenta

el mismo inconveniente que se daba con la cinemática directa, y es la diferencia en los cálculos que se produce para cada lado del robot. Se comenzará calculando la cinemática directa correspondiente al lado izquierdo, en este caso consiste en determinar el punto en el que se encuentra el extremo de la pata a partir de los parámetros q_1, q_2 y q_3, tomando como referencia la configuración de la Figura 3.2 y la Figura 3.3. Las expresiones obtenidas se muestran en la Figura 3.4.

$$\begin{aligned} s &= \sqrt{l_2^2 + l_3^2 - 2 \cdot l_2 \cdot l_3 \cdot cos(q_3 + \tfrac{\pi}{2})} \\ \beta &= arccos(\tfrac{s^2 + l_2^2 - l_3^2}{2 \cdot s \cdot l_2}) \\ \alpha &= q_2 + \beta \\ p_z &= h_1 + s \cdot sin(\alpha) \\ p_y &= -(l_1 + s \cdot cos(\alpha)) \cdot cos(q_1) \\ p_x &= -p_y \cdot tan(q_1) \end{aligned} \tag{3.4}$$

A partir de las expresiones anteriores, es posible obtener el punto donde se encuentra el extremo de la pata si se conocen las posiciones angulares de sus articulaciones, pero solo para aquellas asociadas al lado izquierdo, en la Ecuación 3.5 se representan las ecuaciones correspondientes a la cinemática directa de las patas del lado derecho del robot.

$$\begin{aligned} s &= \sqrt{l_2^2 + l_3^2 - 2 \cdot l_2 \cdot l_3 \cdot cos(q_3 + \tfrac{\pi}{2})} \\ \beta &= arccos(\tfrac{s^2 + l_2^2 - l_3^2}{2 \cdot s \cdot l_2}) \\ \alpha &= q_2 + \beta \\ p_z &= -h_1 - s \cdot sin(\alpha) \\ p_y &= -(l_1 + s \cdot cos(\alpha)) \cdot cos(q_1) \\ p_x &= -p_y \cdot tan(q_1) \end{aligned} \tag{3.5}$$

3.3. Movimientos del robot

Empleando las ecuaciones que definen la relación entre la posición del extremo de las patas con las posiciones angulares de las mismas es posible definir las secuencias de movimientos que generan los distintos desplazamientos del robot. Estos desplazamientos deberán cumplir con la condición de mantener en todo momento tres de las seis patas apoyadas de modo que el robot se encuentre en equilibrio, tal y como se muestra en la Figura 2.12. Para realizar todos los movimientos el robot deberá dividir sus seis extremidades en dos grupos de tres patas, de forma que cada grupo contendrá las patas delantera y trasera de un lado del robot, junto con la pata intermedia del lado opuesto.

3.3.1. Movimiento de avance

Para realizar el movimiento de avance, uno de los grupos de patas del robot describirá una trayectoria lineal en la dirección opuesta a la del avance con sus extremos apoyados sobre el plano, empujando de este modo el cuerpo del robot hacia la dirección deseada. Al mismo tiempo, el otro conjunto de extremidades deberá avanzar en la misma dirección del avance para posicionarse en la configuración que permitirá a este grupo realizar el movimiento de empuje cuando el primer conjunto haya concluido el movimiento. Para que no se produzcan colisiones ni interferencias, el grupo de patas que realice el movimiento en la dirección de avance deberá elevar sus patas sobre el plano, evitando así generar fricciones con el plano que podrían desestabilizar al robot o dañar sus articulaciones. En la Figura 3.5 se puede observar una secuencia de configuraciones que representan este avance en la dirección frontal.

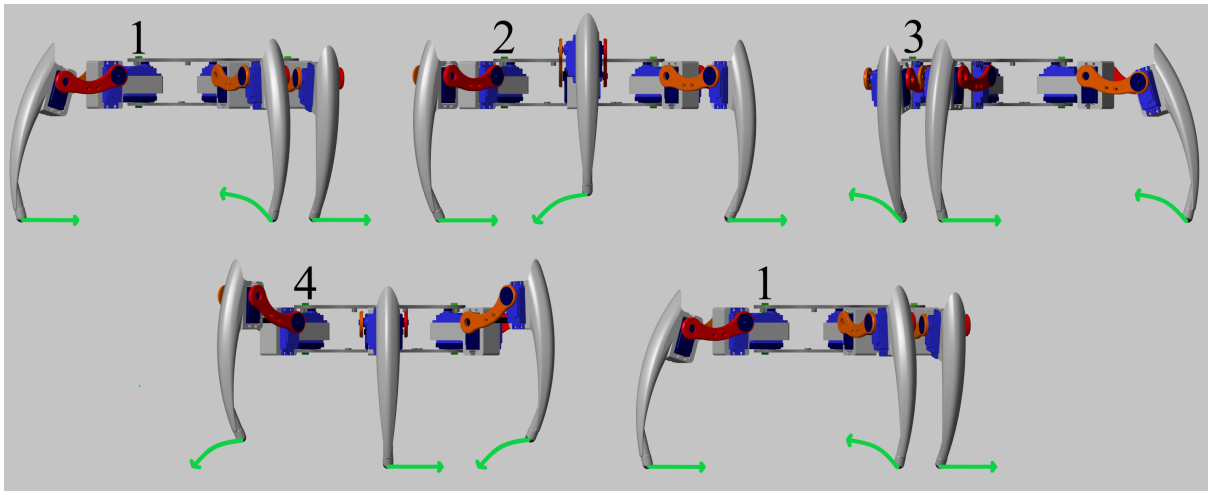


Figura 3.5: Secuencia de imágenes que describen el movimiento de avance.

En la figura anterior se ha representado la secuencia de movimientos realizados por las patas correspondientes al lado izquierdo del robot para producir un avance lineal del cuerpo. Se puede observar el movimiento que realiza cada pata de este lado del robot en el movimiento, en los pasos 1 y 2, dos de las patas empujan el cuerpo del robot hacia delante al describir con sus extremos una línea recta hacia atrás, mientras que la pata restante avanza sin hacer contacto con el suelo para poder continuar empujando el cuerpo del robot cuando las otras patas hayan completado su movimiento hacia atrás, instante en el que estas empezarán a llevar la pata al punto adelantado describiendo una arco elevado para no interferir con las patas que en ese instante se encontrarán apoyadas. No se han mostrado las patas del lado derecho en la misma imagen para evitar confusión, pero se encontrarán realizando la misma secuencia de movimientos, pero con un desfase de dos etapas, lo que asegurará que siempre haya tres patas sobre el plano para garantizar la estabilidad.

3.3.2. Movimiento de giro

El movimiento de giro comparte varios aspectos con el movimiento de avance, dado que cada grupo de patas debe realizar movimiento sincronizados, pero también debido a que uno de los grupos debe describir una trayectoria en la cual eleva los extremos de sus patas del plano para no producir fricción que interferiría con el otro grupo de patas, cuyos extremos permanecerían apoyados sobre el plano. La principal diferencia reside en que el movimiento que antes era lineal para empujar el cuerpo del robot en una dirección, ahora debe generar una rotación. Para ello,

las patas que se encuentran apoyadas deberán trazar un arco de circunferencia con centro en el centro de gravedad del robot y de amplitud igual a la del giro deseado. Estos movimientos se representan en la Figura 3.6, la cual se puede observar a continuación.

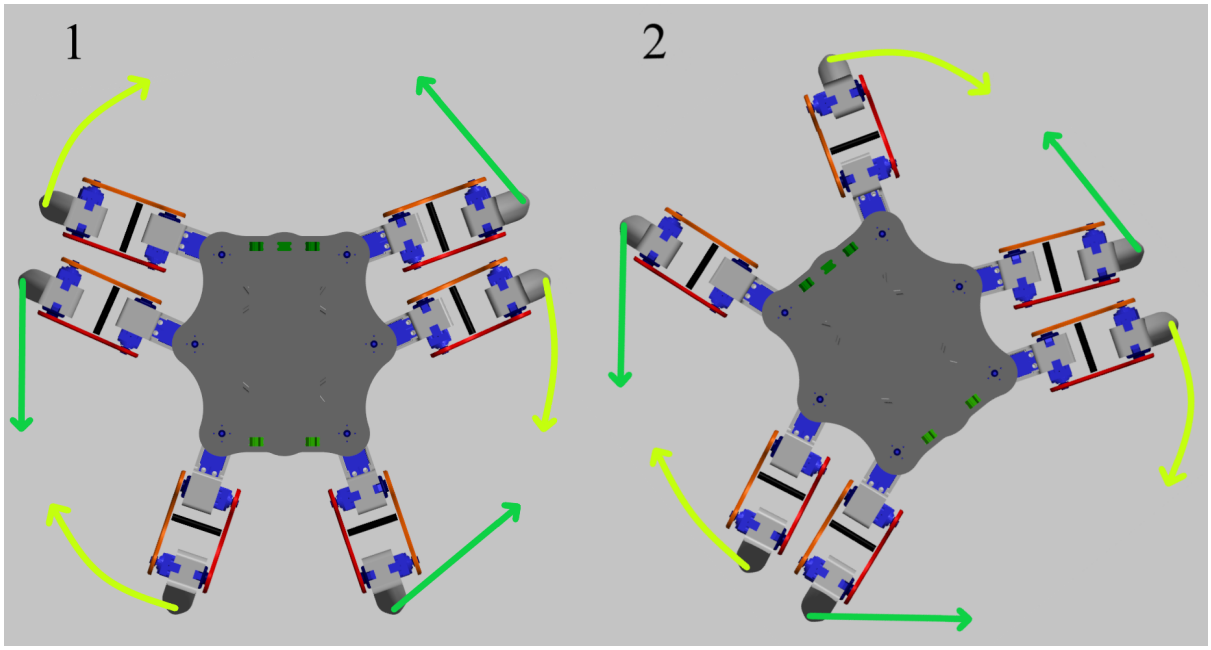


Figura 3.6: Configuraciones para realizar giros.

Sobre la imagen anterior se representan en color verde claro las trayectorias en forma de arco que llevarán a cabo las patas sobre el plano, con las que rotarán el cuerpo del robot mediante la coordinación de estos movimientos, es decir, los tres arcos descritos por los extremos de las patas comparten la amplitud del ángulo que los describe. Al mismo tiempo, las otras tres patas realizan movimientos elevándose, representados en verde oscuro en la imagen, para llevar el extremo de la pata al punto inicial del siguiente paso del giro.

4

Simulación

4.1. Montaje de la simulación

La simulación se ha llevado a cabo con Simulink Simscape Multibody, una herramienta de Matlab para simular sistemas mecánicos. En primer lugar se debe montar el modelo de la simulación, para ello se han empleado los archivos disponibles en Internet con el modelo tridimensional de las piezas del robot Antdroid[12], junto con el modelo del motor y el de la rueda que se une a su eje. Se puede observar el aspecto de estas piezas en el entorno de simulación en la Figura 4.1, la Figura 4.2 y la Figura 4.3.

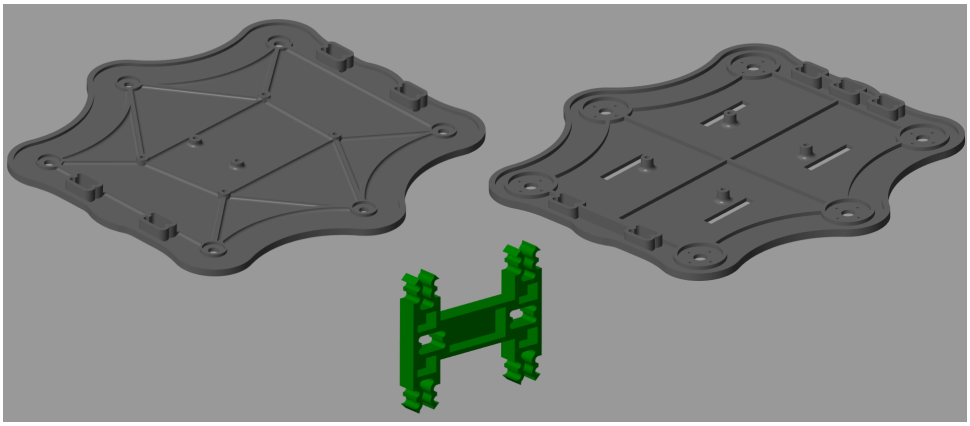


Figura 4.1: Modelos en Simscape de las piezas empleadas para el cuerpo del robot.

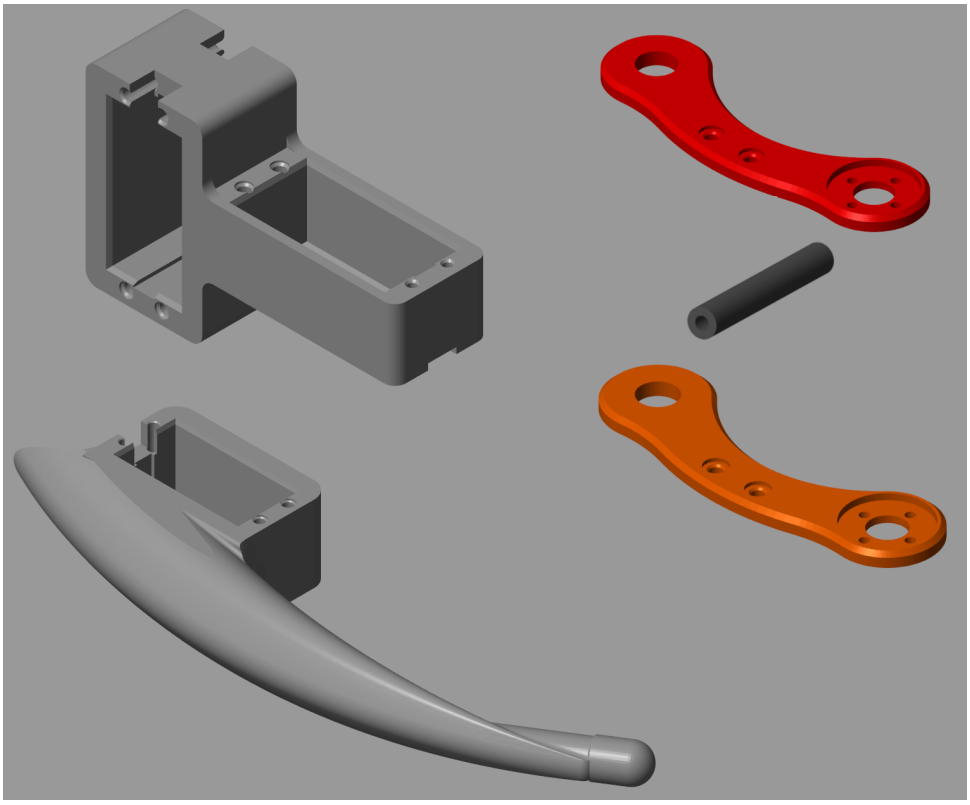


Figura 4.2: Modelos en Simscape de las piezas empleadas para las patas del robot.

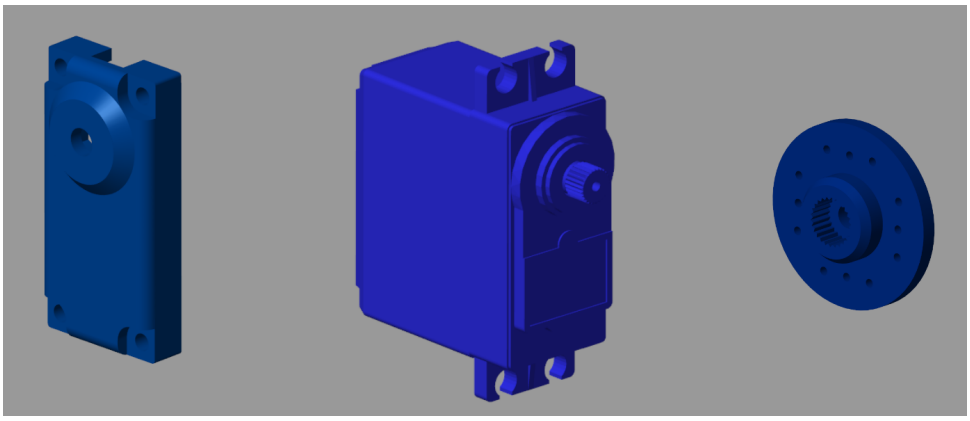
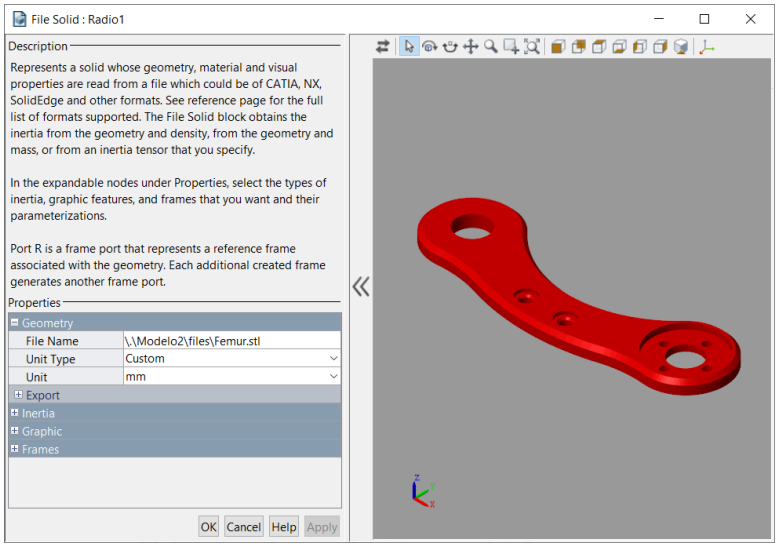


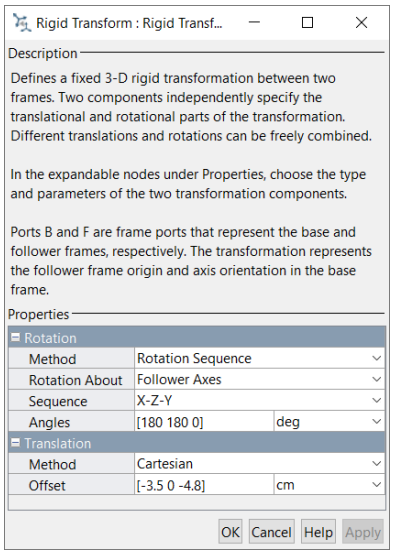
Figura 4.3: Modelos en Simscape del motor y de las piezas que se unen a él.

Para introducir estos modelos en Simulink se ha empleado el bloque *File Solid* de la librería Simscape Multibody, que permiten crear un objeto de la simulación a partir de un archivo con información tridimensional como se ve en la Figura 4.4a. Todas las componentes a excepción de los motores se han supuesto hechas de plástico ABS, dado que sería el material con el cual se imprimirían posteriormente. De este modo, partiendo la densidad de las piezas, Simscape calcula la masa de las mismas a partir de su geometría. Para los motores se obtuvo el dato de su masa

desde su hoja de especificaciones, y se introdujo directamente, de modo que Simscape calcula su densidad para la simulación. En este caso se disponía de la gran mayoría de las piezas con extensión *.stl*, la cual presenta el inconveniente de que no permite definir sistemas de referencia sobre la propia pieza. Por otra parte se han empleado bloques *Rigid Transform* para generar los desplazamientos y rotaciones necesarios para ensamblar las piezas de manera correcta. Este bloque permite realizar rotaciones alrededor de un único eje o realizar una secuencia de giros alrededor de una determinada combinación de ejes, lo que hace posible modificar completamente la orientación de una pieza con un único bloque. Adicionalmente también permite añadir un desplazamiento a lo largo de un eje o una serie de desplazamientos en los tres ejes cartesianos como se puede observar en la Figura 4.4b.



(a) Ejemplo de bloque *File Solid* empleado.



(b) Ejemplo de bloque *Rigid Transform* empleado.

Figura 4.4: Capturas de la interfaz de los bloques de Simulink Simscape Multibody.

El modelo completo creado emplea un gran número de los bloques indicados anteriormente para definir el aspecto y la geometría del robot, pero además de estos se requieren otros bloques. En primer lugar se necesitan los bloques *Solver Configuration*, encargado de definir una serie de ajustes para el correcto funcionamiento de la simulación; el bloque *World Frame*, que crea un sistema de coordenadas de referencia para toda la simulación, es decir, todos los sistemas de referencia que se empleen quedarán referenciados a este; y el bloque *Mechanism Configuration*, empleado para simular el efecto de la gravedad. También se han añadido un bloque *Infinite Plane*, que define un plano horizontal sobre el cual puede moverse el robot, y una serie de

bloques *Spacial Contact Force* que se emplean para simular los efectos del contacto entre los extremos de las patas del robot y el plano horizontal. Para simular dicho contacto es necesario que las piezas entre las cuales se lleva a cabo hayan exportado su geometría. El plano infinto la genera por defecto, pero los extremos de la patas del robot no, y dado que resultaría muy impreciso aproximar la geometría como un cono o un cilindro, se ha optado por crear una serie de pequeñas esferas de masa despreciable, la cuales se han situado en los extremos de dichas patas y exportar la geometría de estas, como se muestra en la Figura 4.5.

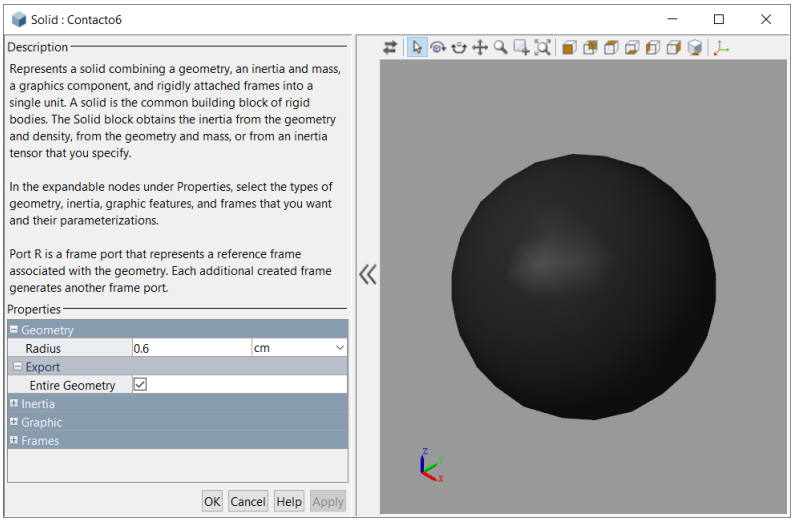
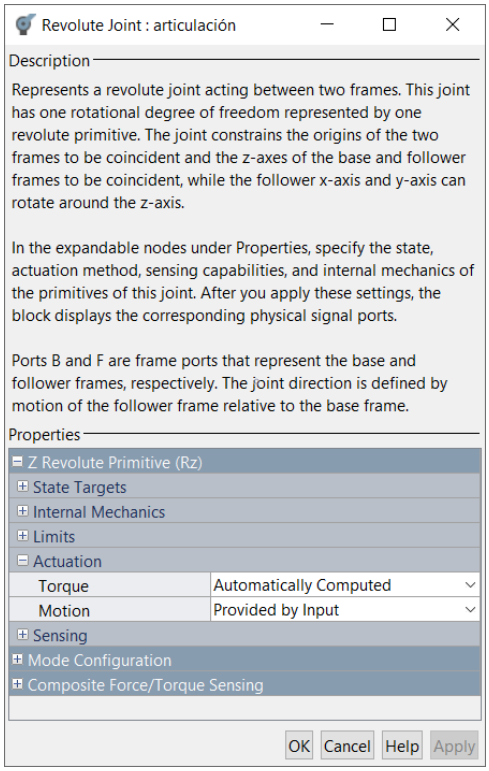


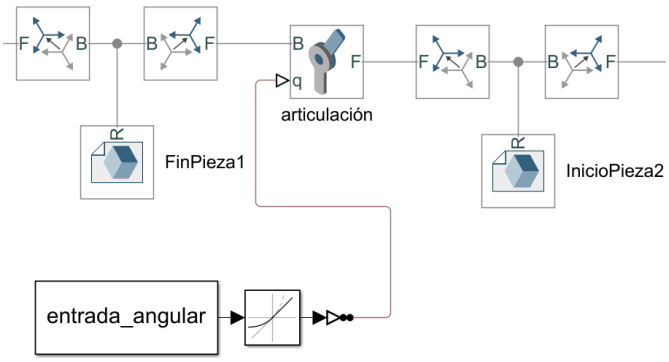
Figura 4.5: Bloque *Solid* con una pequeña esfera para el contacto.

Finalmente, resulta imprescindible incorporar los bloques que definen los grados de libertad del robot, los bloques correspondientes a las articulaciones. Para habilitar el movimiento del robot en el espacio se ha usado un bloque *Gimbal Joint*, que proporciona los tres grados de libertad de orientación, es decir la orientación en los tres ejes cartesianos, y un bloque *Cartesian Joint*, que proporciona los tres grados de libertad espaciales. Al tener estos dos bloques no actuados, el robot es libre de moverse y girar por el entorno de la simulación, y los movimientos que realice serán la consecuencia de las acciones de sus patas; no obstante, sí que se encuentran sensorizados, para poder llevar a cabo un análisis del comportamiento del robot. Para habilitar dichos movimientos cada pata incorpora tres bloques *Revolute Joint*, que simulan una articulación de revolución. Estas articulaciones sí que se encuentran actuadas, y se ha determinado que funcionarán por posición, es decir, la entrada será la posición angular que deberá tomar la articulación. Esto implica que el par requerido para el movimiento será generado de manera automática, lo cual no es necesariamente negativo, dado que ese es el funcionamiento de los servomotores digitales con los que se realizará la implementación. En la Figura 4.6a se puede

observar la configuración de los bloques *Revolute Joint* empleados. Cabe destacar también que estos bloques definen una articulación de rotación respecto al eje Z de los dos sistemas a los que está unida, un sistema actúa como base de la rotación (etiqueta B del bloque) y el otro como seguidor (follower en inglés, etiqueta F del bloque). Para asegurar que el movimiento se produzca de la manera esperada, suele ser necesario emplear bloques *Rigid Transform* para alinear los ejes alrededor de los cuales se desea que se produzca el giro, por este motivo las estructuras empleadas para llevar a cabo los giros son de la forma mostrada en la Figura 4.6b.



(a) Interfaz del bloque.



(b) Ejemplo de montaje con el bloque.

Figura 4.6: Imágenes del funcionamiento del bloque *Revolute Joint*.

Empleando todo lo anterior se ha construido el esquema de Simulink que incluye todas las componentes y articulaciones del robot, como se muestra en la Figura 4.7.

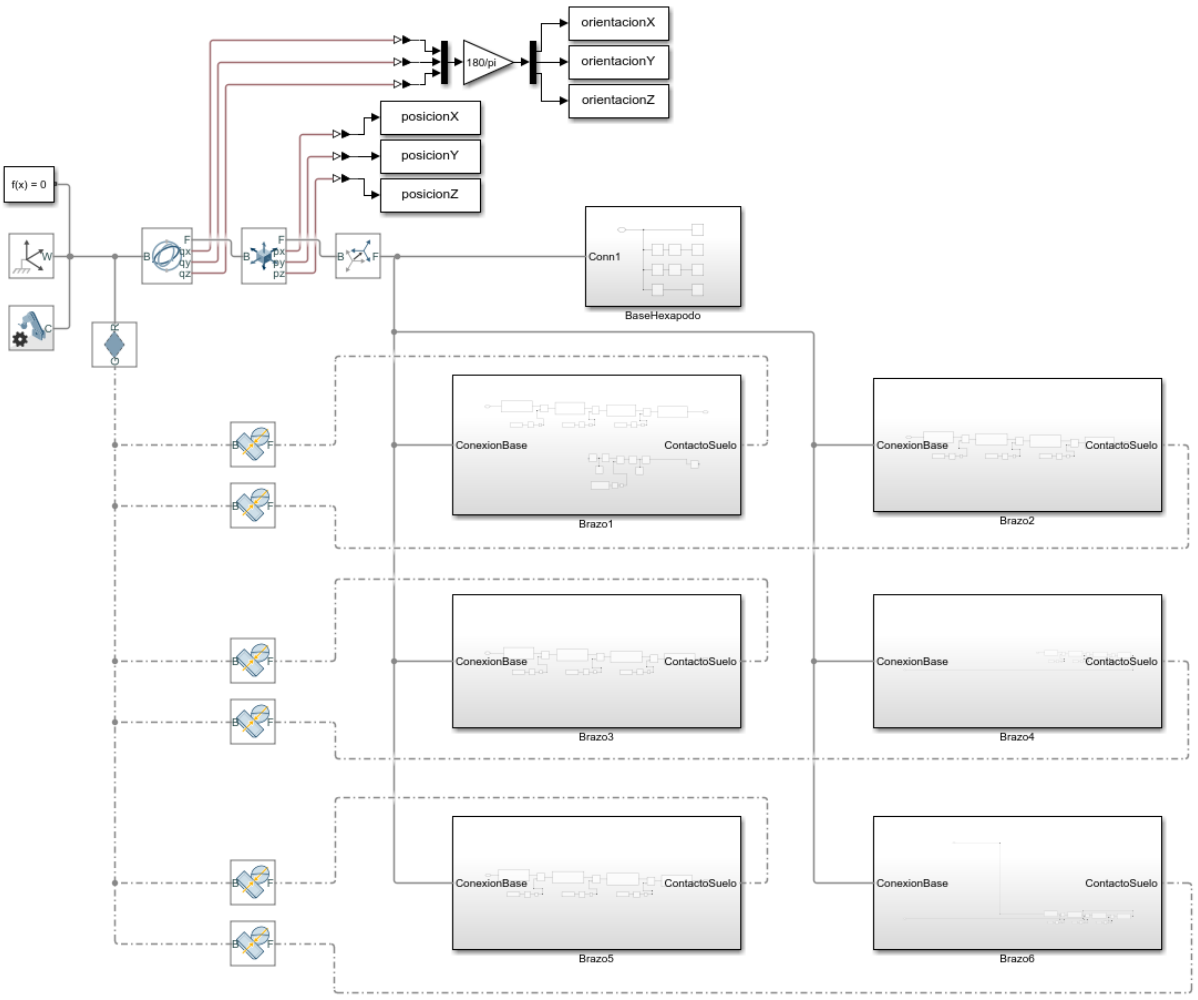
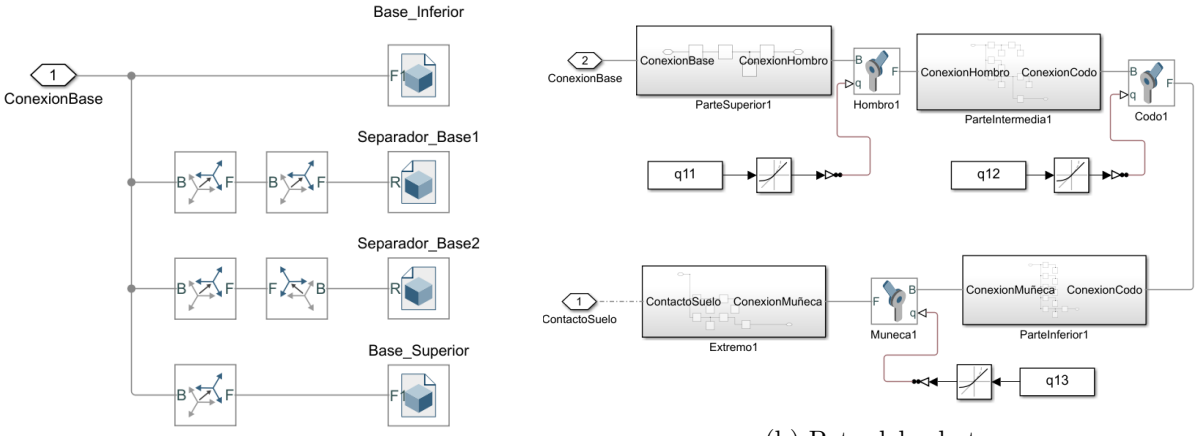


Figura 4.7: Esquema de Simulink para la simulación del robot hexápodo.

Este esquema ha sido dividido en una serie de subsistemas para aumentar su visibilidad, se han creado una serie de subsistemas para las patas y otro más para el cuerpo. Al inicio de todos estos esquemas se introduce el sistema de referencia central del robot, y a la salida de los correspondientes a las patas se encuentra la geometría de las esferas situadas en sus extremos. Se pueden observar estos subsistemas en la Figura 4.8. En el esquema correspondiente a las patas se pueden observar las articulaciones, las cuales han sido nombradas como las articulaciones del brazo humano para que resulte más sencillo imaginar el mecanismo. Estas articulaciones se encuentran actuadas, y para su entrada se ha empleado un bloque *From Workspace*, que proporciona los valores angulares de la misma con una matriz que representa el valor y el instante de tiempo asociado a ese valor, que se debe calcular previamente con Matlab. También se ha empleado un bloque *Rate Limiter* que suaviza los cambios angulares producidos, ya que se trata de escalones de posición, para simular el comportamiento real del servomotor.

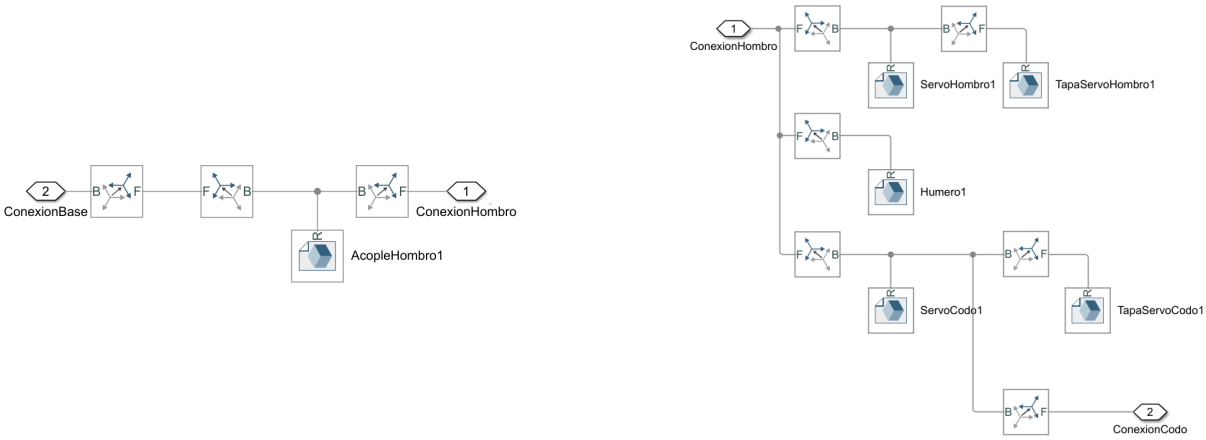


(a) Cuerpo del robot.

(b) Pata del robot.

Figura 4.8: Subsistemas de Simulink.

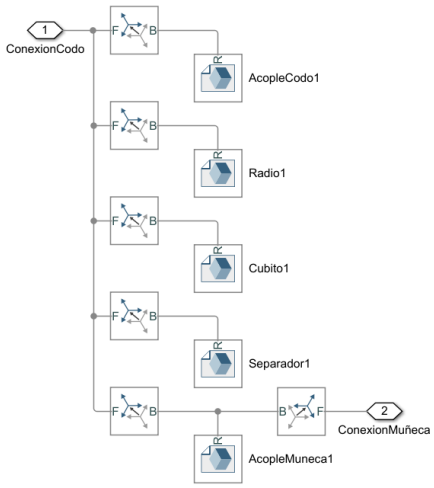
A su vez el esquema correspondiente a las patas ha sido subdivido en otros subsistemas, cada uno de ellos relaciona un conjunto de piezas que se mueven de manera solidaria. Cada uno de ellos presenta un sistema de referencia de entrada, que actúa como la base a la que el conjunto está sujeto, y otro que representa el sistema respecto al cual rotará el siguiente eslabón de la pata. Estos bloques son relativamente simples, dado que su única función es definir la geometría de un conjunto de piezas del robot en la simulación, como se puede apreciar en la Figura 4.9 y en la Figura 4.10.



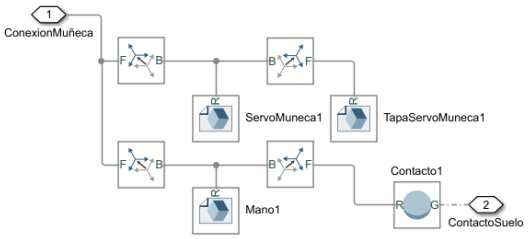
(a) Parte superior.

(b) Parte intermedia.

Figura 4.9: Subsistemas de una pata del robot.



(a) Parte inferior.

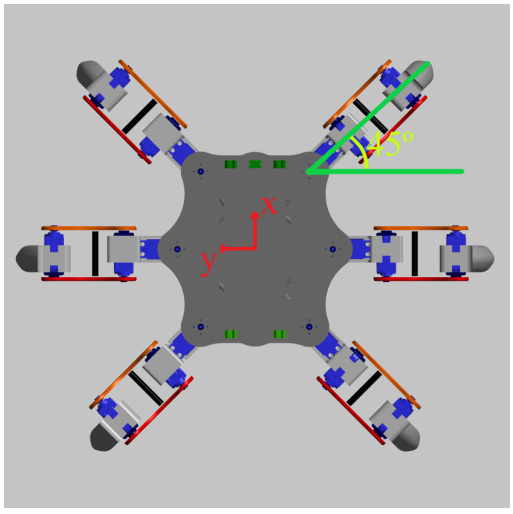


(b) Extremo.

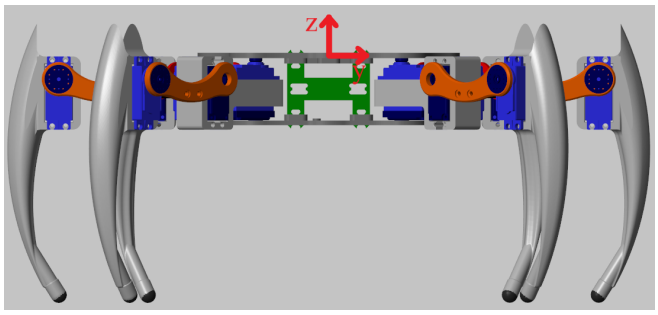
Figura 4.10: Subsistemas de una pata del robot.

4.2. Configuración del robot para la simulación

Una vez el robot se encuentra montado en la simulación debe llevarse a un estado donde pueda trabajar correctamente. Atendiendo al diseño, se ha optado por una configuración que separa ligeramente las patas delanteras y traseras de las intermedias para dar más rango de movimiento, y mantiene lás demás articulaciones en su posición base, es decir, una posición donde permitan su máximo movimiento en ambos sentidos. Esta posición se muestra en la Figura 4.11



(a) Vista superior.



(b) Vista frontal.

Figura 4.11: Configuración inicial del robot.

En la Figura 4.11 se puede observar la configuración donde las patas siguen una distribución aproximadamente radial, con las delanteras rotadas 45º y las traseras -45º, esto permite que el robot pueda tener un rango de movimientos razonable para sus patas, ya que, de otro modo, podrían producirse colisiones de las patas delanteras y traseras con las intermedias al moverse. Se ha comprobado manualmente en la simulación que, a partir de esta configuración, existe una separación entre extremidades suficiente como para permitir que dos patas consecutivas del mismo lado del robot no colisionen al realizar la primera de ellas un desplazamiento lineal hacia atrás y la segunda un movimiento lineal hacia delante de 6 cm de amplitud. Dado que amplitudes superiores provocan que patas consecutivas tomen posiciones demasiado cercanas, se ha optado por el paso de 12 cm de longitud, ya que asegura que no se produzcan colisiones entre las distintas extremidades.

En lo referente a los dos motores que se encuentran a lo largo de las patas, su posición base es aquella que permite la máxima amplitud de movimiento al siguiente eslabón, por ello es la empleada en el cálculo de la cinemática directa e inversa. En la realidad, aunque se construya el robot con esta libertad de movimientos, se deberá acotar su rango, ya que algunos componentes de una pata podrían chocar contra otros componentes de la misma. Por este motivo, la elección de la posición base de estas dos articulaciones resulta algo arbitraria. Finalmente se ha escogido aquella que presenta el rango máximo y que presenta la ventaja de que la forma de L de las patas en esta configuración facilita el cálculo del par necesario por los motores para soportar el peso del robot y desplazarlo.

4.3. Programación de la cinemática

Una vez completado el esquema se deben programar los movimientos, para ello se deben traducir las ecuaciones definidas en el Capítulo 3 a funciones que sean capaces de calcular una combinación de ángulos para alcanzar un punto con el extremo de la pata u obtener el punto en el que se encuentra dicho extremo a partir de las posiciones de las articulaciones en un instante según se requiera.

4.3.1. Programación de la cinemática inversa

La cinemática inversa de las extremidades permite calcular la combinación de ángulos necesaria para llevar el extremo de una pata del robot a un punto dado, y su programación consiste en la traducción de la Ecuación 3.1 y la Ecuación 3.2 a una función de Matlab, obteniéndose como resultado la función de la Figura 4.12.

```
function [q1, q2, q3] = inverseKinI(p, params)
% desde la base de cada brazo
    p.x = real(p.x);
    p.y = real(p.y);
    p.z = real(p.z);
    if p.x < 0.0000005 && p.x > -0.0000005
        p.x = 0.00000001;
    end
    if p.y < 0.0000005 && p.y > -0.0000005
        p.y = 0.00000001;
    end
    if p.z < 0.0000005 && p.z > -0.0000005
        p.z = 0.00000001;
    end
    q1 = atan2(p.x, -p.y); % - porque es el lado izq del robot
    r = sqrt(p.x^2 + p.y^2);
    s = sqrt((r - params.l1)^2 + (p.z - params.h1)^2);
    alfa = atan2(p.z - params.h1, r - params.l1);
    beta = acos((s^2 + params.l2^2 - params.l3^2) / (2 * s * params.l2));
    ganma = acos((params.l2^2 + params.l3^2 - s^2) / (2 * params.l2 * params.l3));
    q2 = alfa - beta;
    q3 = ganma - pi / 2;
end
```

Figura 4.12: Función para el cálculo de la cinemática inversa del lado izquierdo.

La función devuelve un vector con tres componentes, que son los tres ángulos necesarios para llevar la pata del robot al punto deseado, y toma como entrada dos estructuras, p es una estructura que contiene las coordenadas en los tres ejes cartesianos del punto donde se encuentra el extremo de la pata, mientras que *params* es una estructura que contiene las dimensiones del robot y se define al inicio del script. En primer lugar es necesario eliminar la parte imaginaria del punto, dado que es posible que Matlab genere una diminuta parte imaginaria a la hora de calcularlo, y si no se previene generará un error. También es necesario eliminar los valores de cero, ya que podrían generar indeterminaciones que terminarían en errores de cálculo o errores durante la simulación. Tras realizar estas comprobaciones, se calculan los valores de los ángulos atendiendo a las ecuaciones correspondientes.

Resulta de interés destacar el uso de la función *atan2* de Matlab. Esta función es llamada

arcotangente de dos parámetros, y presenta una ventaja respecto a la función arcotangente tradicional. La arcotangente tradicional analiza el signo de su argumento para determinar en qué cuadrante se encuentra el ángulo; no obstante, únicamente con esta información no es posible determinar completamente el cuadrante. Esto es debido a que existen ángulos en el primer y tercer cuadrantes que comparten arcotangente, al igual que ocurre para ángulos del segundo y cuarto cuadrantes. La arcotangente de dos parámetros analiza el signo de sus argumentos de entrada, para después calcular la tangente como el cociente de estos y finalmente el ángulo de esta. Gracias a este análisis, tras obtener el valor angular, se le añade π al resultado si el ángulo corresponde al segundo o tercer cuadrante, lo que permite que la función devuelva valores en el intervalo [-π, π], mientras que la función *atan* tradicional devuelve valores en el intervalo [-$\frac{\pi}{2}$, $\frac{\pi}{2}$], lo que puede resultar en errores para determinados cálculos[20]. En la Tabla 4.1 se puede ver la relación existente entre el signo del seno, del coseno y de la tangente del ángulo con el cuadrante al que pertenece, la función *atan2* emplea estas relaciones para proporcionar su resultado.

Cuadrante	Ángulo	Signo del seno	Signo del coseno	Signo de la tangente
I	$0 < \alpha < \pi/2$	+	+	+
II	$\pi/2 < \alpha < \pi$	+	-	-
III	$\pi < \alpha < 3{\cdot}\pi/2$	-	-	+
IV	$3{\cdot}\pi/2 < \alpha < 2{\cdot}\pi$	-	+	-

Tabla 4.1: Relación entre los signos del seno, del coseno y la tangente.

Existe también otra función prácticamente idéntica, pero que implementa la Ecuación 3.3, y calcula los ángulos para llevar una pata del lado derecho del robot al punto objetivo, esta función se encuentra junto al código de la simulacion en el Apéndice C.

4.3.2. Programación de la cinemática directa

De igual manera a la aplicada con la cinemática inversa, se han creado dos funciones para la cinemática directa, una para las patas de cada lado del robot. En la Figura 4.13 se puede observar la función para el lado izquierdo.

```
function p = directKinI(q, params)
% desde la base de cada brazo
    p.x = 0; p.y = 0;
    s = sqrt(params.l2^2 + params.l3^2 - 2 * params.l2 * params.l3 * cos(q(3) + pi / 2));
    beta = acos((s^2 + params.l2^2 - params.l3^2) / (2 * s * params.l2));
    alfa = q(2) + beta;
    p.z = params.hl + s * sin(alfa);
    p.y = -(params.l1 + s * cos(alfa)) * cos(q(1));
    p.x = -p.y * tan(q(1));
end
```

Figura 4.13: Función para el cálculo de la cinemática directa del lado izquierdo.

Por la configuración del sistema, los ángulos nunca aparecen con parte imaginaria, por lo que no es necesario extraer su parte real, y aunque tomasen valores de cero no se producirían errores en el cálculo, por lo que esta función simplemente debe implementar la Ecuación 3.4. A partir de un vector de entrada con los ángulos en los que se encuentran las articulaciones de la pata, y la estructura con las dimensiones del robot, se calcula y devuelve el punto donde se encuentra el extremo en forma de estructura. También ha sido necesaria otra función que implementa la cinemática directa del lado derecho, la única diferencia es que sus ecuaciones provienen de la Ecuación 3.5.

4.4. Cálculo de los movimientos

Una vez se dispone de las funciones que implementan la cinemática, se pueden calcular las secuencias de movimientos necesarias para provocar el desplazamiento del robot. Estos movimientos se han planteado de la manera siguiente: tres patas deben realizar un movimiento desde un punto adelantado hacia atrás a una altura constante, la del plano sobre el que se encuentre el robot; y las otras tres realizarán un movimiento parabólico desde un punto atrasado en el plano a otro adelantado, sin hacer contacto con el plano a excepción de en los momentos incial y final. El desplazamiento llevado a cabo por cada conjunto de patas será de la misma amplitud, para que la posición final de un paso pueda ser la inicial de la siguiente.

4.4.1. Movimientos de articulaciones en la simulación

El primer paso para implementar esta clase de movimientos en la simulación es la generación de las matrices de tiempos y valores angulares para las articulaciones de manera que el esquema

pueda interpretarla. La función *moveMotor* mostrada en la Figura 4.14 es la encargada de realizar esta tarea.

```
function v = moveMotor(qObj, Tini, T, Tfin)
% ~ moveAbsJ
    aux1 = [];
    aux2 = [];
    i = Tini;
    while i < Tfin
        aux1 = [aux1, i];
        aux2 = [aux2, real(qObj)];
        i = i + T;
    end
    v = [aux1' aux2'];
end
```

Figura 4.14: Función para generar el movimiento de una articulación.

Esta función crea la matriz de tiempos y posiciones necesaria. Emula la función encargada del movimiento absoluto de las articulaciones de un robot real. Recibe como entradas la posición objetivo de la articulación y también el tiempo actual de la simulación, el periodo empleado y el instante de tiempo en el que se desea que finalice el movimiento. De este modo se crea un bucle desde el instante inicial hasta el final con un incremente de valor igual al periodo que genera la secuencia de instantes de tiempo y al mismo tiempo el vector con la posición deseada. Todas las componentes de este segundo vector serán idénticas, ya que es el bloque *Rate Limiter* el encargado de suavizar los cambios bruscos en los ángulos en la simulación, como se explica en la Sección 4.1. Finalmente se realiza la transposición de los vectores, dado que deben tener este formato y se devuelve la matriz resultante de su concatenación.

A partir de este punto se han dividido todas las funciones en su versión para el lado derecho y el izquierdo, ya que presentan ligeras diferencias y deben llamar a distintas funciones para la cinemática. Debido a que son prácticamente iguales, de ahora en adelante se tratará únicamente con las funciones del lado izquierdo. La primera que se tratará será la función *moveJI*, cuyo código se muestra en la Figura 4.15.

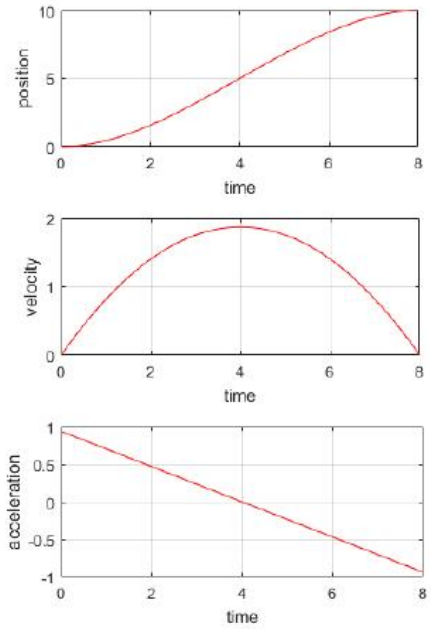
```
function v = moveJI(pObj, params, Tini, T, Tfin)
    [q1, q2, q3] = inverseKinI(pObj, params);
    v = moveMotor(q1, Tini, T, Tfin);
    v = [v, moveMotor(q2, Tini, T, Tfin)];
    v = [v, moveMotor(q3, Tini, T, Tfin)];
end
```

Figura 4.15: Función para llevar una pata a un punto.

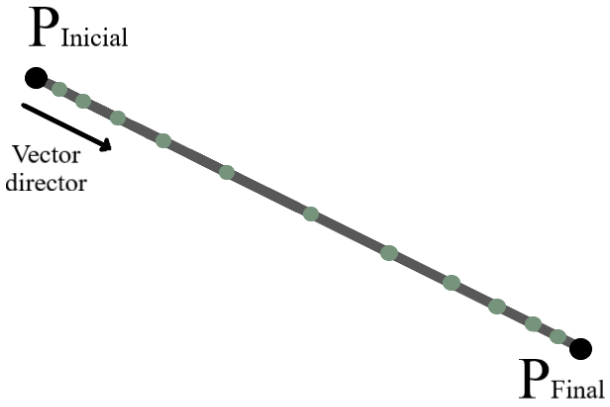
La función *moveMotor* explicada anteriormente realiza el movimiento cuando ya se dispone del ángulo objetivo al que se desea avanzar. Debido a la necesidad de mover las patas entre puntos se ha creado otra función que mueve las articulaciones de una pata para llevarla a un punto objetivo empleando la cinemática inversa. Esta función ha sido nombrada *moveJI* y emplea la cinemática inversa para hallar las posiciones objetivo, y entonces llama a la función *moveMotor* por cada articulación para realizar el desplazamiento. El resultado de estas llamadas se une en una única matriz de seis columnas, donde las columnas impares contendrán el tiempo de la simulación y las pares el ángulo en dicho instante para cada articulación de la pata, y así resulta posible devolver una única variable, la cual se volverá a dividir para enviar las señales a la simulación.

4.4.2. Movimiento lineal de las extremidades

La función *moveJI* presenta el inconveniente de que al modificar directamente la posición angular de las articulaciones para alcanzar el punto objetivo no se realiza ningún control de la trayectoria descrita en el espacio por el extremo de la pata. Esto podría provocar que la extremidad colisionase con algún objeto del entorno y no permite realizar movimientos que describan rectas en el espacio. Para solucionar este problema se ha implementado otra función, que emplea la cinemática directa para hallar el punto donde se encuentra la pata en el instante actual, y junto con el punto objetivo calcula la distancia y la dirección en la que debe llevarse a cabo el desplazamiento del extremo de la pata. Adicionalmente, se ha empleado una trayectoria cúbica para realizar este movimiento, esto se debe a que en la realidad no sería realista que el movimiento se realizase a velocidad constante, dado que eso supondría una aceleración muy elevada al inicio y al final de la trayectoria, lo cual podría dañar los motores. Esta trayectoria implica que la posición del extremo de la pata comenzará a moverse lentamente y alcanzará una velocidad máxima que empezará a reducirse cuando se acerque al objetivo, lo que asegura que los movimientos de los motores son suaves al inicio y al final del movimiento. Para implementarla se dividirá la trayectoria lineal que deberá seguir la pata en segmentos de distinta longitud, como se puede observar en la Figura 4.16b, que se recorrerán a través la función *moveJI*, y debido a la pequeña amplitud de estos movimientos se generará una trayectoria aproximadamente lineal en el espacio.



(a) Perfiles de la trayectoria.



(b) División del movimiento en el espacio.

Figura 4.16: Trayectoria cúbica.

De este modo, siendo la trayectoria cúbica $s(t)$, la posición en un instante dado, $P(t)$ se define como se muestra en la Ecuación 4.1.

$$P(t) = P_{Ini} + s(t) \cdot \frac{\overrightarrow{(P_{Fin} - P_{Ini})}}{\sqrt{P_{Fin}^2 - P_{Ini}^2}} \tag{4.1}$$

Y para definir la trayectoria cúbica se debe considerar que se desea que la posición inicial sea nula y la final sea igual a la distancia que se desea recorrer, mientras que la velocidad debe ser nula al principio y al final del movimiento, como se puede observar en la Figura 4.16a. De este modo se obtienen las expresiones de la Ecuación 4.2.

$$\begin{aligned} s(t) &= a \cdot t^3 + b \cdot t^2 + c \cdot t + d \\ \dot{s}(t) &= 3 \cdot a \cdot t^2 + 2 \cdot b \cdot t + c \end{aligned} \tag{4.2}$$

Si se denomina T al instante final de la trayectoria, se pueden expresar los valores de los

parámetros a, b, c y d de la forma representada en la Ecuación 4.3.

$$a = -\frac{2 \cdot \sqrt{P_{Fin}^2 - P_{Ini}^2}}{T^3} \qquad b = \frac{3 \cdot \sqrt{P_{Fin}^2 - P_{Ini}^2}}{T^2} \qquad c = 0 \qquad d = 0 \tag{4.3}$$

Al implementar las ecuaciones de la trayectoria cúbica en la función del movimiento lineal es posible realizar el desplazamiento a lo largo de una recta cumpliendo con la trayectoria definida, como se muestra en la Figura 4.17.

```
function v = moveLI(q0, pObj, params, Tini, T, Tfin)
    p0 = directKinI(q0, params);
    distEuc = sqrt((pObj.x - p0.x)^2 + (pObj.y - p0.y)^2 + (pObj.z - p0.z)^2);
    if distEuc < 0.0000005 && distEuc > -0.0000005
        vUnitario.x = 0;
        vUnitario.y = 0;
        vUnitario.z = 0;
    else
        vUnitario.x = (pObj.x - p0.x) / distEuc;
        vUnitario.y = (pObj.y - p0.y) / distEuc;
        vUnitario.z = (pObj.z - p0.z) / distEuc;
    end

    a = -2 * distEuc / (Tfin - Tini)^3; % Trayectoria cúbica
    b = 3 * distEuc / (Tfin - Tini)^2;

    iteracionFinal = (Tfin - Tini) / T;
    Tactual = Tini;
    v = [];
    i = 1;
    while i <= iteracionFinal
        t = i * T;
        s = a * t^3 + b * t^2;
        p.x = s * vUnitario.x + p0.x;
        p.y = s * vUnitario.y + p0.y;
        p.z = s * vUnitario.z + p0.z;
        v = [v; moveJI(p, params, Tactual, T, Tactual + T)];
        i = i + 1;
        Tactual = Tactual + T;
    end
end
```

Figura 4.17: Función para el desplazamiento lineal del extremo de la pata.

Para construir esta función se deben hallar el punto inicial, a partir de la configuración inicial de las articulaciones y la cinemática directa, y la distancia euclídea entre los puntos inical y final, es decir la distancia en línea recta entre ellos. En caso de que la distancia sea nula se debe anular el vector director del movimiento, ya que de otra manera podrían aparecer indeterminaciones. Esta situación se dará cuando se desea que unas patas se muevan mientras que el resto deba permanecer en el sitio. Tras obtener el vector director del movimiento se

calcularán los parámetros a y b de la trayectoria, dado que c y d serán nulos. Finalmente se llevará a cabo un bucle que partirá del instante actual de tiempo y terminará cuando se alcance el tiempo final, con un incremento igual al periodo de la simulación, que calculará el tiempo de la trayectoria en cada instante, y con él calculará el valor de la misma y la siguiente posición a alcanzar, para así llamar a la función *moveJI* para que lleve a cabo el movimiento.

4.4.3. Movimiento parabólico de las extremidades

Con la función *moveLI* y su contraparte para el lado derecho, ya se podrían realizar movimientos lineales del cuerpo del robot en la simulación, y se podría crear una secuencia para llevar a cabo movimientos de avance en la cual tres patas describirían una recta hacia atrás en el plano a la vez que las otras tres patas describirían una secuencia de movimientos en los que levantarían la pata del plano, la llevarían hacia delante y la volverían a apoyar en este. Por otra parte, este movimiento resulta muy forzado, y además implicaría acciones muy bruscas para los motores, por lo que se ha de crear otro movimiento que describa una parábola, es decir, describirá una recta en el plano XY y una parábola en el plano Z. De este modo, el extremo de la pata se levantará del suelo a la vez que avanza hacia el punto objetivo describiendo una parábola como la mostrada en la Figura 4.18.

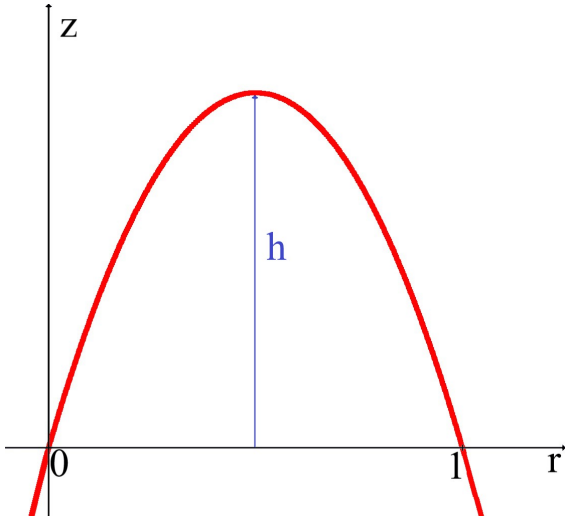


Figura 4.18: Parábola a implementar.

Para llevar a cabo esta tarea se debe comenzar por analizar el movimiento en parábola. La parábola que se pretende obtener se elevará una altura máxima determinada por una constante

h en el eje Z, y recorrerá una distancia en el plano XY llamada r determinada por la diferencia en estas coordenadas de los puntos origen y destino. Cabe destacar que se supondrá que estos dos puntos tendrán una componente Z idéntica, dado que se pueden producir comportamientos indeseados si esto no se cumple. Aunque en la Figura 4.18 el segundo punto de corte con la horizontal se encuentre en 1, se implementará de modo que el término que llevará a cabo la parábola obtenga valores entre 0 y 1, es decir, no necesariamente será esta unidad la recorrida en todos los casos. A continuación, en la Ecuación 4.4 se puede observar la ecuación que define de una parábola genérica y las condiciones que debe cumplir la parábola deseada.

$$\begin{gathered} z(r) = a^2 \cdot r + b \cdot r + c \\ z(0) = 0 \qquad z(1) = 0 \qquad \dot{z}(\tfrac{1}{2}) = h \end{gathered} \tag{4.4}$$

A partir de las expresiones anteriores, se pueden obtener los valores de sus parámetros y la expresión que tomará finalmente la parábola, como se muestra en la Ecuación 4.5.

$$\begin{gathered} a = -4 \cdot h \qquad b = 4 \cdot h \qquad c = 0 \\ z(r) = -4 \cdot h \cdot r^2 + 4 \cdot h \cdot r \end{gathered} \tag{4.5}$$

Para llevar a cabo la parábola definida, r debe tomar valores entre 0 y 1, para ello se definirá como una expresión dependiente de la distancia a recorrer en el plano XY y de una trayectoria cúbica que, de igual manera a la de la Figura 4.16, toma valores entre 0 y la distancia total que se desea recorrer. La variable que contendrá el valor de dicha trayectoria cúbica será denominado s. De este modo se obtiene la expresión mostrada en la Ecuación 4.6 de la parábola en función del valor en cada instante de la trayectoria cúbica.

$$\begin{gathered} r(t) = \tfrac{s(t)}{s_{max}} \\ r(0) = 0 \qquad r(T) = 1 \\ z(t) = -4 \cdot h \cdot \left(\tfrac{s(t)}{s_{max}}\right)^2 + 4 \cdot h \cdot \tfrac{s(t)}{s_{max}} \end{gathered} \tag{4.6}$$

Empleando la relación entre r y s, junto con la declaración de la constante h, que tomará valor de 3 cm, dado que es suficiente para apreciar el movimiento, pero no provoca una elevación excesiva, que resultaría más brusca para los motores, se puede definir el movimiento parabólico en el eje Z como parte de la trayectoria cúbica y se puede calcular la secuencia de posiciones de las articulaciones para llevar a cabo el movimiento, lo que se realiza en la función *moveCI* representada en la Figura 4.19.

```
function v = moveCI(q0, pObj, h, params, Tini, T, Tfin)
    p0 = directKinI(q0, params);
    dist = sqrt((pObj.x - p0.x)^2 + (pObj.y - p0.y)^2);
    vUnitario.x = (pObj.x - p0.x) / dist;
    vUnitario.y = (pObj.y - p0.y) / dist;

    a = -2 * dist / (Tfin - Tini)^3; % Trayectoria cúbica
    b = 3 * dist / (Tfin - Tini)^2;

    A = - 4 * h;
    B = - A;

    iteracionFinal = (Tfin - Tini) / T;
    Tactual = Tini;
    v = [];
    i = 1;
    while i <= iteracionFinal
        t = i * T;
        s = a * t^3 + b * t^2;

        p.x = s * vUnitario.x + p0.x;
        p.y = s * vUnitario.y + p0.y;
        p.z = p0.z - (A * (s / dist)^2 + B * s / dist);

        v = [v; moveJI(p, params, Tactual, T, Tactual + T)];
        i = i + 1;
        Tactual = Tactual + T;
    end
end
```

Figura 4.19: Función para el desplazamiento parabólico del extremo de la pata.

La función anterior también calcula la distancia entre el punto final e inicial, al igual que se realiza en la función *moveLI*, pero en este caso se trata únicamente de la distancia en el plano XY, puesto que se supone que la componente Z de ambos puntos será igual. Tras obtener el vector director del movimiento, se han de calcular los valores de los parámetros correspondientes a la trayectoria cúbica, a y b, pero también los de la parábola, A y B. Para concluir, se realiza el mismo bucle que se realiza en la función para el movimiento lineal, solo que en este caso el cálculo de la componente Z del siguiente punto a alcanzar se obtiene como la coordenada Z del punto inicial menos el valor actual de la parábola. Esto se debe al sistema de referencia de las

patas del lado izquierdo, donde los valores positivos en el eje Z implican la distancia que separa el extremo de la pata del cuerpo, y si se incrementase aún más esta distancia, la pata ejercería más fuerza contra el suelo, como tratando de repelerlo, cuando el objetivo es que se eleve del plano. Por este motivo el valor de la parábola debe restarse, mientras que para el lado derecho se sumará.

4.4.4. Avance lineal

Con las funciones *moveLI* y *moveCI*, junto con su contraparte para las extremidades del lado derecho del robot, *moveLD* y *moveCD* es posible llevar a cabo la secuencia que haga avanzar al robot en línea recta. Esta secuencia se realizará de la manera descrita en la Subsección 3.3.1, con unas fases tal y como se pueden observar en la Figura 4.20 representada a continuación.

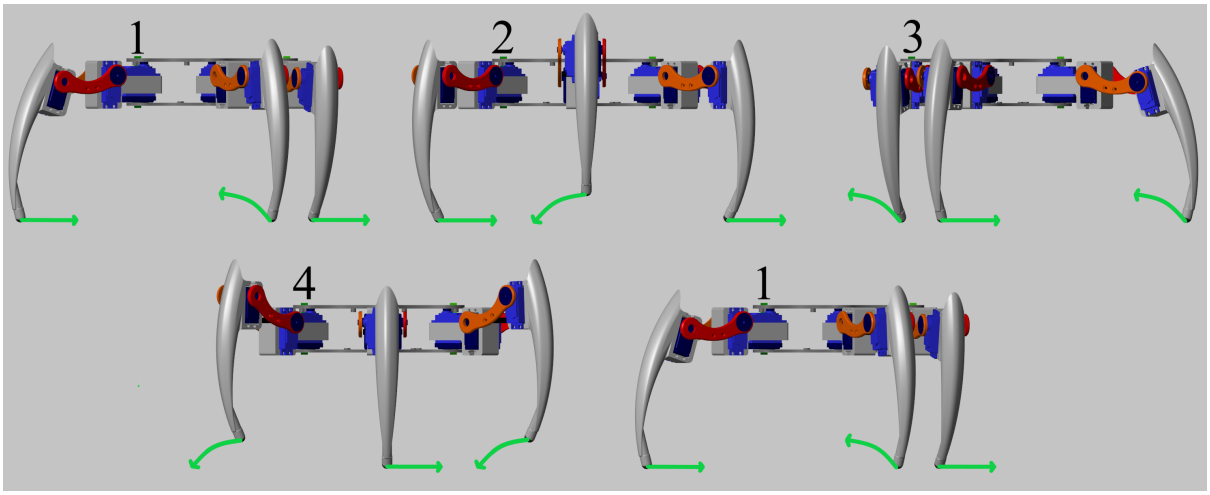


Figura 4.20: Secuencia de imágenes que describen el movimiento de avance.

Debido a la configuración del robot, es posible que realice desplazamientos en todas las direcciones del plano horizontal sin necesidad de girar, aplicando la misma secuencia de movimientos, solo que la trayectoria recta que realicen las patas deberá estar orientada en la dirección deseada. Para proporcionar esta libertad de movimiento han sido definidos tres vectores de estructuras que representan de puntos en el espacio y un ángulo, con el cual se obtendrá el vector director del movimiento. El primer vector contiene los puntos en los que se encuentran los extremos de las patas en la configuración inicial, es decir, los puntos en la configuración de la Figura 4.11, y su valor será constante. Empleando el valor del ángulo llamado *direccion*, cuyo valor se puede apreciar en la Figura 4.21, se puede obtener el vector director de la dirección de avance deseada.

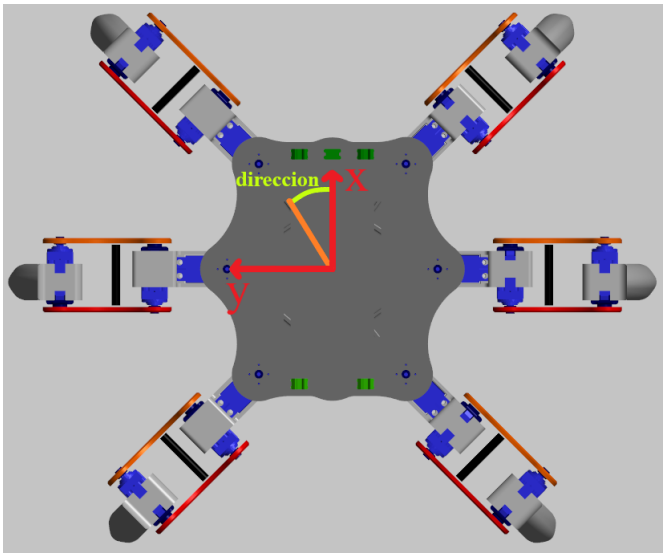


Figura 4.21: Representación del valor de *direccion* sobre el robot.

Empleando este vector director, que se representará como una estructura con dos componentes, junto con la constate *avance*, que tomará valor 6 cm como se ha explicado en la Sección 4.2, se obtendrán los valores de los otros dos vectores de puntos. Este código se muestra en la Figura 4.22

```
direccion = 0;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));

%% Posición base
p0 = [struct('x',  88.3883, 'y', -88.3883, 'z',  alturaBase);
      struct('x',  88.3883, 'y', -88.3883, 'z', -alturaBase);
      struct('x',        0, 'y',     -125, 'z',  alturaBase);
      struct('x',        0, 'y',     -125, 'z', -alturaBase);
      struct('x', -88.3883, 'y', -88.3883, 'z',  alturaBase);
      struct('x', -88.3883, 'y', -88.3883, 'z', -alturaBase)];
```

(a) Inicialización de *direccion* y $p0$.

```
function [p1, p2] = puntosAvance(p0, avance, vectorDireccion)
    p1 = [struct('x', p0(1).x + avance * vectorDireccion.x, 'y', p0(1).y - avance * vectorDireccion.y, 'z', p0(1).z);
          struct('x', p0(2).x - avance * vectorDireccion.x, 'y', p0(2).y - avance * vectorDireccion.y, 'z', p0(2).z);
          struct('x', p0(3).x - avance * vectorDireccion.x, 'y', p0(3).y + avance * vectorDireccion.y, 'z', p0(3).z);
          struct('x', p0(4).x + avance * vectorDireccion.x, 'y', p0(4).y + avance * vectorDireccion.y, 'z', p0(4).z);
          struct('x', p0(5).x + avance * vectorDireccion.x, 'y', p0(5).y - avance * vectorDireccion.y, 'z', p0(5).z);
          struct('x', p0(6).x - avance * vectorDireccion.x, 'y', p0(6).y - avance * vectorDireccion.y, 'z', p0(6).z)];

    p2 = [struct('x', p0(1).x - avance * vectorDireccion.x, 'y', p0(1).y + avance * vectorDireccion.y, 'z', p0(1).z);
          struct('x', p0(2).x + avance * vectorDireccion.x, 'y', p0(2).y + avance * vectorDireccion.y, 'z', p0(2).z);
          struct('x', p0(3).x + avance * vectorDireccion.x, 'y', p0(3).y - avance * vectorDireccion.y, 'z', p0(3).z);
          struct('x', p0(4).x - avance * vectorDireccion.x, 'y', p0(4).y - avance * vectorDireccion.y, 'z', p0(4).z);
          struct('x', p0(5).x - avance * vectorDireccion.x, 'y', p0(5).y + avance * vectorDireccion.y, 'z', p0(5).z);
          struct('x', p0(6).x + avance * vectorDireccion.x, 'y', p0(6).y + avance * vectorDireccion.y, 'z', p0(6).z)];
end
```

(b) Función para el cálculo de $p1$ y $p2$.

Figura 4.22: Código para la obtención de los puntos.

El código anterior se ha empleado para la inicialización y cómputo de los puntos para los movimientos. Una vez definido *direccion* se calcula *vectorDireccion* en base al sistema de referencia del robot, de modo que el ángulo que forme con el eje X sea *direccion*, como se puede observar en la Figura 4.21. Empleando este vector director se calculan los puntos $p1$ y $p2$, de modo que en $p1$ las patas 1, 4 y 5 se encuentren adelantadas en la dirección del vector la distancia deseada, mientras que las patas 2, 3 y 6 se encuentren atrasadas en la dirección del movimiento. Por su parte, $p2$ tendrá las patas 1, 4 y 5 atrasadas y las 2, 3 y 6 adelantadas. Se puede considerar que $p1$ corresponde con el paso 1 de la Figura 4.20 y $p2$ con el paso 3 de la misma figura. Una vez se dispone de los puntos, es posible usar las funciones *moveLI* y *moveCI*, junto con *moveLD* y *moveCD*, para hacer avanzar al robot en la dirección deseada.

4.4.5. Movimiento de descripción de un arco de las extremidades

Pese a que un robot hexápodo pueda aprovechar su configuración con patas para desplazarse sin tener que orientarse previamente, resulta conveniente disponer de la posibilidad de corregir la orientación. Para ello se debe implementar una funcionalidad que permita que el robot gire alrededor de su propio eje Z. Este movimiento se debe implementar de manera similar al avance, tres de las patas describirán una trayectoria de arco en el plano mientras que las otras tres se elevan y se desplazan hacia el punto inicial del siguiente movimiento con la función para el movimiento parabólico.

Esta tarea se dividirá en dos funciones, una de ellas encargada de obtener los puntos para el giro y otra que realizará el movimiento. En primer lugar se deben obtener los puntos inicial y final del giro, que vendrán determinados por la configuración inicial y el ángulo que se desee girar el cuerpo, ángulo que será denominado *ang* y que se puede observar en la Figura 4.23.

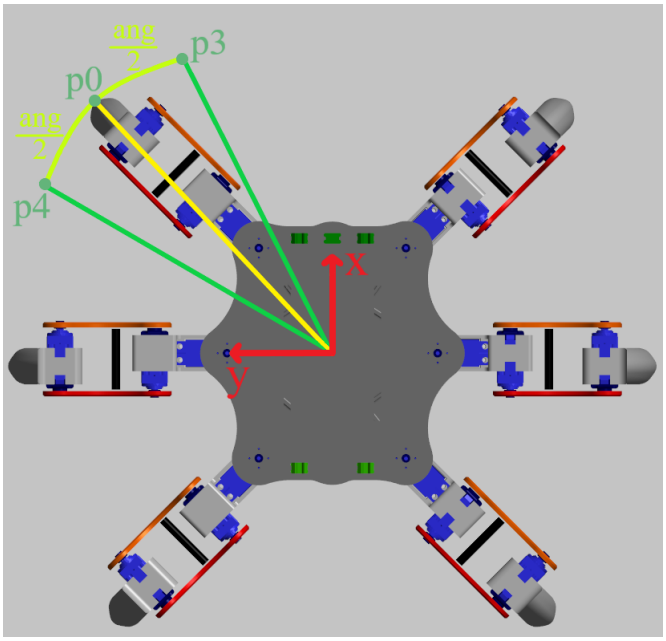


Figura 4.23: Representación de *ang* y los puntos *p*3 y *p*4 para la pata 1.

En la imagen anterior se puede apreciar la metodología usada para obtener los puntos *p*3 y *p*4. Desde el extremo de cada pata en la posición base se traza una recta hasta el sistema de coordenadas del cuerpo del robot, y se calcula el punto final de dicha línea si estuviese inclinado un ángulo igual a su posición anterior menos la mitad del ángulo del giro para obtener *p*3, y más la mitad para obtener *p*4.

Para obtener los puntos *p*3 y *p*4, desde la posición base de cada pata se debe pasar al sistema de referencia del robot, ya que el giro se produce sobre el eje Z de este sistema, para ello se han empleado unas variables constantes que relacionan el origen de cada extremidad con el sistema de referencia central, que se muestran en la Figura 4.24a. Una vez en este sistema se obtiene el ángulo que forma el segmento que une el origen con el extremo de la pata con el eje Y de este sistema, y a partir de este ángulo junto con el valor de *ang* se obtiene el ángulo que formaría dicho segmento para *p*3 y *p*4. Finalmente, al ser conocida la longitud del segmento, se obtienen las coordenadas de los puntos respecto a la base del robot, y empleando de nuevo las coordenadas del origen de su pata se vuelve a calcular la posición de los puntos, pero en este caso para su propio sistema de referencia. La función que realiza este cálculo se puede apreciar en la Figura 4.24b. Cabe destacar que la coordenada Z de los puntos será la misma que se emplee para la posición base y que existirá una contraparte de la función *giroI* para el lado derecho, que será llamada *giroD*.

```
origenPata1 = struct('x',  77.8, 'y',  54.5);
origenPata2 = struct('x',  77.8, 'y', -54.5);
origenPata3 = struct('x',     0, 'y',  78.0);
origenPata4 = struct('x',     0, 'y', -78.0);
origenPata5 = struct('x', -77.8, 'y',  54.5);
origenPata6 = struct('x', -77.8, 'y', -54.5);
```

(a) Inicialización de los orígenes de cada pata.

```
function [p3, p4] = giroI(p0, ang, origen)
    posOrigen0.x = origen.x + p0.x;
    posOrigen0.y = origen.y - p0.y;

    r = sqrt(posOrigen0.x^2 + posOrigen0.y^2);
    alfa0 = atan2(posOrigen0.x, posOrigen0.y);

    alfa3 = alfa0 - ang / 2;
    posOrigen3.x = r * sin(alfa3);
    posOrigen3.y = r * cos(alfa3);
    p3.x = posOrigen3.x - origen.x;
    p3.y = origen.y - posOrigen3.y;
    p3.z = p0.z;

    alfa4 = alfa0 + ang / 2;
    posOrigen4.x = r * sin(alfa4);
    posOrigen4.y = r * cos(alfa4);
    p4.x = posOrigen4.x - origen.x;
    p4.y = origen.y - posOrigen4.y;
    p4.z = p0.z;
end
```

(b) Función para el cálculo de $p3$ y $p4$.

Figura 4.24: Código para la obtención de los puntos.

Una vez se dispone de los puntos que actuarán como extremos del arco se puede expresar la función que realizará dicha trayectoria. Según el sentido del giro deseado, el arco se realizará de $p3$ a $p4$ o viceversa, pero también se puede definir empleando solo uno de los puntos y el ángulo que se desea girar. Se implementará este último método ya que supone menos cálculos, puesto que de otro modo debería calcularse este ángulo a partir de los puntos inicial y final del arco. El código de la función empleada para realizar este movimiento se puede observar en la Figura 4.25.

```
function v = moveGI(q0, ang, origen, params, Tini, T, Tfin)
    posPata0 = directKinI(q0, params);
    posOrigen0.x = origen.x + posPata0.x;
    posOrigen0.y = origen.y - posPata0.y;

    r = sqrt(posOrigen0.x^2 + posOrigen0.y^2);
    alfa0 = atan2(posOrigen0.x, posOrigen0.y);

    a = -2 * ang / (Tfin - Tini)^3; % Trayectoria cúbica
    b = 3 * ang / (Tfin - Tini)^2;

    iteracionFinal = (Tfin - Tini) / T;
    Tactual = Tini;
    v = [];
    i = 1;
    while i <= iteracionFinal
        t = i * T;
        s = a * t^3 + b * t^2;

        alfa = alfa0 + s;
        posOrigen.x = r * sin(alfa);
        posOrigen.y = r * cos(alfa);

        posPata.x = posOrigen.x - origen.x;
        posPata.y = origen.y - posOrigen.y;  % -(posOrigen.y - origen.y);
        posPata.z = posPata0.z;

        v = [v; moveJI(posPata, params, Tactual, T, Tactual + T)];
        i = i + 1;
        Tactual = Tactual + T;
    end
end
```

Figura 4.25: Código para la función que realiza el movimiento en arco.

Para el movimiento de arco de circunferencia se ha empleado la trayectoria cúbica para obtener la evolución del ángulo que define el arco. A partir de la posición inicial del extremo de la pata, obtenido con la cinemática directa, se obtiene el segmento que lo une con el origen, al igual que en la función *giroI*, y el ángulo que este segmento forma con el eje Y del sistema de referencia de la base del robot, el cual se denominará $\alpha 0$. Una vez se dispone de estos parámetros junto con las constantes de la trayectoria cúbica, se realiza el bucle que mueve las articulaciones. En cada instante se toma el valor actual de la trayectoria y se le suma a $\alpha 0$ para encontrar la inclinación correspondiente a la iteración actual. Empleando este nuevo ángulo se obtiene el punto a alcanzar respecto al sistema de referencia del cuerpo y finalmente se convierte al sistema de referencia del origen de la pata, del mismo que se ha realizado previamente en la función de la Figura 4.24b.

4.4.6. Giro sobre sí mismo

Combinando las funciones *moveCI* y *moveGI*, y sus contrapartes para las patas del lado derecho, es posible que el robot realice giros sobre su propio cuerpo. Estos giros se realizarán de la forma descrita en la Subsección 3.3.2, donde tres patas describirán un arco sobre el plano que rotará el cuerpo del robot, y las otras tres emplearán el movimiento parabólico para ir al siguiente punto.

Un dato importante es que el ángulo más empleado para realizar los giros es 30^{o}, con este ángulo se asegura que las patas no colisionen entre sí, y permite realizar giros de 90^{o} y 180^{o} si se realizan varios giros en una secuencia. Por otra parte, si fuese necesario realizar giros de menor amplitud o para alcanzar ángulos que no sean múltiplos de 30, se puede programar una secuencia donde se realicen varios giros de 30^{o}, y finalmente concluir con un giro de un ángulo menor recalculando los puntos $p3$ y $p4$ del giro, empleando para ello las funciones *giroI* y *giroD*.

4.5. Configuración de la simulación

Una vez han sido programadas todas las funciones necesarias para llevar a cabo los diferentes movimientos del robot, llega el momento de comprobar el funcionamiento del robot en el entorno de la simulación. Para poder ejecutar la simulación ha sido necesario ajustar una serie de parámetros, en primer lugar el periodo de muestreo y el tiempo total de la misma, pero también varios coeficientes para los contactos entre el plano y las extremidades y para las propias articulaciones. Es necesario destacar la importancia del periodo de muestreo, se ha tomado un periodo de 1 ms, dado que es una simulación donde existen 18 articulaciones que representan los motores y 2 más que proporcionan los grados de libertad al cuerpo del robot, es decir, debe calcular muchos movimientos compuestos en cada instante y simular los contactos de los extremos de todas las patas con el plano, lo que la convierte en una simulación relativamente costosa computacionalmente. Si se toma un periodo mayor, por ejemplo 10 ms, se producen errores en los cálculos de la simulación y el robot realiza pequeños movimientos aparentemente aleatorios debido a errores al procesar algunos contactos, y en ocasiones la simulación ha abortado debido a que el periodo de muestreo demasiado grande ha generado un error que no se ha podido solucionar.

Otro de los parámetros a mencionar es el amortiguamiento de las articulaciones que le proporcionan los grados de libertad al cuerpo del robot. Al tener un valor distinto de 0 se suaviza de manera muy ligera el movimiento, especialmente durante el arranque. Se ha escogido un valor de 100 $\frac{\mu N}{m/s}$.

Unos de los coeficientes más importantes de la simulación son el de rigidez y amortiguamiento en el contacto entre las patas y el plano. El coeficiente de rigidez modela la resistencia de los materiales a deformaciones producidas en el contacto, se ha tomado un valor por defecto de 100 μN/m dado que este parámetro prácticamente no afecta a la simulación. El coeficiente de amortiguamiento define la fuerza con la que se ve repelido el robot al caer sobre el plano. Si toma un valor demasiado pequeño el robot lo atravesará al entrar en contacto con él, ya que no ejerce la fuerza suficiente como para mantenerlo sobre él, y si es demasiado grande el robot saldrá disparado al entrar en contacto con el plano puesto que la fuerza ejercida por él será demasiado fuerte. Se ha ajustado experimentalmente a un valor de 10 $\frac{mN}{m/s}$.

Finalmente, los últimos parámetros a ajustar son los coeficientes de fricción estático y dinámico. Estos parámetros modelan el deslizamiento y la fricción existente entre las superficies, en este caso, entre el plano y los extremos de las patas. Se ha tomado unos coeficientes de 0.3 para la fricción estática y 0.25 para la dinámica, dado que finalmente el robot trabajará sobre suelos que permitirán cierto deslizamiento y el extremo de sus patas se cubrirá con goma, situación que se aproxima al contacto entre cemento húmedo y caucho, que son dos materiales que presentan los coeficientes indicados anteriormente al entrar en contacto[21].

4.6. Generación de movimientos.

Tras haber configurado y ajustado correctamente la simulación, el último paso antes de ejecutarla es generar los movimientos. Para ello se han empleado una serie de llamadas a las funciones desarrolladas anteriormente como se muestra en la Figura 4.26.

```
% Configuración inicial
q1 = [  pi/4, 0, 0];
q2 = [  pi/4, 0, 0];
q3 = [     0, 0, 0];
q4 = [     0, 0, 0];
q5 = [- pi/4, 0, 0];
q6 = [- pi/4, 0, 0];
t = 0;
% Avance recto
movs1 = moveLI(q1, p0(1), params, t, T, t + 1);
movs2 = moveLD(q2, p0(2), params, t, T, t + 1);
movs3 = moveLI(q3, p0(3), params, t, T, t + 1);
movs4 = moveLD(q4, p0(4), params, t, T, t + 1);
movs5 = moveLI(q5, p0(5), params, t, T, t + 1);
movs6 = moveLD(q6, p0(6), params, t, T, t + 1);
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, p0(2), params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, p0(3), params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, p0(6), params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
```

(a) Inicio del movimiento y avance recto.

```
% Avance derecha
direccion = -pi/2;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
paux2 = p1(2);
paux3 = p1(3);
paux6 = p1(6);
[p1, p2] = puntosAvance(p0, avance, vectorDireccion);
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
for i = 1:2    % TFinSecuencia = 1 + ifinal * 3
    movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
    movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
    movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
    movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
    movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
    movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
    movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
    movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
    movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
    movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
    movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
    movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
end % t = 16
```

(b) Avance hacia la derecha.

Figura 4.26: Fragmentos de código para realizar avances en línea recta.

Puesto que se necesita mantener el estado actual, se han creado unos vectores con los ángulos de cada pata, los cuales serán actualizados tras cada movimiento, se han denominado $q1, q2, ...q6$ y se encuentran al inicio de la Figura 4.26a. En el primer momento se debe llevar al robot a la posición base, definida por el vector $p0$. Tras realizar cualquier movimiento de las patas, se deben actualizar los valores de las posiciones actuales de las articulaciones y el instante actual, la variable t, con el tiempo que se le haya dedicado a dichos movimiento. Tras alcanzar la posición inicial, el robot debe ir a $p1$ para iniciar el movimiento en bucle, esto se hace en dos pasos, primero se llevan tres patas al punto empleando el movimiento parabólico, mientras las otras tres permanecen en el sitio, y tras acabar las otras tres que en el movimiento anterior habían permanecido inmóviles usan el movimiento parabólico para ir a $p1$, actualizando las variables de las articulaciones y el tiempo entre los movimientos. Una vez alcanzada esta posición se ejecuta un bucle donde se encuentran dos secuencias idénticas a la que se encuentra al final de

la Figura 4.26b, la primera para ir a $p2$, y la segunda para volver a $p1$, lo que habilita poder repetir la secuencia en bucle. Tras concluir el bucle, si se desea avanzar en otra dirección, deberán actualizarse el valor de *direccion* y recalcular $p1$ y $p2$. Adicionalmente, se han empleado tres variables auxiliares, $paux2$, $paux3$ y $paux6$, para almacenar la posición previa de los extremos de las patas 2, 3 y 6, para poder así mantenerlas inmóviles mientras se lleva al robot en dos fases al nuevo $p1$. Una vez hecho esto se entra en un nuevo bucle para avanzar en la nueva dirección tanto como se desee.

El movimiento de giro resulta muy similar al de avance en lo referente a su programación. Tras obtener $p3$ y $p4$, se llevan las patas a $p3$ si se desea girar hacia la derecha o a $p4$ si el giro es hacia la izquierda. Para hacer esto se mueven primero tres patas y luego las tres restantes del mismo modo que se ha descrito anteriormente. Finalmente se llevan a cabo los giros empleando las funciones para los movimientos en arco y en parábola.

En este caso, tras llevar las tres primeras patas a $p3$, se puede dar inicio al movimiento, dado que estas tres patas ya se encuentran en la posición adecuada para describir los arcos del giro. Mientras se realiza este segundo movimiento las tres patas que habían permanecido inmóviles en la secuencia anterior pueden avanzar con el movimiento parabólico a $p3$, con lo que se puede ahorrar una secuencia para llevar las otras tres patas al punto de inicio de los arcos. El funcionamiento descrito se puede apreciar en la figura Figura 4.27, y si se repite la secuencia mostrada al final de la figura intercalando los grupos de patas, se puede alargar el giro para alcanzar el ángulo de orientación deseado.

```
% Giro hacia la derecha (q2, q3 y q6 empiezan en p1)
[p13, p14] = giroI(p0(1), -ang, origenPata1);
[p23, p24] = giroD(p0(2),  ang, origenPata2);
[p33, p34] = giroI(p0(3), -ang, origenPata3);
[p43, p44] = giroD(p0(4),  ang, origenPata4);
[p53, p54] = giroI(p0(5), -ang, origenPata5);
[p63, p64] = giroD(p0(6),  ang, origenPata6);

p3 = [p13; p23; p33; p43; p53; p63];
p4 = [p14; p24; p34; p44; p54; p64];

movs1 = [movs1; moveCI(q1, p3(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p3(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p3(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
movs1 = [movs1; moveGI(q1, -ang, origenPata1, params, t, T, t + 1.5)];
movs2 = [movs2; moveCD(q2, p3(2), h, params, t, T, t + 1.5)];
movs3 = [movs3; moveCI(q3, p3(3), h, params, t, T, t + 1.5)];
movs4 = [movs4; moveGD(q4,  ang, origenPata4, params, t, T, t + 1.5)];
movs5 = [movs5; moveGI(q5, -ang, origenPata5, params, t, T, t + 1.5)];
movs6 = [movs6; moveCD(q6, p3(6), h, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1.5;
```

Figura 4.27: Código para producir giros sobre sí mismo en el robot.

Se ha creado un código que repite estas secuencias para describir trayectorias algo más complejas, pero debido a su extensión y a que resulta de poco interés tras analizar los fragmentos anteriores, se ha añadido como Apéndice C.

4.7. Resultados de la simulación.

Tras haber ajustado correctamente la simulación se ha ejecutado una secuencia de movimientos donde el robot realiza avances en diferentes direcciones, describiendo un cuadrado y después un rombo en el espacio y concluye realizando algunos giros sobre sí mismo. En la Figura 4.28 se muestra la trayectoria descrita por el robot a lo largo de la simulación.

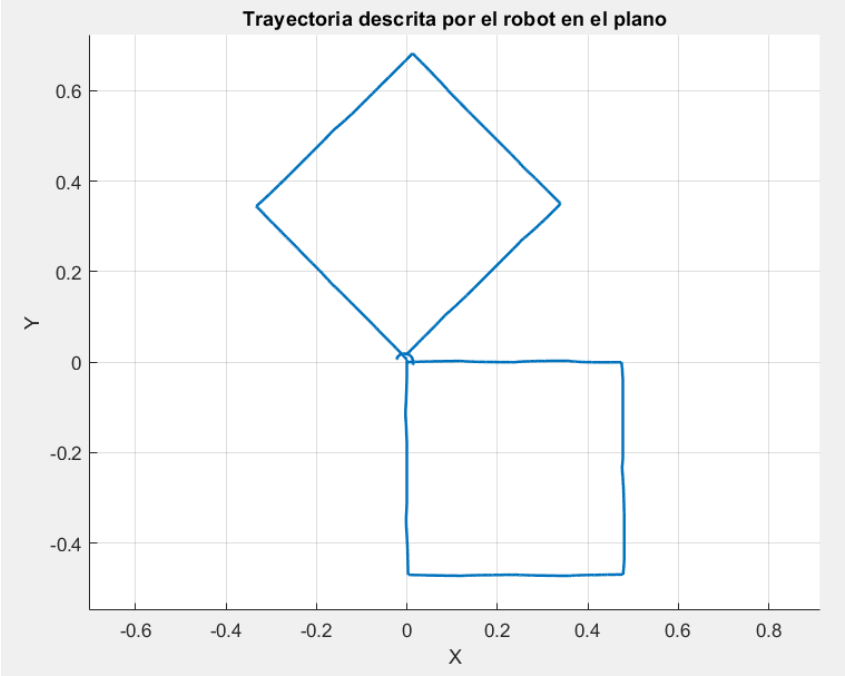
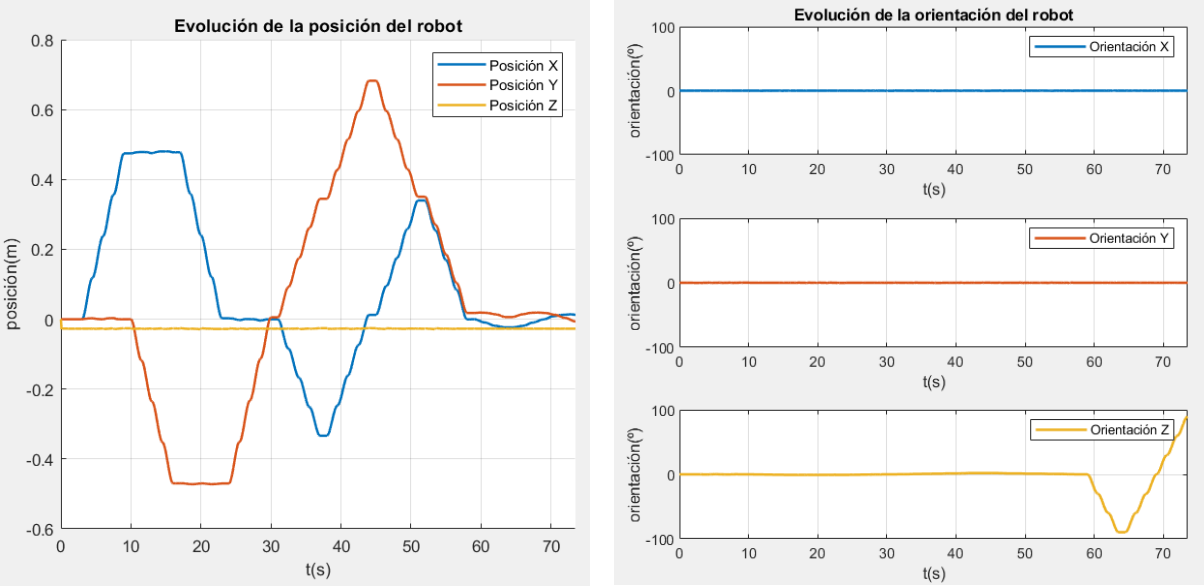


Figura 4.28: Representación de la trayectoria seguida por el robot.

En esta trayectoria se realizan ciclos de avance hacia delante, derecha, izquierda y atrás para construir un cuadrado y probar los desplazamientos frontales y laterales. Tras acabar estos movimientos, se realizan una serie de desplazamientos en diagonal de modo que la trayectoria descrita dibuja un rombo, para comprobar el correcto funcionamiento de los movimientos en diagonal. Finalmente, el robot realiza un giro de 90^{o} sobre sí mismo hacia la derecha, y después uno de 180^{o} hacia la izquierda, de modo que se prueban todos los movimientos implementados.

Además de la trayectoria descrita, resulta de interés observar la evolución de los valores que toma el cuerpo del robot en sus diferentes grados de libertad, para ello se han representado las gráficas de la Figura 4.29. Se aprecia los valores de la posición en el eje Z, al igual que los de la orientación en los ejes X e Y, se mantienen prácticamente constantes a lo largo de la simulación. Esto representa que la altura del robot se mantiene constante, que es como debería ocurrir ya que en ningún momento se realizan movimientos que impliquen la modificación de este parámetro. Ocurre la misma situación con las orientaciones en los ejes X e Y, lo que indica que la base del robot no se ha visto inclinada ni hacia delante o hacia atrás, y tampoco hacia los lados. A la vista de estos resultados, es posible afirmar que la estabilidad del robot se ha cumplido de la manera prevista.



(a) Evolución de la posición.

(b) Evolución de la posición.

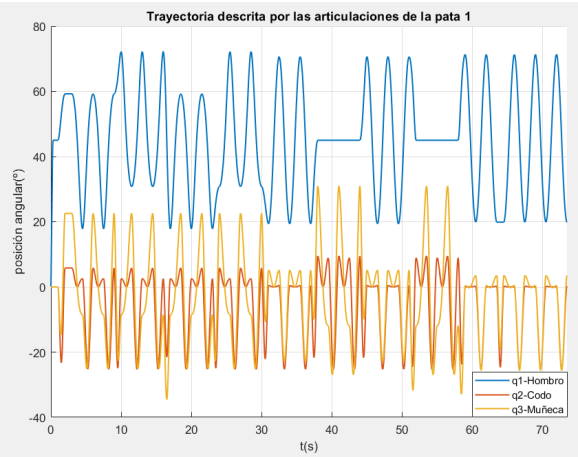
Figura 4.29: Representaciones gráficas de la posición y orientación del robot a lo largo de la simulación.

Analizando en detalle las gráfica de la Figura 4.29a, se observa que, hasta el instante 30, solo se produce movimiento o bien en la coordenada X o bien en la Y, lo que concuerda con la descripción de la trayectoria cuadrada, mientras que desde ese instante hasta el 60, se produce movimiento en ambos ejes, lo que representa los movimientos diagonales del rombo. Finalmente, se pueden apreciar algunos cambios en las posiciones X e Y hacia el final de la simulación, los cuales representan los giros sobre sí mismo. Se podría pensar que en un giro sobre sí mismo las posiciones no deberían verse afectadas, y lo hacen en pequeña medida debido a posibles errores en la calibración de la posición del sistema de referencia de la base junto a que el giro no se realiza con el cuerpo sobre un punto inmóvil del plano XY, sino que existen pequeños errores en el cálculo de los movimientos que causan este efecto. Por otra parte, dado que la orientación evoluciona de manera adecuada y los cambios en la posición son muy pequeños no se ha considerado necesario tratar de mejorar el rendimiento en este aspecto.

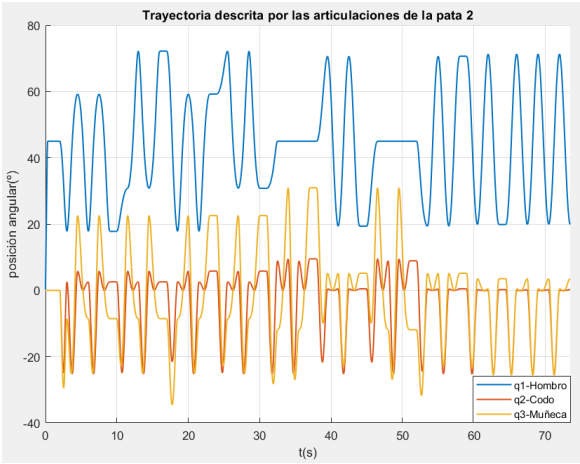
Respecto a la gráfica de la Figura 4.29b, resulta de menos interés, dado que durante la mayor parte de la simulación, las tres componentes de la orientación del cuerpo del robot permanecen constantes. Son los últimos momentos de la simulación, cuando se producen los giros sobre el eje Z del cuerpo del robot, los que representan la evolución en los giros. Si se observa detenidamente, se puede apreciar un pico hacia el instante 64, en el que la orientación

en el eje Z se estabiliza brevemente alrededor de -90º, y también que al concluir la simulación, dicha orientación se encuentra en 90º aproximadamente. El hecho de que se alcancen estas orientaciones proporciona la seguridad de que los giros se han realizado correctamente, es decir, los giros han sido programados como secuencias de rotaciones de 30º, y cada uno de estos ha producido un giro real de este valor.

De manera adicional, se ha considerado que puede ser de interés analizar la evolución de las posiciones de las articulaciones. Por este motivo se han representado las evoluciones de las articulaciones de las patas 1 y 2 en la Figura 4.30. Se han escogido estas patas ya que no funcionan de manera coordinada, es decir, para producir el movimiento una debe avanzar mientras la otra retrocede y viceversa.



(a) Evolución para la pata 1.



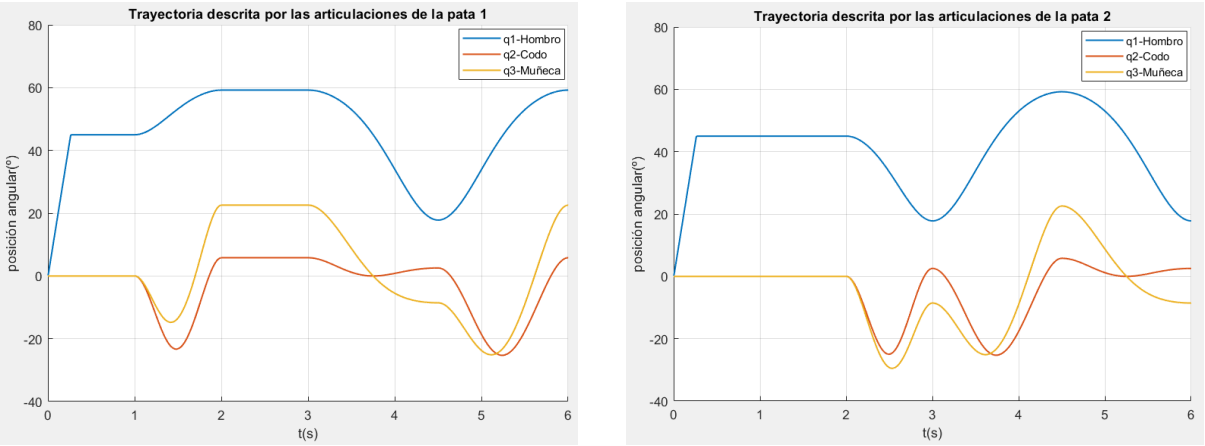
(b) Evolución para la pata 2.

Figura 4.30: Representaciones gráficas de las posiciones de las articulaciones de las patas 1 y 2 a lo largo de la simulación.

Las gráficas obtenidas pueden parecer señales con ruido, pero simplemente se trata de que las variables representadas tienen un significado complejo de entender directamente. Para describir trayectorias lineales, parabólicas y de arco, se requiere de una composición de movimientos de rotación en las articulaciones donde resulta difícil entender que está ocurriendo. Por otra parte, sí que hay algunos detalles de funcionamiento que se pueden deducir de las gráficas. En concreto, en ambas gráficas se pueden apreciar a partir del instante 30 de la simulación, unos tramos donde $q1$ permanece constante alrededor de 6 segundos. Esto se debe a que, en estos momentos se está produciendo el desplazamiento en diagonal, el cual forma un ángulo de 45º con la horizontal, y cuando la diagonal se alinea con las patas, estas solo necesitan llevar a

cabo movimientos en $q2$ y $q3$ para realizar el movimiento, dado que estas articulaciones ya se encuentran en el plano por el cual se desea desplazarse sin necesidad de reorientar $q1$.

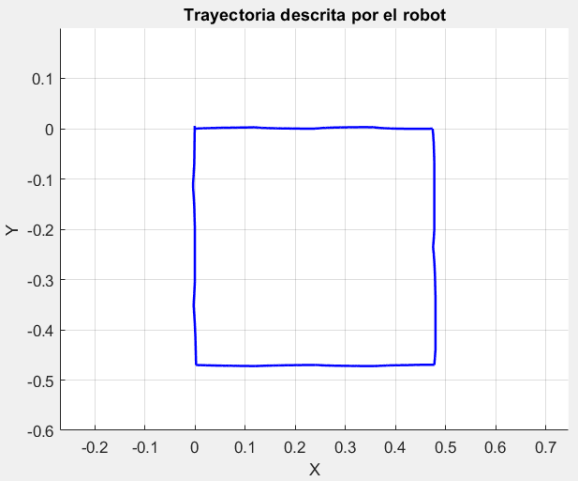
Se ha considerado representar de manera separada los instantes iniciales de estas gráficas para poder apreciar mejor su significado. Las gráficas de la Figura 4.31 muestran el resultado obtenido. En ellas resulta algo más sencillo comprender lo que está sucediendo. En el instante 1 las patas se mueven a la posición base, que para el caso de las patas 1 y 2, mantienen las articulaciones $q2$ y $q3$ en 0^{o}, pero llevan $q1$ a 45^{o}. En los instantes 2 y 3, cada conjunto de patas se mueve a la posición definida por $p1$, por este motivo, la posición de la pata 2 permanece constante en el instante entre 1 y 2 mientras la pata 1 realiza el movimiento. Este efecto se produce al revés en el instante siguiente, donde la pata 1 permanece inmóvil mientras la 2 se posiciona. Entre los instantes 3 y 6 se produce un movimiento lineal y uno parabólico en cada pata, los cuales son mucho más difíciles de analizar, en especial para $q2$ y $q3$. En estos movimientos se puede apreciar una forma parabólica en $q1$, lo que implica que las patas 1 y 2 se desplazaron a una posición adelanta y retrasada respectivamente, definida por $p2$, para posteriormente volver al estado anterior, la configuración definida por $p1$.



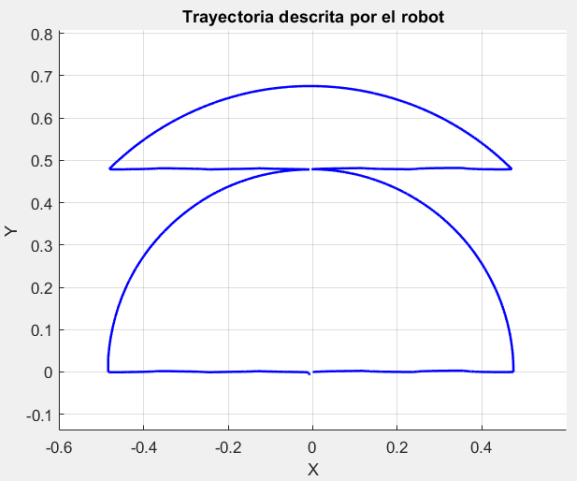
(a) Evolución para la pata 1. (b) Evolución para la pata 2.

Figura 4.31: Representaciones gráficas de las situaciones de las articulaciones de las patas 1 y 2 en un tramo de avance.

También se ha considerado de interés comparar el recorrido que realiza el robot tratando de seguir una misma trayectoria de dos métodos distintos, en primer lugar empleando únicamente desplazamientos lineales y laterales, y posteriormente empleando únicamente el avance hacia delante junto a los movimientos de giro para orientar su cuerpo. Se ha propuesto una sencilla trayectoria en forma de cuadrado, y sus resultados se muestran en la Figura 4.32 y la Figura 4.33.



(a) Trayectoria con desplazamientos laterales.



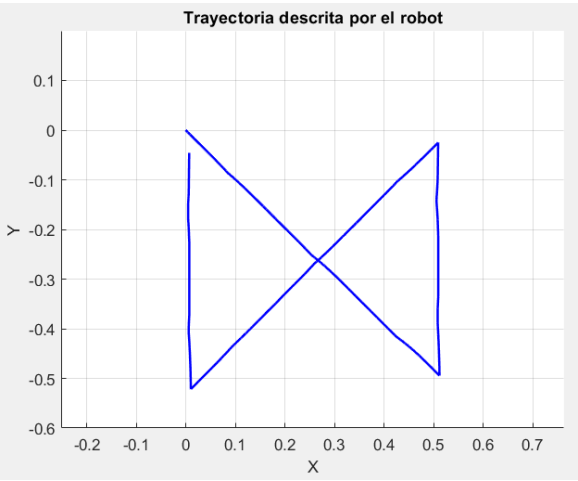
(b) Trayectoria con giros.

Figura 4.32: Representaciones gráficas de las trayectorias realizadas.

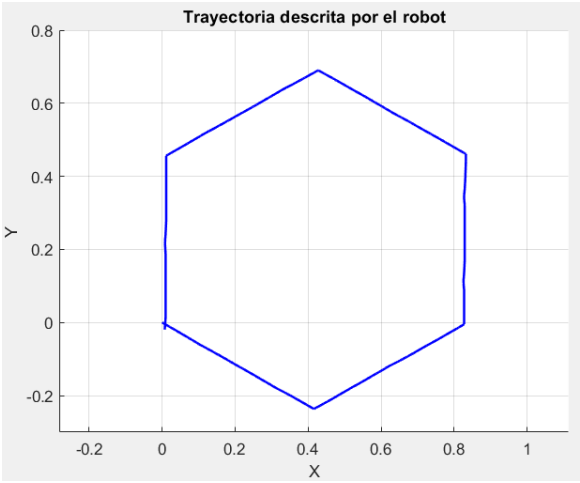
Figura 4.33: Trayectoria realizada por el robot sobre el plano de la simulación.

En la Figura 4.32a se aprecia la trayectoria descrita empleando desplazamientos hacia delante, atrás y hacia la derecha y la izquierda, obteniéndose un resultado bastante satisfactorio. Por otra parte, en la Figura 4.32b, se observa una trayectoria bastante extraña, esto es debido a que el sensor del que dispone el robot en la simulación realiza las mediciones sobre el propio sistema de referencia del robot, el cual rota junto a él. Aunque sea posible apreciar los segmentos rectos que describe, y el hecho de que finaliza en el mismo punto en el que empieza, resulta conveniente tratar los datos para proporcionar una imagen de la trayectoria que se describe sobre el plano horizontal de la simulación. De este modo se ha construido la gráfica de la Figura 4.33, en la cual se puede comprobar que la trayectoria se describe con precisión similar tanto aplicando los giros como sin ellos.

Para finalizar se ha tratado de evaluar el comportamiento del robot ante trayectorias de carácter más complejo, obteniéndose los seguimientos representados en la Figura 4.34.



(a) Trayectoria con ditintos movimientos.



(b) Trayectoria de forma hexagonal.

Figura 4.34: Representaciones de las trayectorias adicionales realizadas.

En la figura se puede observar que el seguimiento es algo menos preciso que el producido con la trayectoria cuadrada. En los dos casos se produce un pequeño error en la posición final del robot, puesto que no consigue alcanzar de nuevo la posición del inicio de la simulación. Estos errores pueden ser debidos a que la combinación de movimientos y los cambios en las configuraciones para realizar los avances, junto con las imperfecciones de la cinemática obtenida causan que los movimientos no alcancen la amplitud deseada. Al no alcanzar esta amplitud, la distancia recorrida en determinadas direcciones puede ser menor que la esperada, y por consiguiente puede no alcanzarse la posición inicial al final de la simulación. Pese a este error, se ha obtenido un seguimiento más que aceptable, y se ha comprobado con la trayectoria hexagonal de la Figura 4.34b, que pueden llevarse a cabo desplazamientos en una variedad de direcciones más compleja que las rectas y diagonales.

5

Implementación

A la hora de construir el robot real se han planteado diversas opciones. Se han tomado una buena parte de las recomendaciones del montaje en la página del modelo Antdroid[12]; sin embargo, se han realizado varias modificaciones para aproximarse en mayor medida a los objetivos propuestos.

5.1. Componentes

Los componentes básicos del robot son la estructura mecánica o chasis del mismo, los actuadores, que serán los servomotores encargados de mover las extremidades, y finalmente la electrónica que proporcionará la energía y el control necesarios.

5.1.1. Estructura y motores

Las piezas que componen el robot han sido impresas en 3D a partir de los modelos de las mismas obtenidas a través de Internet. Estas piezas se encontraban en formato *.stl*, lo que complica en gran medida su edición. Por este motivo se imprimieron las piezas con las impresoras disponibles en la universidad, empleando plástico de tipo ABS y las recomendaciones de montaje del robot.

Es necesario destacar el estudio realizado acerca de la capacidad de los motores para mover el robot. Se ha supuesto una configuración con las patas en forma de L, dado que en esta posición

cada una de las articulaciones se encuentra con la mayor separación del centro de masas del cuerpo del robot. En este caso, el par soportado por las articulaciones es el que debe ser capaz de ejercer el motor para mover el robot. Es necesario tener en cuenta que el robot mantiene en todo momento tres patas apoyadas, lo que implica que el par que necesita un solo motor será la tercera parte del valor obtenido anteriormente. Para el cálculo de dicho par se debe tener en cuenta las distancias desde el centro de masas del robot hasta cada una de las articulaciones, tal y como se muestran en la Figura 5.1.

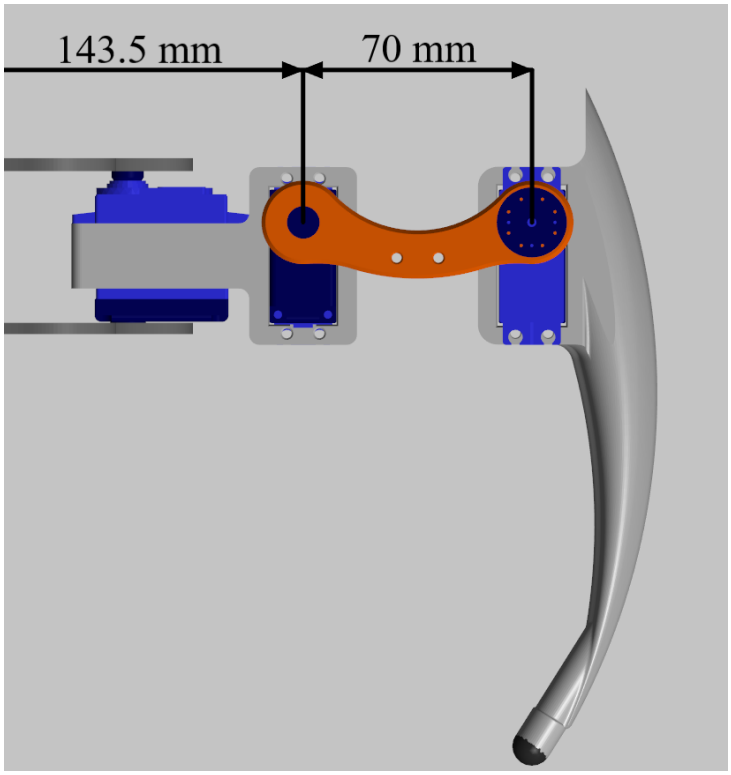


Figura 5.1: Representación de las distancias para el cálculo del par.

También ha sido necesaria una estimación de la masa del cuerpo del robot, para ello se han considerado las masas de los motores, las indicadas en la página de Antdroid para las piezas, la de la batería y la del adaptador de tensión. Es posible observar el cálculo realizado en la Ecuación 5.1. Tras el cálculo se ha añadido un margen que representa el resto de la electrónica.

$$12 \cdot 55g(\text{motores}) + 503g(\text{plástico}) + 220g(\text{batería}) + 42g(\text{adaptador}) = 1425g \quad (5.1)$$

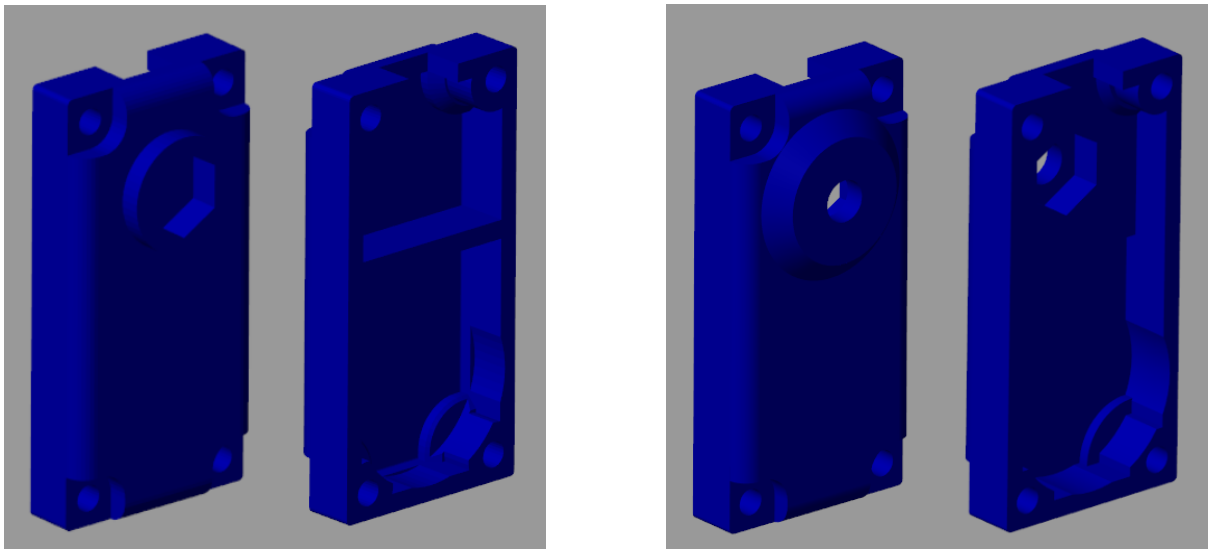
De este modo se obtiene un valor de 1425 g, pero se ha tomado un valor de 1.5 kg para sobre-

dimensionar y considerar la masa del resto de elementos electrónica, con el fin de tener cierto margen a la hora de escoger los motores. Finalmente se realiza el cálculo del par necesario, para ello se expresarán las distancias en centímetros, ya que los pares de los servomotores digitales de esta clase proporcionan su par en kg·cm. Se realizará el cálculo para las dos articulaciones que se encuentran a lo largo de cada extremidad, dado que la restante no necesita ejercer tanto par, pues no levanta el peso del robot. Los resultados obtenidos se pueden apreciar en la Ecuación 5.2.

$$\begin{aligned} T_{q2_{min}} &= \tfrac{1{,}5}{3} \cdot 14{,}35 = 7{,}175\ kg \cdot cm \\ T_{q3_{min}} &= \tfrac{1{,}5}{3} \cdot 21{,}35 = 10{,}675\ kg \cdot cm \end{aligned} \tag{5.2}$$

Se observa de este modo que los pares necesarios para las articulaciones $q2$ y $q3$ son de 7.175 y 10.675 kg·cm respectivamente, valores que se encuentran por debajo de los 11 kg·cm de par máximo ofrecido por los servomotores Towerpro MG996r recomendados para el modelo del robot. Dado que los motores recomendados cumplen con el par necesario para el movimiento del robot, serán estos los empleados en la implementación.

Cabe destacar también que una de las piezas del modelo Antdroid original, la mostrada en la Figura 5.2a, causaba problemas debido a la escasa sujeción que esta proporcionaba al tornillo de cabeza hexagonal que actúa como eje de la articulación. En particular, debido al peso del robot y a la fuerza que las articulaciones necesitan ejercer para el movimiento, se produce un esfuerzo en la unión entre el tornillo y la pieza que provoca que se separen. Por este motivo, a partir de las dimensiones de la pieza original, se ha diseñado la pieza representada en la Figura 5.2b, cuya principal diferencia es que incorpora la muesca para la cabeza hexagonal del tornillo en la cara interior de la pieza, detalle que, junto con el pequeño espacio cilíndrico para el tornillo, asegura que el esfuerzo producido no separe las piezas ni se produzcan deformaciones en la estructura de la articulación. Con esta mejora en las piezas, se ha podido mejorar notablemente el desempeño del robot.



(a) Pieza original.

(b) Pieza modificada.

Figura 5.2: Representación de las piezas que actúan como tapa de los servomotores.

5.1.2. Electrónica

En primer lugar se planteó usar el microcontrolador Wroom ESP32 para el robot, dado que es un microcontrolador pequeño y versátil que ofrece buenas prestaciones. Tiene una alta velocidad de procesamiento, su CPU funciona a un máximo de 240 MHz. Para proporcionar una comparativa, el reloj de la CPU del ATmega328, el microcontrolador empleado por muchas placas Arduino, entre ellas Arduino UNO y Arduino Nano, puede alcanzar un máximo de 20MHz, y normalmente no superan los 16 MHz en las placas Arduino si no se llevan a cabo ciertas operaciones sobre ellas.

Otro de los motivos de especial interés para emplear esta placa es que dispone de un módulo Bluetooth, un módulo Bluetooth Low Energy y un módulo Wi-Fi, de modo que no se requiere de ningún componente adicional para realizar las comunicaciones de la placa.

También se trata de un microcontrolador que permite configurar interrupciones en prácticamente todos sus pines, y también permite generar salidas analógicas empleando señales PWM. Inicialmente fue planteado manejar los 18 motores empleando directamente los pines del microcontrolador, pero este solo posee 16 canales PWM, es decir, puede generar un máximo de 16 señales de este tipo, lo cual es insuficiente para controlar todos los motores.

Dado que no es posible controlar todos los motores directamente, se ha empleado una con-

troladora de servomotores PCA9865, que permite manejar hasta 16 motores a través de una comunicación por I2C. El ESP32 funciona correctamente con la misma alimentación que los motores, pero la controladora permite alimentar por separado la lógica y los motores, por lo que se ha empleado un convertidor DC/DC MP1584EN adicional para obtener 4 V para su lógica, dado que las tensiones de 5 V o superiores causan oscilaciones en los movimientos de los servomotores.

Para la alimentación del robot se ha empleado una batería LiPo, esto es debido a que se prevee un consumo elevado de corriente durante el funcionamiento. Dado que solo son necesarios 5 o 6 V para los motores sería adecuada una batería de 2 celdas, la cual proporciona una tensión nominal de 7.4 V, y es relativamente ligera en comparación con otras baterías LiPo con más celdas y presenta más capacidad que otras opciones, como un Powerbank, de masa similar. Teniendo en cuenta lo anterior, se ha escogido una batería LiPo 4500 mAh 60C 7.4V. En lo referente al consumo de corriente, en las características de los servomotores se indica que su consumo oscila entre 500 y 900 mA. Si se toma el valor máximo y se multiplica por los 18 motores de los que dispone el robot, se obtiene un consumo de hasta 16.2 A, aunque experimentalmente se ha comprobado que el pico de arranque de los motores puede superar los 20 A, pero aunque el consumo medio sea mucho menor, la batería será capaz de proporcionar corriente suficiente en estos picos. La autonomía que proporciona la batería no se ha considerado de especial interés, dado que el robot implementado servirá para el estudio de los movimientos en un entorno controlado, y no realizará trabajos autónomos donde se tenga que asegurar el tiempo de funcionamiento.

Adicionalmente se ha requerido un convertidor DC-DC para pasar de la tensión de la batería a la del sistema, para ello se ha empleado un adaptador de tensión 20A HV SBEC, que es un convertidor de altas prestaciones, con salida seleccionable a 5, 5.5, 6, 7 o 9 V mediante un saltador y capacidad de proporcionar hasta 20 A de salida, valor capaz de alimentar los 18 servomotores durante su arranque.

En la Figura 5.3 se puede observar el circuito empleado en el robot. La mitad de los motores son controlados con el propio microcontrolador, los correspondientes a uno de los conjuntos de patas que realiza movimientos sincronizados, en concreto los que pertenecen a las patas 1, 4 y 5. La otra mitad de los servomotores, los que corresponden a las patas 2, 3 y 6 son manejados por la PCA9685.

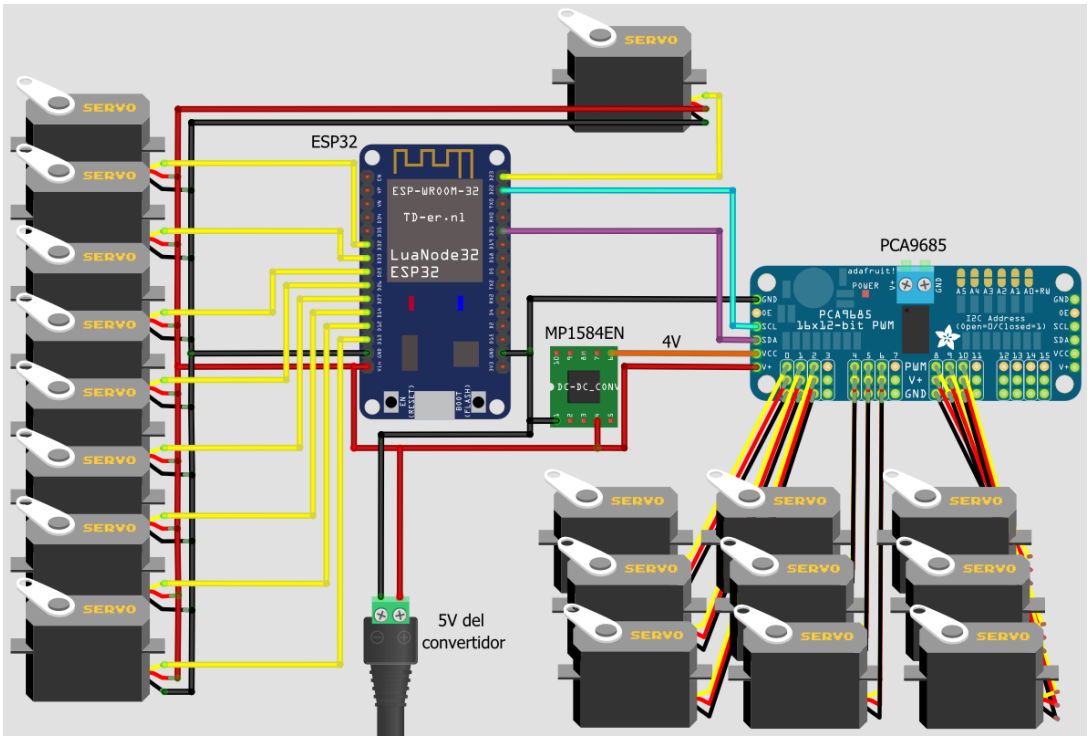


Figura 5.3: Esquema del circuito empleado en el robot.

El circuito ha sido montado sobre una placa perforada, como se puede observar en la Figura 5.4. Se han empleado pines machos para poder incorporar los conectores de los servomotores, y hembras para introducir el microcontrolador en la placa. Cabe destacar que se han incorporado dos entradas de alimentación desde el convertidor ya que este dispone de dos salidas, y de este modo, se simplifica ligeramente el trabajo a realizar sobre la placa.

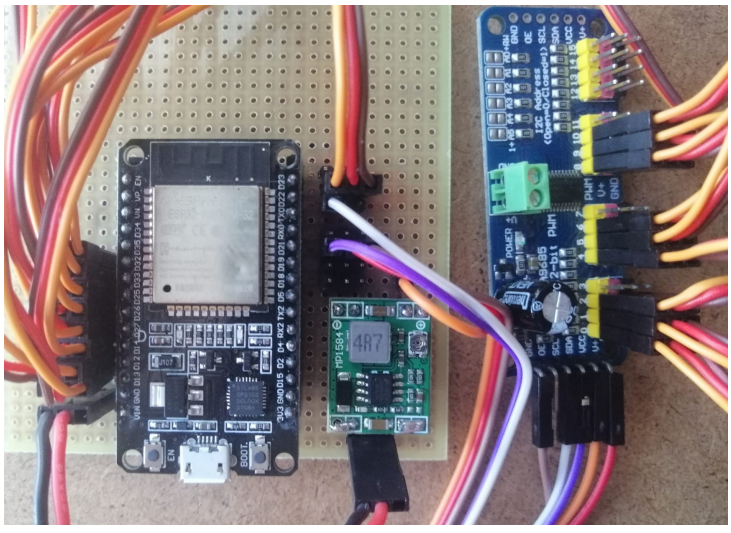


Figura 5.4: Imagen del circuito montado.

Inicialmente se planteaba usar la salida del adaptador a 6 V para la alimentación de los motores y el ESP32, pero, debido a que la salida del convertidor no es exactamente de este valor, y que la controladora de servomotores genera oscilaciones en los movimientos de los motores con esta alimentación, se ha optado por usar la salida de 5 V, que es la representada en la Figura 5.3. Esta salida está unida al adaptador, el cual se alimenta directamente de la batería LiPo.

5.2. Montaje

Para llevar a cabo el montaje del robot se han empleado diferentes tornillos de métricas estandarizadas, siguiendo de manera aproximada las recomendaciones del modelo del robot.

En la Tabla 5.1 se puede observar el listado de tornillos, tuercas y arandelas necesarios para el montaje. No se han incluido algunos de los tornillos empleados en la tabla, dado que son aquellos que están incluidos con los servomotores. Por otra parte, debido a la dificultad de obtener cojinetes de dimensiones tan específicas, estas piezas han sido impresas en 3D.

Componente	Usado en	Número
Tornillo M2x8.25	Discos del eje de los servomotores	36
Tornillo M3x20	Servomotores y separadores	84
Tornillo M4x16 cabeza hexagonal	Tapas de los servomotores	18
Tuerca M4 autoblocante	Tapas de los servomotores	18
Arandela M4	Tapas de los servomotores	36
Cojinete 4x10x4	Arcos intermedios de las extremidades	12

Tabla 5.1: Tornillería empleada.

Finalmente se ha llevado a cabo el ensamblaje del robot siguiendo la guía detallada que se muestra en la página web del modelo Antdroid[12], en primer lugar se sitúan los servomotores en su posición intermedia, para posibilitar el máximo rango de movimientos una vez se hayan fijado piezas a su eje, después se deben montar todas las patas por separado para finalmente unirlas a la base del robot. Una vez completada la estructura se han pegado unas cubiertas de goma sobre el extremo de las patas para mejorar la adherencia, y se ha colocado la batería y la

placa perforada sobre la base, obteniéndose como resultado el robot mostrado en la Figura 5.5.

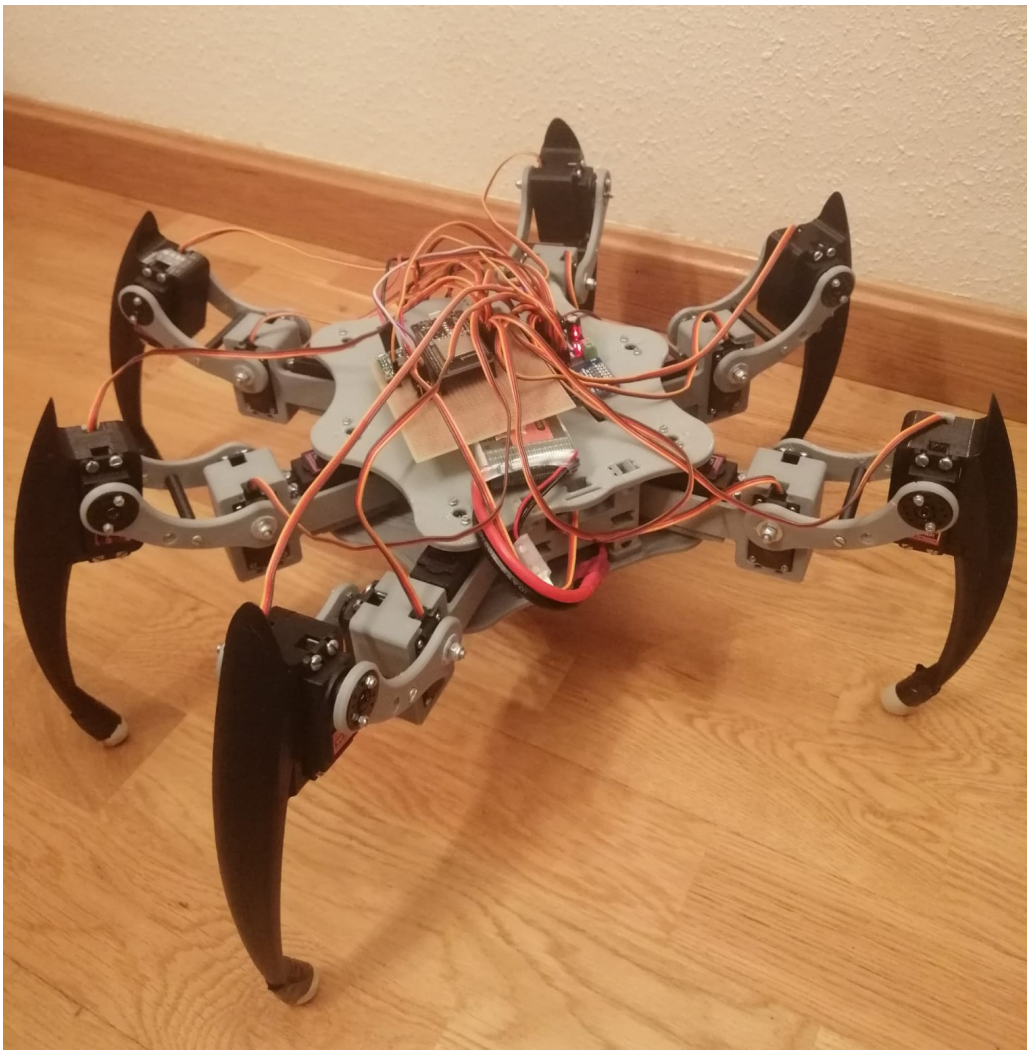


Figura 5.5: Imagen tomada del robot una vez montado.

5.3. Programación del robot

Se ha empleado el entorno de desarrollo de Arduino para realizar el programa del robot debido a la facilidad que ofrece para trabajar y al gran soporte disponible para esta plataforma.

Para poder trabajar con el microcontrolador Wroom ESP32, es necesario configurar adecuamente el entorno de Arduino. Para ello se debe acceder al menú de preferencias, desde la pestaña *Archivo-¿Preferencias*, en ese punto se mostrará una ventana similar a la mostrada en la Figura 5.6, donde será necesario hacer clic en el desplegable del Gestor de URLs Adicionales de tarjetas, que aparece resaltado en la figura, e incluir la dirección siguiente:

"https://dl.espressif.com/dl/package_esp32_index.json".

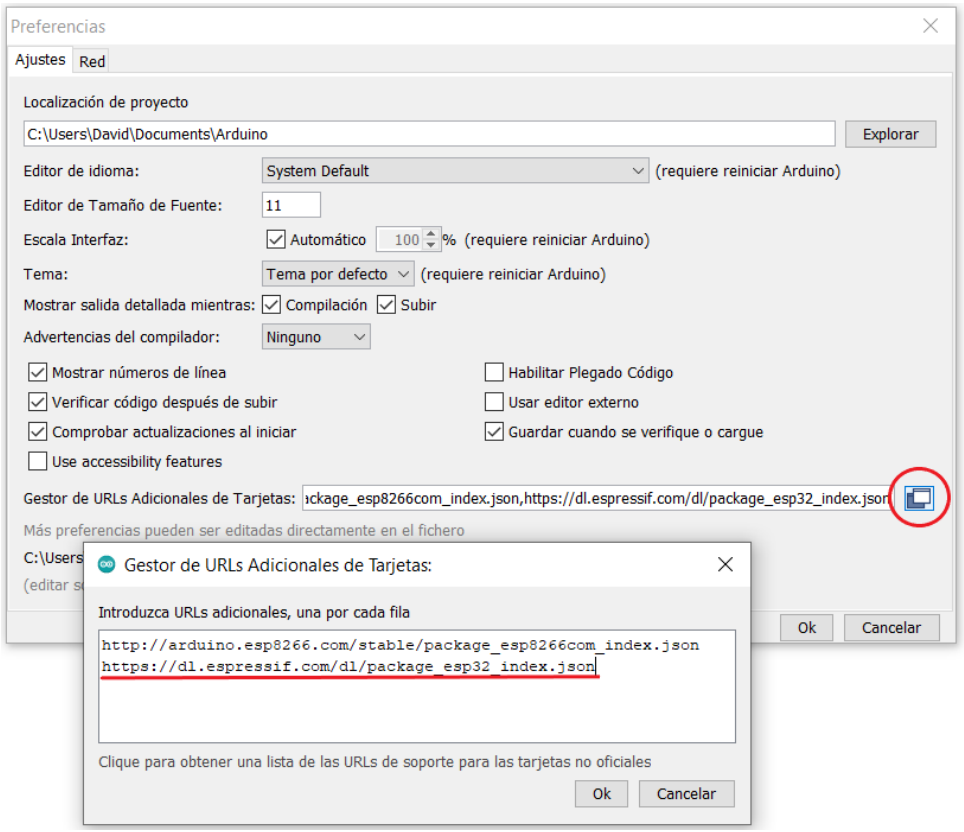


Figura 5.6: Configuración del entorno de programación.

Tras haber configurado correctamente las preferencias se debe seleccionar el microcontrolador desde la pestaña Herramientas del entorno de desarrollo, tal y como se muestra en la Figura 5.7. Una vez completados estos pasos, se puede proceder a realizar el programa sin mayores preocupaciones.

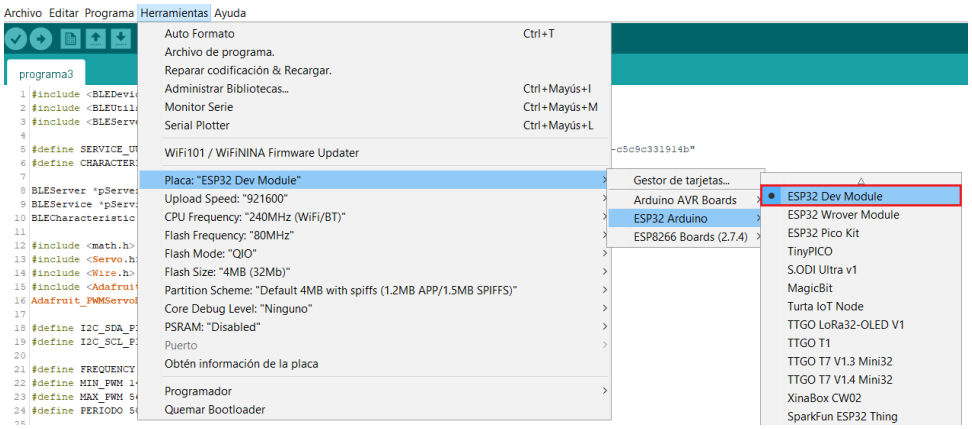


Figura 5.7: Configuración de la placa para la que se desea compilar el programa.

5.4. Programación de los movimientos

Para incorporar los movimientos de la simulación al robot real se han traducido los movimientos del código de Matlab al lenguaje C adaptado del entorno de Arduino. En su mayor parte, el código es idéntico, por lo que solo resulta de interés explicar las partes que difieren.

Una de las partes más importantes es la actuación directa de las articulaciones, dado que se trata de manejar servomotores. Para cumplir esta tarea se han empleado una serie de librerías, las mostradas en la Figura 5.8. Puesto que la mitad de los servomotores se controlan directamente con el microcontrolador, se ha empleado la librería *Servo.h*, que permite crear variables que representan un servomotor, que ,tras asociarse a un pin de la placa, ofrece la posiblidad de escribir directamente el ángulo deseado. Para los servomotores correspondientes a la controladora de servomotores, se ha empleado la librería *Wire.h*, para gestionar la comunicación I2C, y la *Adafruit_PWMServoDriver.h* para trabajar con la propia controladora. Al disponer de una sola controladora, se debe crear una variable de esta librería, que será inicializada a la velocidad de comunicación serie para su correcto funcionamiento.

```
#include <Servo.h>
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();
```

Figura 5.8: Librerías para el control de los servomotores.

Otro aspecto importante es la calibración de los motores, dado que el montaje y el sentido de giro de los motores no tienen por qué coincidir exactamente con el funcionamiento que se da en la simulación. Por este motivo se han definido las dos estructuras de la Figura 5.9a, donde se definen unos valores de offset, que ajustarán la posición inicial, ya que para disponer del máximo rango de movimiento, los motores se han montado en la posición base cuando se encontraban a 90^{o}, es decir, su posición intermedia. También se ha definido un parámetro signo, que indica si el sentido de giro de ese motor es idéntico o está invertido respecto al sentido de los cálculos. Además se requieren unos parámetros de posición mínima y máxima, para limitar el alcance de los movimientos y evitar colisiones entre piezas del robot o posiciones donde las piezas y los motores sean forzados en exceso. Finalmente, la estructura *ServoTipo2*, dispone de un parámetro pin, que indica en que pin de la controladora se encuentra el servomotor. En la Figura 5.9b, se pueden observar los valores resultantes de la calibración de los motores a partir de la estructura física del robot.

```
typedef struct {
  int offset;
  int signo;
  int min_pos;
  int max_pos;
} ServoTipo1;

typedef struct {
  uint8_t pin;
  int offset;
  int signo;
  int min_pos;
  int max_pos;
} ServoTipo2;
```

(a) Defición de las estructuras.

```
patal[0].offset = 90 - 8;   patal[0].signo = -1;
patal[0].min_pos = -75;     patal[0].max_pos = 90;
patal[1].offset = 90 - 5;   patal[1].signo =  1;
patal[1].min_pos = -90;     patal[1].max_pos = 35;
patal[2].offset = 45 - 0;   patal[2].signo =  1;
patal[2].min_pos =   15;    patal[2].max_pos = 90;

pata4[0].offset = 90 - 0;   pata4[0].signo =  1;
pata4[0].min_pos = -75;     pata4[0].max_pos = 90;
pata4[1].offset = 90 + 5;   pata4[1].signo = -1;
pata4[1].min_pos = -90;     pata4[1].max_pos = 35;
pata4[2].offset = 90 - 0;   pata4[2].signo = -1;
pata4[2].min_pos = -28;     pata4[2].max_pos = 28;

pata5[0].offset =  90 - 0;  pata5[0].signo = -1;
pata5[0].min_pos = -75;     pata5[0].max_pos = 90;
pata5[1].offset =  90 + 0;  pata5[1].signo =  1;
pata5[1].min_pos = -90;     pata5[1].max_pos = 35;
pata5[2].offset = 135 - 0;  pata5[2].signo =  1;
pata5[2].min_pos = -90;     pata5[2].max_pos =  -15;

pata2[] = {{0,  90 - 20,  1,  -75, 90},
           {1,  90 - 15, -1,  -90, 35},
           {2, 135 - 25, -1,   15, 90}};

pata3[] = {{4, 90 - 15, -1,  -75, 90},
           {5, 90 - 15,  1,  -90, 35},
           {6, 90 - 20,  1,  -28, 28}};

pata6[] = {{ 8, 90 - 20,  1,  -75,  90},
           { 9, 90 -  4, -1,  -90,  35},
           {10, 45 - 25, -1,  -90, -15}};
```

(b) Inicialización con los valores para cada motor.

Figura 5.9: Estructuras creadas para configurar los servomotores.

Como se puede observar en la Figura 5.10, mostrada a continuación, la actuación sobre ambos tipos de servomotores emplea la función *constrain* para asegurar que se toman valores dentro del intervalo de la posición mínima y máxima del servo. Se multiplica el valor resultante por el signo y se le añade el offset, obteniendo así el valor deseado en la posición real.

```
servosPatal[0].write(constrain( 0, patal[0].min_pos, patal[0].max_pos) * patal[0].signo +  patal[0].offset);
servosPatal[1].write(constrain( 0, patal[1].min_pos, patal[1].max_pos) * patal[1].signo +  patal[1].offset);
servosPatal[2].write(constrain(45, patal[2].min_pos, patal[2].max_pos) * patal[2].signo +  patal[2].offset);
```

(a) Actuación sobre los servomotores del ESP32.

```
void writeServo2(ServoTipo2 motor, int angle) {   // Mueve el ángulo del servo especificado al ángulo recibido como parámetro
  int pulse_width;
  angle = constrain(angle, motor.min_pos, motor.max_pos);
  pulse_width = map(angle * motor.signo + motor.offset, 0, 180, MIN_PWM, MAX_PWM);
  pwm.setPWM(motor.pin, 0, pulse_width);
}
```

(b) Función para mover los servomotores de la PCA9685.

Figura 5.10: Código para la actuación de los servomotores.

Otro aspecto importante es la función que traduce los movimientos deseados en los movimientos de las articulaciones. Funciona de manera similar a la función *moveMotor* de la Sub-

sección 4.4.1, realiza un movimiento a partir de las posiciones angulares objetivo. Esta función se muestra en la Figura 5.11.

```
void moveAbsJ(double qT[][NumServosPata], double T) {              // Mueve el robot entre dos configuraciones de ángulos
  double a[NumPatas][NumServosPata], b[NumPatas][NumServosPata];   // double c[JOINTS] = {0, 0, 0}, no se incluye, dado que tiene valores nulos
  double d[][NumServosPata] = {{q0[0][0], q0[0][1], q0[0][2]},
                               {q0[1][0], q0[1][1], q0[1][2]},
                               {q0[2][0], q0[2][1], q0[2][2]},
                               {q0[3][0], q0[3][1], q0[3][2]},
                               {q0[4][0], q0[4][1], q0[4][2]},
                               {q0[5][0], q0[5][1], q0[5][2]}};

  double T2 = T * T, T3 = T2 * T, q[NumPatas][NumServosPata];      // Se crea T2 para no tener que calcular este valor varias veces
  int i, j;

  for (i = 0; i < NumPatas; i++) {                                 // Se calculan los parámetros a y b para cada articulación
    for (j = 0; j < NumServosPata; j++) {
      a[i][j] = -2 * (qT[i][j] - q0[i][j]) / T3;
      b[i][j] = 3 * (qT[i][j] - q0[i][j]) / T2;
    }
  }

  int itFinal = T / PERIODO;                                       // Se define la cantidad de iteraciones necesarias, sabiendo que cada una durará PERIODO ms
  double t1, t2, t3;                                               // Se declaran las variables para no tener que repetir su declaración
  unsigned long t;

  for (int itActual = 0; itActual < itFinal; itActual++) {         // El bucle calcula las siguientes posiciones de los servos para seguir la trayectoria cúbica
    t = millis();
    t1 = itActual * PERIODO;
    t2 = t1 * t1;
    t3 = t2 * t1;

    for (i = 0; i < NumPatas; i++) {
      for (j = 0; j < NumServosPata; j++) {
        q[i][j] = a[i][j] * t3 + b[i][j] * t2 + d[i][j];           // + c[i] * t1, no se incluye, al no afectar al valor
      }
    }
    for (i = 0; i < NumServosPata; i++) {                          // Se mueven los motores
      servosPata1[i].write(constrain(round(q[0][i]), pata1[i].min_pos, pata1[i].max_pos) * pata1[i].signo +  pata1[i].offset);
      servosPata4[i].write(constrain(round(q[3][i]), pata4[i].min_pos, pata4[i].max_pos) * pata4[i].signo +  pata4[i].offset);
      servosPata5[i].write(constrain(round(q[4][i]), pata5[i].min_pos, pata5[i].max_pos) * pata5[i].signo +  pata5[i].offset);
    }
    delay(1);
    for (i = 0; i < NumServosPata; i++) {
      writeServo2(pata2[i], round(q[1][i]));
      writeServo2(pata3[i], round(q[2][i]));
      writeServo2(pata6[i], round(q[5][i]));
    }
    delay(PERIODO - (millis() - t));                               // Se espera el tiempo necesario para alcanzar los PERIODO ms de cada iteración
  }
}
```

Figura 5.11: Función para el movimiento de las articulaciones del robot.

La función *moveAbsJ* desarrollada para este propósito se genera una trayectoria cúbica para cada servomotor, en función de la posición inicial del mismo y la posición final deseada. Para crear dicha trayectoria se emplea un sistema similar al descrito para el movimiento lineal en la Subsección 4.4.2. Finalmente se realizan los movimientos siguiendo la trayectoria en un bucle temporizado, para asegurar el cumplimiento de los tiempos. También se emplea una pequeña espera de 1 ms entre los movimientos de las tres primeras patas y los del resto, para suavizar ligeramente el pico de intensidad producido por el arranque de los servomotores. Es necesario mencionar que esta función puede ser llamada de modo que se desee realizar la trayectoria cúbica en un solo periodo, por lo que se realizaría el movimiento de manera directa; no obstante, esto solo ocurre cuando la llamada proviene de las funciones para el movimiento lineal o parabólico, que también describen una trayectoria de esta clase, por lo que se cumplirá el objetivo de evitar movimientos bruscos de los motores.

El último aspecto destacable del programa es el sistema empleado para sincronizar el avance, es decir, que a cada paso sean tres patas distintas las que hagan avanzar el cuerpo del robot. Para ello se ha definido una variable global de tipo carácter, llamada *movimiento*, que solo toma dos valores, I o D, haciendo referencia de este modo al conjunto de patas que debe realizar el avance. Si el valor de la variable es I, el avance lo realiza la pata 1 y su conjunto asociado, las patas 4 y , y si la variable toma valor D, será la pata 2, junto con las patas 3 y 6 las que realicen el avance.

De este modo se ha creado una función que realiza tres trayectorias lineales y tres parabólicas de manera simultánea. Esta función implementa el avance del robot, según el valor de la variable *movimiento*, calcula la trayectoria lineal para el desplazamiento de las tres patas a las que hace referencia la variable en ese instante, y la trayectoria lineal para el resto de ellas. Tras calcular las trayectorias cúbicas que seguirán los extremos de cada pata, realiza los movimientos de acuerdo a las trayectorias empleando la función *moveJ*, que a su vez emplea las funciones de la cinemática inversa y la función *moveAbsJ* descrita anteriormente para realizar los movimientos.

5.5. Aplicación móvil

Una parte del proyecto consiste en el desarrollo de una aplicación móvil para manejar el robot. Se ha optado por emplear la comunicación a través del módulo Bluetooth Low Energy del microcontrolador Wroom ESP32, dado que resulta más sencilla que la comunicación Wi-Fi y requiere de menos corriente que el Bluetooth clásico. La comunicación consistirá en el envío de una serie de bytes por parte del teléfono al microcontrolador, estos bytes representaran un código que será interpretado por el programa del robot para realizar una determinada acción.

La aplicación se ha creado empleando Thunkable X, una plataforma para el desarrollo de aplicaciones móviles. La plataforma ofrece la posibilidad de crear proyectos tras registrarse, con cada usuario es posible registrar hasta 10 proyectos, y cada proyecto está dividido en dos partes, el diseño y la programación. La pestaña donde se lleva a cabo el diseño se muestra en la Figura 5.12, y en el centro se encuentra la representación de la aplicación que se está desarrollando. Adicionalmente, dispone de una estructura jerarquizada de los componentes de la pantalla en el lado superior izquierdo, de un menú con todos los posibles componentes en la parte inferior izquierda, y de un menú con las propiedades del objeto seleccionado en el lado

derecho de la interfaz.

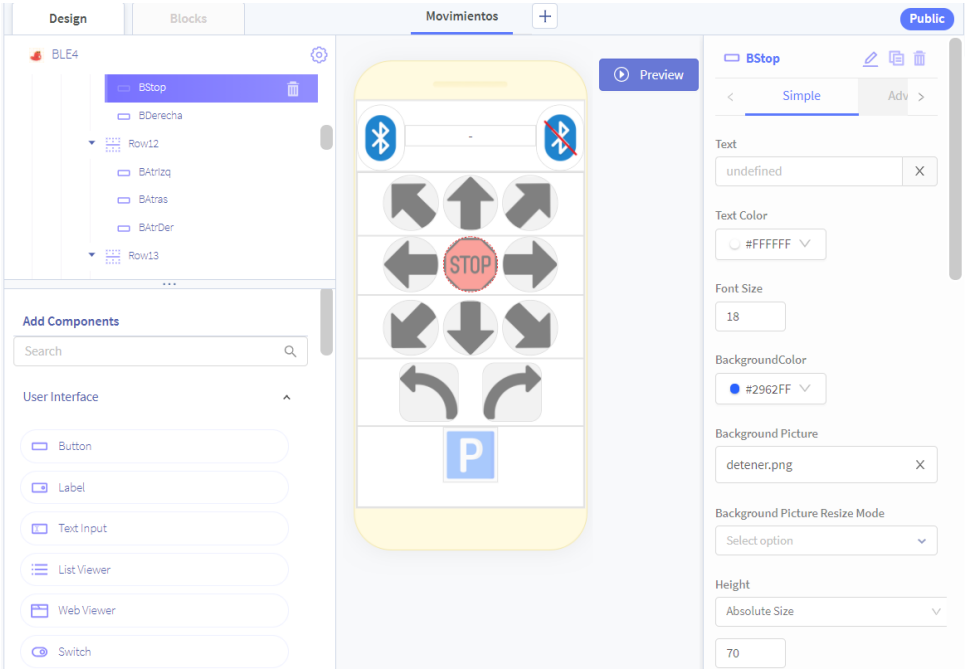


Figura 5.12: Pestaña de diseño de Thunkable X.

La parte de programación se lleva a cabo en la pestaña de bloques, que es la representada en la Figura 5.13. Thunkable X ofrece un entorno de programación orientado a eventos, es decir, el programa responde ante un evento como puede ser una acción del usuario, la recepción de un mensaje o el final de un temporizador. Esta programación se lleva a cabo mediante bloques, lo cual hace el programa mucho más intuitivo, aunque dificulta la realización de algunas tareas.

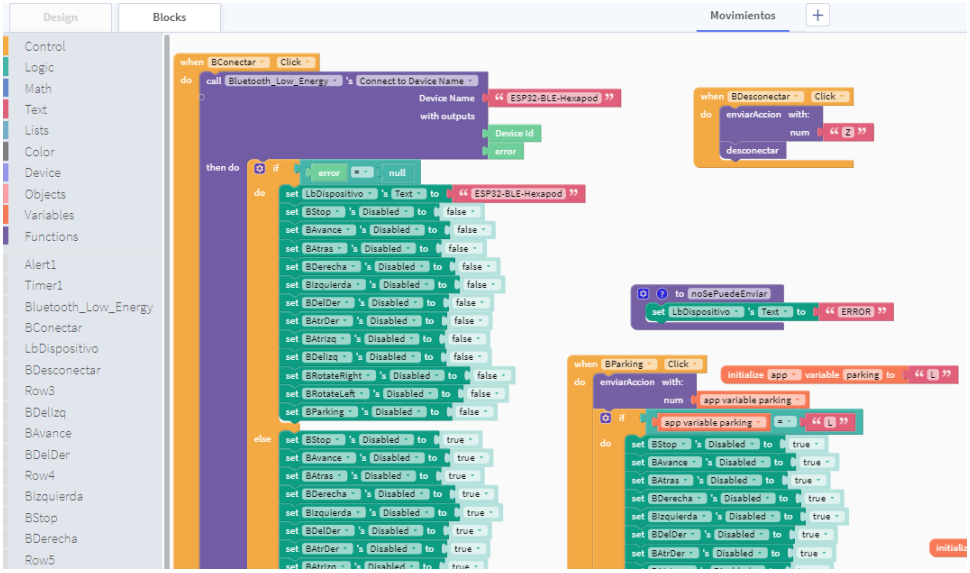


Figura 5.13: Pestaña de programación mediante bloques de Thunkable X.

En la Figura 5.14 se puede observar el aspecto de la aplicación desarrollada. En primer lugar se ha de usar el botón de la parte superior izquierda para establecer la conexión, en caso de que se establezca correctamente, se habilitarán los demás botones, pudiendo así enviar códigos al robot para que realice distintos movimientos. En concreto se han implementado códigos para realizar desplazamientos en línea recta y diagonal, para girar en ambos sentidos, para detener el movimiento y para entrar en una posición de reposo. Todo el detalle del código se encuentra en el Apéndice E.

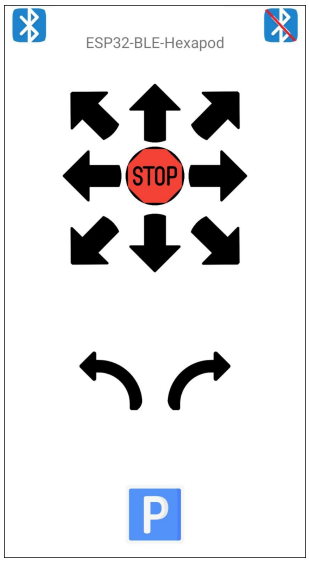


Figura 5.14: Captura de la aplicación desarrollada.

5.6. Programación de la comunicación

La comunicación a través del Bluetooth Low Energy emplea una serie de identificadores llamados UUID (Universally Unique Identifier), que hacen referencia al servicio o característica con el que se está tratando. En este caso, el microcontrolador actuará como servidor, y esperará peticiones por parte de la aplicación móvil, que actuará como cliente. Dada la sencillez de la comunicación y las limitaciones de Thunkable X para trabajar con este tipo de Bluetooth, se ha empleado el mismo UUID tanto para el servicio como para la característica y la aplicación contactará con el microcontrolador empleando directamente el nombre que se le ha dado al servidor, en este caso, *ESP32-BLE-Hexapod*. De este modo se evita la necesidad de escanear en busca del servidor del robot.

Una propiedad del protocolo de Bluetooth Low Energy es que no mantiene la conexión, tras

el envío de cada paquete de datos, el módulo se desconecta hasta necesitar realizar otro envío, con el objetivo de reducir el consumo. Esto puede suponer problemas de funcionamiento si algún mensaje no se envía correctamente, y dado que no se va a realizar una gestión exhaustiva de los posibles errores, se desea la posibilidad de abortar la comunicación y poder reiniciarla desde cero. Con el fin de gestionar esta comunicación se ha desarrollado la función representada en la Figura 5.15.

```
void BLE() {
  BLEDevice::deinit(false);
  BLEDevice::init("ESP32-BLE-Hexapod");
  pServer = BLEDevice::createServer();
  pService = pServer->createService(SERVICE_UUID);
  pCharacteristic = pService->createCharacteristic(
                                        CHARACTERISTIC_UUID,
                                        BLECharacteristic::PROPERTY_READ |
                                        BLECharacteristic::PROPERTY_WRITE
                                      );
  pCharacteristic->setValue("-");
  pService->start();
  //BLEAdvertising *pAdvertising = pServer->getAdvertising();
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  //pAdvertising->setMinPreferred(0x06);
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
}
```

Figura 5.15: Función para iniciar el servidor.

Para poder implementar estas características se ha realizado una búsqueda de información sobre el funcionamiento de las funciones de las librerías del Bluetooth Low Energy. En primer lugar se ha empleado la función *deinit(false)* para eliminar todos los datos de la comunicación anterior en el caso de que hubiese una, pero sin borrar los datos de la caché de los mensajes, dado que esta funcionalidad no ha sido implementada correctamente en la librería. Tras eliminar el servidor previo, lo que permitirá volver a realizar una conexión sin tener que reiniciar el programa del robot, da inicio uno nuevo, su nombre será el indicado anteriormente ya que debe ser idéntico al contenido en la aplicación, *ESP32-BLE-Hexapod*. Una vez iniciado, establece su UUID y el de su caracterísica, que en este caso será el mismo, pero también las propiedades de esta última, que son de lectura y escritura, aunque solo se usará la de lectura. Finalmente se configura el servicio ofrecido por el servidor y se anuncia la presencia del servidor, lo que permitirá que otros dispositivos lo detecten[22][23].

5.7. Funcionamiento del programa

El programa desarrollado combina muchos aspectos, desde la cinemática del robot hasta la gestión de la comunicación, pasando por los sistemas que permiten enlazar movimientos. Pero en conjunto, debe ser visto como un programa que recibe una serie de códigos mediante el módulo Bluetooth Low Energy del microcontrolador y actúa en consecuencia. Para poder visualizar rápidamente el funcionamiento del robot junto con la aplicación, se ha desarrollado el diagrama de flujo mostrado en la Figura 5.16.

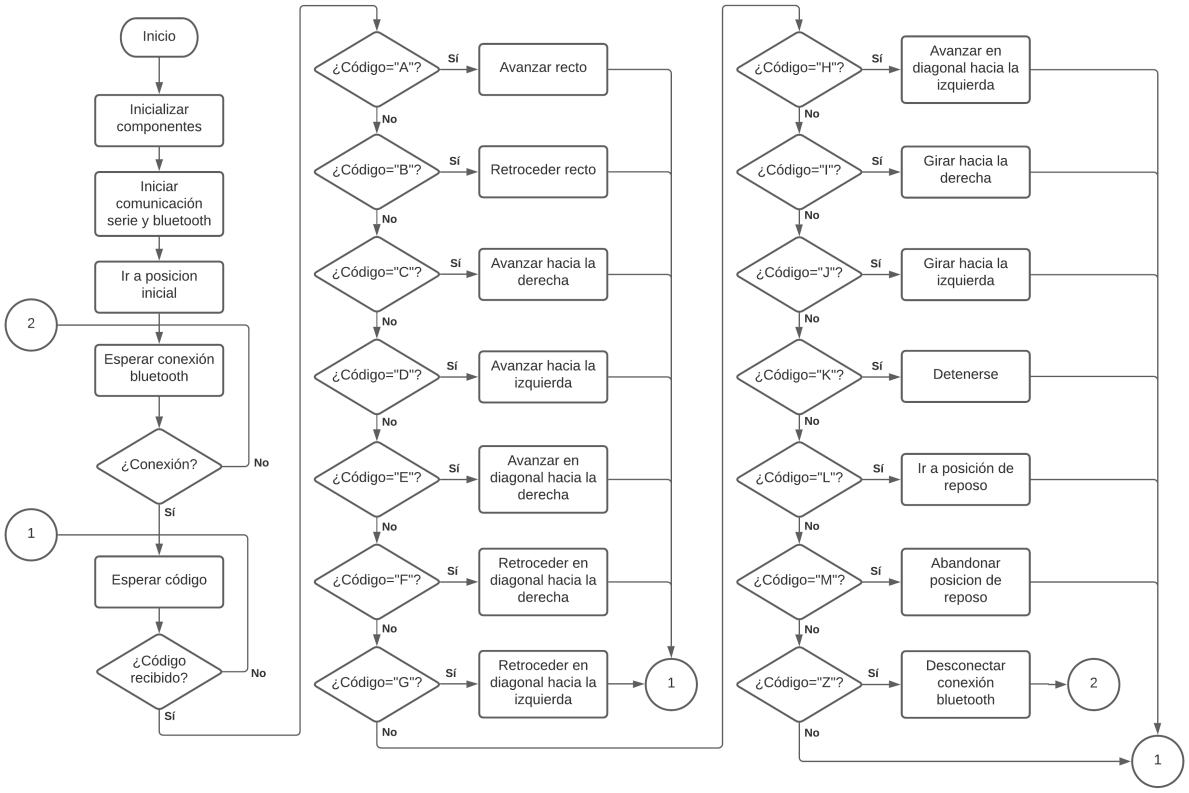


Figura 5.16: Diagrama de flujo del funcionamiento del programa.

En el diagrama de flujo se puede apreciar que el programa se encuentra durante la mayor parte del tiempo en una serie de bucles, o bien esperando que se establezca una conexión, o bien esperando a que se reciba un código. Una vez que se recibe un código, se calculan las posiciones a las que se deben llevar las patas y las trayectorias que deben seguir y se ejecutan. Durante ese tiempo el módulo Bluetooth Low Energy del ESP32 puede recibir otros códigos, pero no serán evaluados hasta que se complete la acción y permanecerán en el buffer de dicho módulo. En caso de que haya más de un código se tomará el que se haya recibido en último lugar, y si no hubiese

ninguno, se tomará el recibido en el instante anterior, ya que el programa está preparado para realizar una acción en bucle hasta que se le indique lo contrario, como por ejemplo, avanzar en una dirección.

6

Análisis de resultados

Se ha desarrollado una simulación del movimiento de un robot hexápodo y resuelto la cinemática directa e inversa de sus extremidades para llevar así a cabo diferentes movimientos. También se ha construido el modelo del robot e implementado dichos movimientos con el robot real proporcionando una interfaz en una aplicación móvil para controlarlo. En lo referente a la simulación, puede observarse en la Figura 6.1 el recorrido realizado.

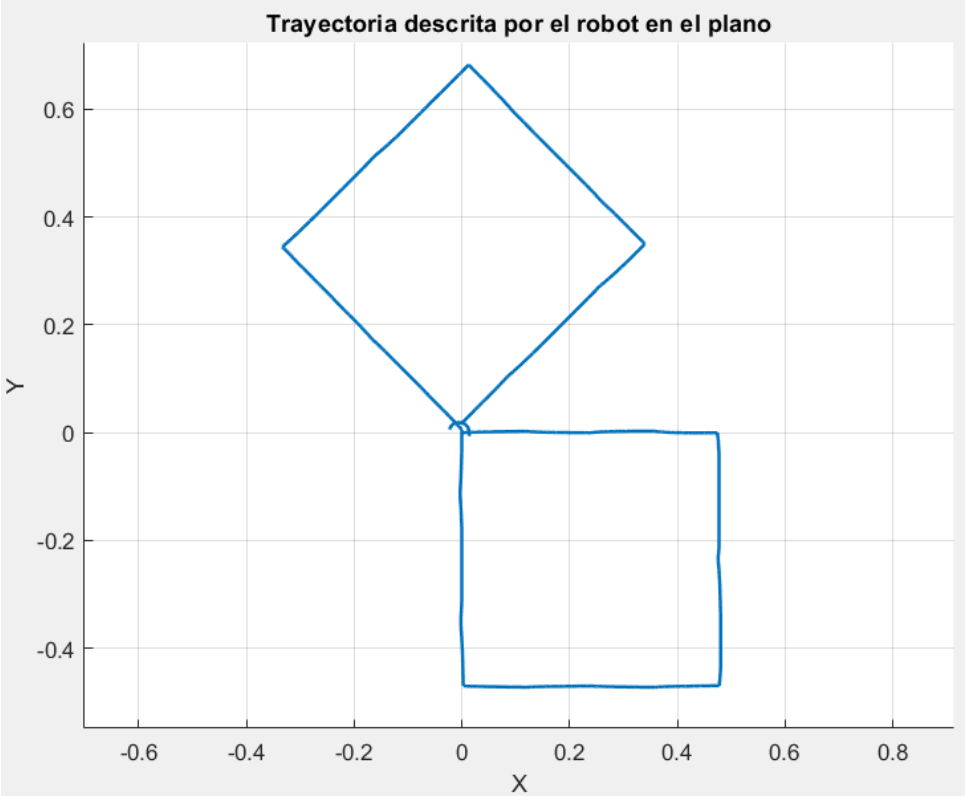
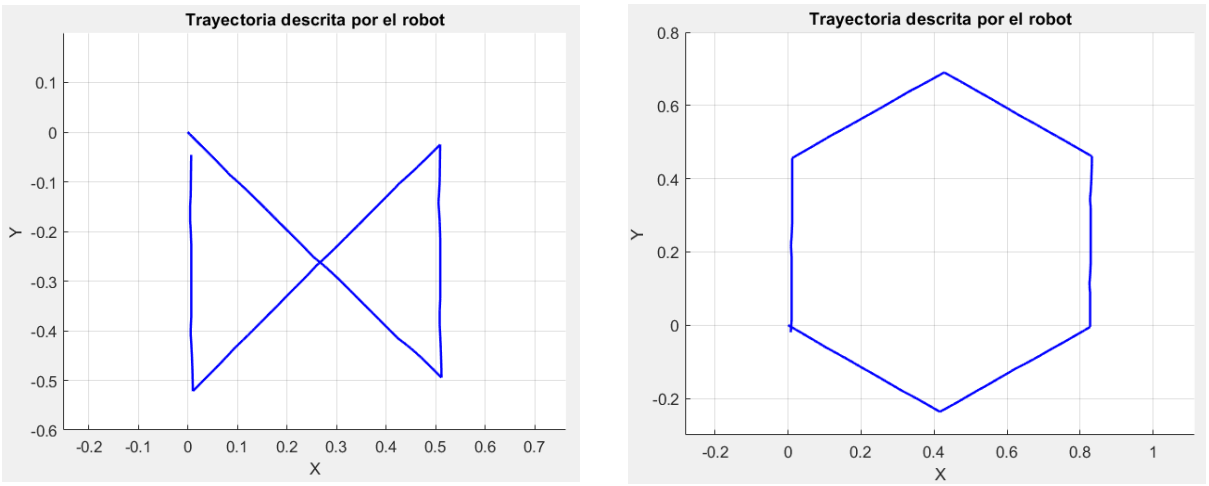


Figura 6.1: Trayectoria descrita por el robot en la simulación.

Como se observa en la figura anterior, el funcionamiento del robot en la simulación es prácticamente ideal, no se producen desviaciones y el robot es capaz de desplazarse en cualquier dirección manteniendo la orientación de su cuerpo constante. Esto implica que la resolución de la cinemática y el desarrollo del movimiento de las articulaciones se ha realizado correctamente, dado que, en un entorno controlado y con una calibración ideal, el robot describe las trayectorias que se le indican prácticamente sin errores. A modo de prueba se han diseñado trayectorias algo más complejas para evaluar el funcionamiento en estos casos, como las mostradas en la Figura 6.2.



(a) Trayectoria en forma de X.

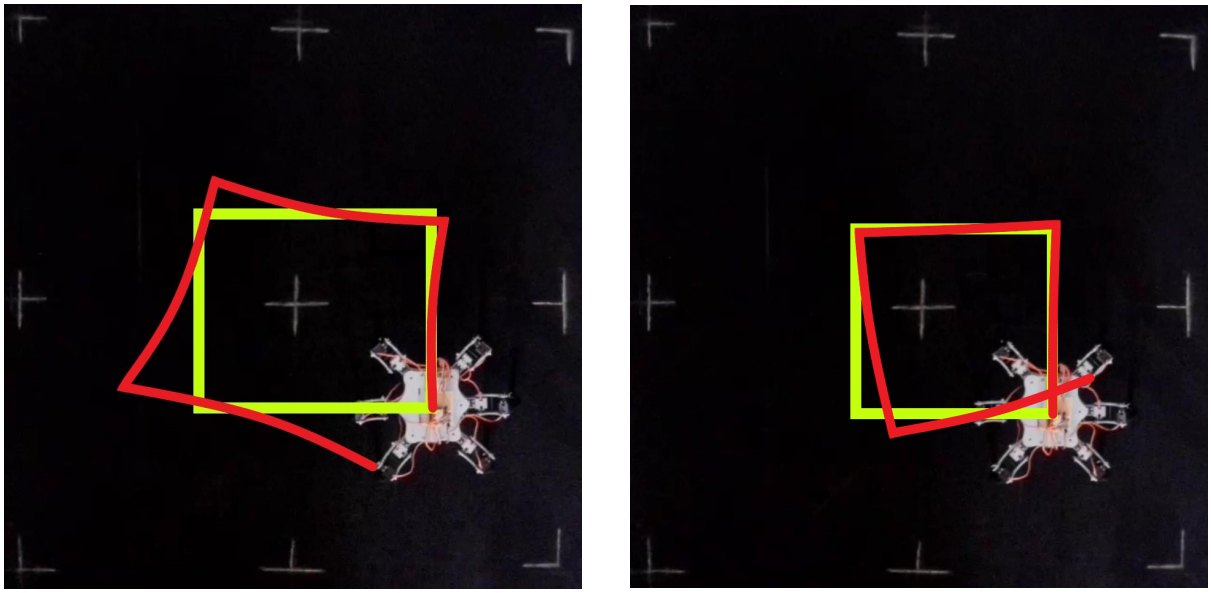
(b) Trayectoria hexagonal.

Figura 6.2: Representación del recorrido realizado por el robot ante distintas trayectorias.

Como se puede apreciar en las imágenes, se produce un pequeño error en el seguimiento, hecho que se puede apreciar dado que los puntos inicial y final de la trayectoria no coinciden. Estos errores se deben a aproximaciones en la cinemática y en los cálculos realizados por Simulink; no obstante, este seguimiento imperfecto es totalmente normal, ya que no existe ningún sistema de control de trayectoria. Se han empleado los sensores de las articulaciones que ofrecen libertad de movimiento y orientación al cuerpo del robot para obtener la información en cada instante y representarla, pero no para establecer una realimentación que permita corregir errores en el seguimiento. A manera de resumen, dado que el robot funciona en bucle abierto y no tiene los medios para corregir su seguimiento, los resultados obtenidos son muy satisfactorios, ya que se realiza una trayectoria muy similar a la deseada pese a las limitaciones.

Por otra parte, se ha empleado una cámara cenital para tomar vídeos del robot realizando dos trayectorias rectangulares, permitiendo así analizar la exactitud con la que se realiza el

seguimiento de la trayectoria. En la Figura 6.3 se pueden observar los resultados obtenidos, las trayectorias propuestas son mostradas en color amarillo, mientras que las trayectorias descritas por el robot se han presentado en rojo.



(a) Trayectoria descrita sin giros.

(b) Trayectoria descrita empleando giros.

Figura 6.3: Representación de las trayectorias descritas por el robot en la realidad.

Para la primera trayectoria, la representada en la Figura 6.3a, el robot modifica la dirección del desplazamiento al llegar a las esquinas, esto es, realiza el recorrido sin reorientar el cuerpo. Como se observa en la figura, el seguimiento contiene varios errores, principalmente debidos a que no se respeta estrictamente la dirección del avance, especialmente en los desplazamientos laterales, sino que se produce una desviación en la dirección, lo cual incrementa el error según se alargue el desplazamiento. En la Figura 6.3b se representa la trayectoria descrita cuando el robot realiza giros sobre sí mismo en las esquinas. En este caso los errores en el avance son menos pronunciados dado que solo se realizan avances frontales, que funcionan algo mejor, pero se producen errores en los giros, si el robot no gira exactamente 90^{o} en el sentido adecuado, no se orientará correctamente, lo que también generá errores de seguimiento.

A la vista de lo anterior, se puede considerar que el seguimiento es aceptable, puesto que las trayectorias realizadas se aproximan a la prevista, y el robot no dispone de ningún sensor. Esto último implica que no se puede aplicar ningún control de trayectoria al no poder adquirir información del medio en el que se encuentra ni de su propia posición y orientación, además de las de sus articulaciones, ya que no disponen de encoders.

7

Conclusiones

Se han cumplido todos los objetivos planteados para el proyecto de manera adecuada. Con la información obtenida de los resultados de simulación se ha comprobado que la resolución de la parte cinemática del modelo ha sido correcta pese a los pequeños errores existentes en las posiciones y seguimientos. Es necesario remarcar que esos errores deben existir puesto que no existe ningún bucle de control que permita eliminarlos. Esto implica que, aún suponiendo que el sistema de la simulación fuese perfecto, que no lo es ya que se realizan pequeñas aproximaciones en los cálculos de Simulink, no sería realista asumir que no deberían existir errores en el seguimiento de las trayectorias si el sistema funciona en bucle abierto.

También se ha completado la implementación real del robot hexápodo y el sistema de comunicaciones para manejarlo de manera teleoperada con un teléfono móvil. Esta parte ha requerido tratar diferentes aspectos de diseño e impresión de piezas en 3D, pero también de solucionar los inconvenientes producidos durante el montaje y la calibración del robot. Tras completar estos aspectos fue necesario programar la cinemática y los movimientos para las extremidades, pero también la comunicación entre el microcontrolador y el teléfono móvil empleando Bluetooth Low Energy. Para resolver este aspecto fue necesario un estudio acerca del funcionamiento y particularidades de este protocolo, pero también de las librerías existentes que le dan soporte en el microcontrolador Wroom ESP32. Una vez realizada la comprobación de la correcta recepción de mensajes por parte del microcontrolador fue posible implementar la aplicación móvil empleando Thunkable X, con la que se puede manejar el robot.

Por otra parte, esta implementación ha presentado varios problemas, en primer lugar, los

movimientos en las distintas direcciones y los giros presentan errores. En concreto, los avances presentan desviaciones y los giros que se realizan no coinciden exactamente en amplitud con los valores angulares indicados para el giro. Estos errores dificultan el seguimiento de trayectorias; sin embargo, ya que el robot se maneja directamente a través de un teléfono móvil estos errores podrían corregirse manualmente en el control, aunque no sea lo ideal. Otro de los problemas que se presentan se deben a la estructura del robot y a sus motores. La estructura alrededor de algunas articulaciones, en concreto con aquellas que se encuentran en el extremo de las patas, no es lo suficientemente rígida, y debido al peso del robot, algunas de ellas se acaban doblando en dichos extremos, lo cual puede contribuir a desviar los movimientos dado que el punto donde se apoya realmente la pata no coincidiría con el teórico. El robot ha sido pesado una vez completado el montaje y se ha obtenido una masa de 2248 g para el conjunto y de 136 g para cada pata, lo que implica que la masa del cuerpo es de 1432 g. Pese a que este valor se encuentra por debajo de lo estimado, aún así el peso del robot ha resultado ser demasiado para los servomotores. Estos últimos presentan serias dificultades para soportar el peso del robot en determinadas configuraciones de las patas, lo cual empeora la estabilidad y en ocasiones causa oscilaciones en los movimientos de las extremidades. Este problema puede ser debido al control de los motores empleando la PCA9685, que imposibilita alimentarlos a su tensión máxima. Por otra parte, a pesar de este hecho, los servomotores Towerpro MG996r empleados no presentan el par adecuado para llevar a cabo algunos movimientos, especialmente si se desea extender el proyecto para tratar de circular por terrenos accidentados.

8

Posibles mejoras

Se han planteado diferentes mejoras que podrían llevarse a cabo sobre el proyecto para mejorar su resultado, y se han ordenado según su complejidad. La primera de ellas sería emplear otro método para el control de los motores, tratando así de conseguir el máximo par de los mismos. Para ello existen varias opciones, como adquirir una controladora de servomotores de altas prestaciones para sustituir a la PCA9685, o incluso obtener una que permitiese controlar los 18 motores a su máxima tensión por sí misma, dado que existen controladoras con mejores prestaciones y capacidad para 24 y 32 servomotores. Otra opción sería desarrollar otra placa que incorporase dos microcontroladores Wroom ESP32, de modo que cada uno de ellos manejase 9 motores. Una de ellos sería el principal y realizaría los cálculos para los movimientos, y finalmente le transmitiría al otro, que actuaría como secundario, las posiciones a las que debe mover las articulaciones de los 9 servomotores que controla en cada momento de la trayectoria. Esta comunicación entre los microcontroladores podría realizarse a través de los módulos Wi-Fi o Bluetooth, pero por sencillez sería recomendable establecer la comunicación serie entre ambos uniendo sus pines RX y TX de la manera correspondiente en la placa donde se introducirían los dos microcontroladores.

Otra forma de realizar la mejora sería mejorando los motores y la estructura, existen servomotores compatibles en tamaño que ofrecen mayores prestaciones, con pares de 15, 20 e incluso 30 kg·cm, como pueden ser los servomotores DS3218MG, capaces de ejercer un par máximo de 21.5 kg·cm. De igual manera, podría diseñarse una carcasa para los motores que proporcionase mayor sujeción a sus ejes y estuviese anclada de modo que dificultase que se tuerzan las patas.

Finalmente, otra mejora de interés sería incluir sensores en el modelo, aunque esto podría considerarse una ampliación del trabajo. Si se combinase la presencia de encoders en los ejes de todas las articulaciones junto con un sensor láser de balizamiento, podrían establecerse leyes de control para asegurar que los movimientos de las patas se realizan tal y como se desea, y además corregir los errores que pudiesen darse en los desplazamientos del robot empleando la información del sensor de balizamiento. Esta clase de sensores presenta el inconveniente de que las balizas deben estar fijas en el entorno, lo cual implica que sería un entorno controlado, el cual no es muy apropiado para un robot caminante, dado que una de sus ventajas es su desempeño en terrenos irregulares. Por este motivo se ha planteado una opción alternativa, empleando acelerómetros junto con empleando otros sensores estereoceptivos, es decir, sensores que obtengan información del entorno en el que se encuentra el robot, como pueden ser sensores de distancia, podría realizarse una estimación bastante acertada de la posición del robot con la que establecer la ley de control. Adicionalmente, si se incorporasen giroscopios también podría establecerse un control de equilibrio para mantener la base del robot estable en espacios accidentados.

9

Presupuesto

En este capítulo se desarrolla el presupuesto del proyecto.

9.1. Precios elementales

9.1.1. Materiales

Referencia	Unidad	Descripción	Precio
E0	ud	Microcontrolador Wroom ESP32 Dev Kit	3.200 €
E1	ud	Servomotor Towerpro MG996r	3.120 €
E2	ud	Controladora PCA9685	2.390 €
E3	ud	Batería LiPo 4500mAh 60C 7.4V	23.490 €
E4	ud	Cable de Carga XT60 to 4.0mm Banana	3.250 €
E5	ud	Adaptador de tensión 20A HV SBEC	27.400 €
E6	ud	Convertidor MP1584EN	0.890 €
E7	ud	Placa perforada 7.5x9 cm	2.340 €
E8	ud	Tira pines macho 40 PIN	0.800 €
E9	ud	Tira pines hembra 40 PIN	1.000 €
E10	m	Estaño	0.250 €
E11	gr	Plástico ABS	0.022 €
E12	ud	Tornillo M2x8.25	0.023 €
E13	ud	Tornillo M3x20	0.056 €
E14	ud	Tornillo M4x16	0.149 €
E15	ud	Tuerca M4 autoblocante	0.049 €
E16	ud	Arandela M4	0.026 €

Tabla 9.1: Precios de los materiales.

9.1.2. Mano de obra

Referencia	Unidad	Descripción	Precio
MO1	h	Ingeniero electrónico	32.600 €

Tabla 9.2: Precios de la mano de obra.

9.1.3. Maquinaria

Referencia	Unidad	Descripción	Precio
MA1	h	Estación soldadura	0.072 €
MA2	h	Taladro de columna	0.950 €
MA3	h	Equipo informático	0.080 €
MA4	h	Multímetro	0.002 €
LS1	h	Licencia Matlab	0.091 €
LS2	h	Licencia Simulink	0.137 €
LS3	h	Licencia Simscape	0.091 €
LS4	h	Licencia Simscape Multibody	0.112 €
LS5	h	Licencia SolidWorks	1.732 €

Tabla 9.3: Precios de la maquinaria, equipo y licencias.

9.1.4. Medios auxiliares

Referencia	Unidad	Descripción	Valor
AUX1	%	Medios auxiliares	0.021

Tabla 9.4: Precios de los medios auxiliares.

9.2. Precios unitarios y mediciones

Referencia	Unidad	Descripción	Precio	Cantidad	Parcial
		MATERIALES			
E0	ud	Microcontrolador Wroom ESP32 Dev Kit	3.200 €	1	3.200 €
E1	ud	Servomotor Towerpro MG996r	3.120 €	18	56.160 €
E2	ud	Controladora PCA9685	2.390 €	1	2.390 €
E3	ud	Batería LiPo 4500mAh 60C 7.4V	23.490 €	1	23.490 €
E4	ud	Cable de Carga XT60 to 4.0mm Banana	3.250 €	1	3.250 €
E5	ud	Adaptador de tensión 20A HV SBEC	27.400 €	1	27.400 €
E6	ud	Convertidor MP1584EN	0.890 €	1	0.890 €
E7	ud	Placa perforada 7.5x9 cm	2.340 €	1	2.340 €
E8	ud	Tira pines macho 40 PIN	0.800 €	2	1.600 €
E9	ud	Tira pines hembra 40 PIN	1.000 €	1	1.000 €
E10	m	Estaño	0.250 €	0.15	0.038 €
E11	gr	Plástico ABS	0.022 €	503	11.066 €
E12	ud	Tornillo M2x8.25	0.023 €	36	0.828 €
E13	ud	Tornillo M3x20	0.056 €	84	4.704 €
E14	ud	Tornillo M4x16	0.149 €	18	2.682 €
E15	ud	Tuerca M4 autoblocante	0.049 €	18	0.882 €
E16	ud	Arandela M4	0.026 €	36	0.936 €
		MANO DE OBRA			
MO1	h	Ingeniero electrónico	32.600 €	310	10106.000 €
		MAQUINARIA Y LICENCIAS			
MA1	h	Estación soldadura	0.072 €	6	4.000 €
MA2	h	Taladro de columna	0.950 €	2	1.900 €
MA3	h	Equipo informático	0.080 €	311	24.880 €
MA4	h	Multímetro	0.002 €	1	0.002 €
LS1	h	Licencia Matlab	0.091 €	160	14.560 €
LS2	h	Licencia Simulink	0.137 €	135	18.495 €
LS3	h	Licencia Simscape	0.091 €	135	12.285 €
LS4	h	Licencia Simscape Multibody	0.112 €	135	15.120 €
LS5	h	Licencia SolidWorks	1.732 €	6	10.392 €
		MEDIOS AUXILIARES			
AUX1	%	Costes directos auxiliares	2 %	10350.49 €	207.01 €
		Total precio de ejecución material			**10557.50 €**

Tabla 9.5: Precios unitarios y mediciones.

9.3. Valoración

Referencia	Descripción	Precio	%
E0	Microcontrolador Wroom ESP32 Dev Kit	3.200 €	0.030 %
E1	Servomotor Towerpro MG996r	56.160 €	0.532 %
E2	Controladora PCA9685	2.390 €	0.023 %
E3	Batería LiPo 4500mAh 60C 7.4V	23.490 €	0.222 %
E4	Cable de Carga XT60 to 4.0mm Banana	3.250 €	0.031 %
E5	Adaptador de tensión 20A HV SBEC	27.400 €	0.260 %
E6	Convertidor MP1584EN	0.890 €	0.008 %
E7	Placa perforada 7.5x9 cm	2.340 €	0.022 %
E8	Tira pines macho 40 PIN	1.600 €	0.015 %
E9	Tira pines hembra 40 PIN	1.000 €	0.009 %
E10	Estaño	0.038 €	0.000 %
E11	Plástico ABS	11.066 €	0.105 %
E12	Tornillo M2x8.25	0.828 €	0.008 %
E13	Tornillo M3x20	4.704 €	0.045 %
E14	Tornillo M4x16	2.682 €	0.025 %
E15	Tuerca M4 autoblocante	0.882 €	0.008 %
E16	Arandela M4	0.936 €	0.009 %
MO1	Ingeniero electrónico	10106.000 €	95.723 %
MA1	Estación soldadura	4.000 €	0.038 %
MA2	Taladro de columna	1.900 €	0.018 %
MA3	Equipo informático	24.880 €	0.236 %
MA4	Multímetro	0.002 €	0.000 %
LS1	Licencia Matlab	14.560 €	0.138 %
LS2	Licencia Simulink	18.495 €	0.175 %
LS3	Licencia Simscape	12.285 €	0.116 %
LS4	Licencia Simscape Multibody	15.120 €	0.143 %
LS5	Licencia SolidWorks	10.392 €	0.098 %
AUX1	Medios auxiliares	207.010 €	1.961 %

Tabla 9.6: Porcentaje del presupuesto correspondiente a cada factor.

PRESUPUESTO DE EJECUCIÓN MATERIAL	10557.499 €
12 % Gastos Generales	1266.900 €
6 % de Beneficio Industrial	633.450 €
PRESUPUESTO	12457.849 €
21 % de IVA	2616.148 €
PRESUPUESTO + IVA	**15074.00 €**

Tabla 9.7: Resumen del presupuesto del proyecto.

10

Bibliografía

[1] KUKA titan. URL: https://www.kuka.com/en-de/products/robot-systems/industrial-robots/kr-1000-titan.

[2] Spyros G. Mouroutsos and Eleftheria Mitka. Classification of domestic robots. 2012. URL: https://www.researchgate.net/publication/236031948_Classification_of_domestic_robots.

[3] Mars Curiosity Rover. URL: https://mars.nasa.gov/msl/home/.

[4] Robots Quirúrgicos Da Vinci. URL: https://www.davincisurgery.com/da-vinci-systems/about-da-vinci-systems.

[5] Teleautonomous Bomb Disposal Research. URL: https://clearpathrobotics.com/husky-bomb-disposal-teleautonomy-hdms/.

[6] Cobot Trends Staff. Cobots vs. industrial robots: what are the differences? *Collaborative Robotics Trends*. URL: https://www.cobottrends.com/cobots-vs-industrial-robots-what-are-differences/.

[7] Brett Israel. Wall-jumping robot is most vertically agile ever built. *Berkeley News*, 2016. URL: https://news.berkeley.edu/2016/12/06/wall-jumping-robot-is-most-vertically-agile-ever-built/.

[8] Lecacy Robots. URL: https://www.bostondynamics.com.

[9] MorpHex. URL: http://zentasrobots.com/robot-projects/morphex-mkii/.

[10] Matt Simon. What Boston Dynamics Rolling Handle Robot Really Means. *Wired*, 2019. URL: https://www.wired.com/story/what-boston-dynamics-rolling-handle-robot-really-means/.

[11] W. H. C. Bassetti. *Robotics and Automation Handbook*. 2005.

[12] Alexander Gil and Javier Román. Antdroid, Fully Open Source Hexapod. URL: https://antdroid.grigri.cloud/.

[13] Guardforce. Types of Robots and Their Applications. *Guardforce*, 2021. URL: https://www.guardforce.com.hk/en/news/blog_115/Types-of-Robots-and-Their-Applications_1729.

[14] Carlos Llopis-Albert Francisco Rubio, Francisco Valero. A review of mobile robots: Concepts, methods, theoretical framework, and applications. *SAGE Journals*, 2019.

[15] Marc Raibert. *Legged Robots That Balance*. MIT Press, 1986.

[16] George A. Bekey. *Autonomous Robots: From Biological Inspiration to Implementation and Control*. MIT Press, 2005.

[17] Jennifer Chu. Mini cheetah is the first four-legged robot to do a backflip. *MIT News*, 2019. URL: https://news.mit.edu/2019/mit-mini-cheetah-first-four-legged-robot-to-backflip-0304.

[18] Evan Ackerman. Six-Legged Robot One-Ups Nature With Faster Gait. *IEEE Spectrum*, 2017.

[19] Joseph Stiles Beggs. *Kinematics*. Taylor and Francis, 1983.

[20] *MathWorks Help Center*.

[21] Sistemas de Medida Fricción. URL: https://www.medidafuerzaytorsion.com/index.php/es/equipos-de-medicion/category/52-medida-de-friccion/lang-es-ES?jjj=1623606246600.

[22] Ravi Teja. How to use BLE in ESP32? ESP32 BLE(Bluetooth Low Energy) Tutorial. *Electronics Hub*, 2021. URL: https://www.electronicshub.org/esp32-ble-tutorial/.

[23] ESP32 BLE Arduino. URL: https://github.com/nkolban/ESP32_BLE_Arduino.

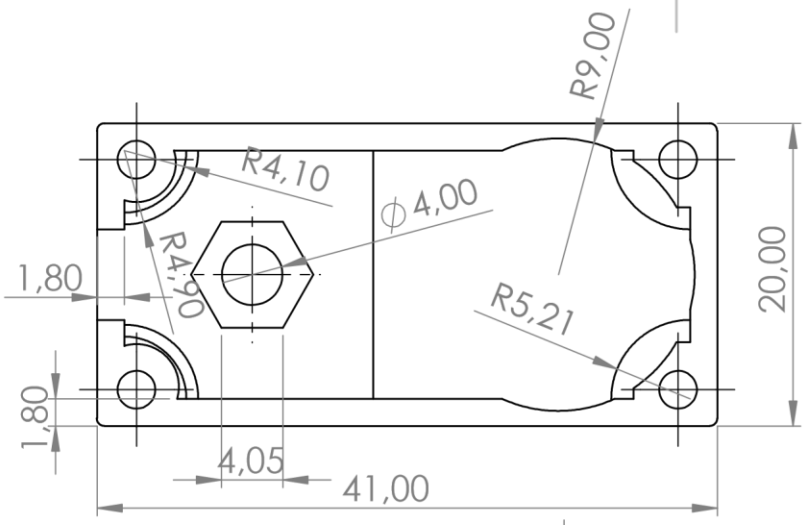
[24] Asociación Española de Normalización y Certificación. AENOR. Normas UNE, 1986. URL: https://portal.aenormas.aenor.com/aenor/.

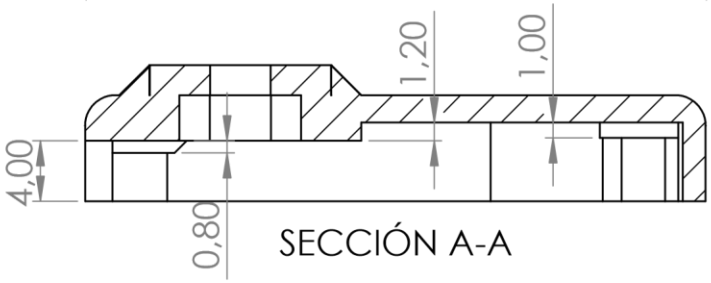
[25] Asociación Española de Normalización y Certificación.BOE. Legislación, 1959. URL: https://www.boe.es/.

[26] JEDEC, Global Standars and Documents. URL: https://www.jedec.org/standards-documents.

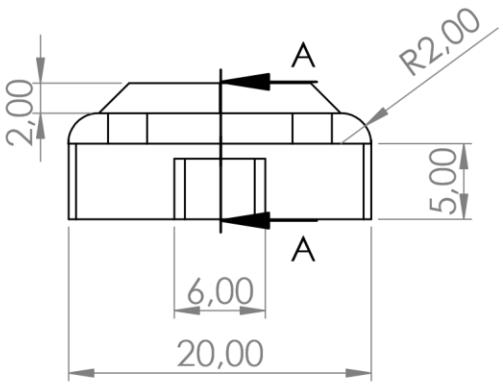
A

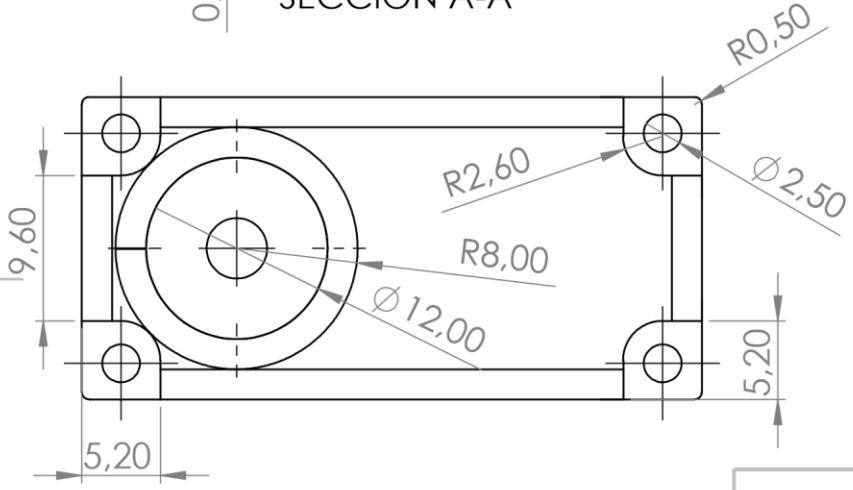
Planos



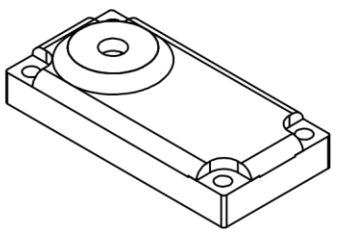


SECCIÓN A-A





Vista Isométrica Escala: 1:1



1	18	Tapa de servomotor	ISO 19062	ABS
Marca	**N.º Pieza**	**Designación y observaciones**	**Norma**	**Material**

	NOMBRE	FECHA
DIBUJ:	David Laseca Pérez	14/06/2021
VERIF:		
APROB:		

CONJUNTO: PIEZAS ESTRUCTURA HEXÁPODO
TITULAR: Universidad Politécnica de Valencia
EMPRESA: EQUIPO INGENIEROS G15 S.L.
REFERENCIA: 2021-02-01

ESCALA: 2:1

Tapa de servomotor modificada

N.º PLANO: 1

B

Pliego de condiciones

B.1. Objeto

La presente especificación técnica se refiere a la fabricación del robot hexápodo de modelo Antdroid[12] empleado para el estudio de esta clase de robots y sus movimientos. Se excluye del documento la especificación de la parte de diseño, al no encontrarse dentro del alcance del proyecto.

Asimismo, se procederá a la descripción de los materiales utilizados, los procesos de ejecución y el control de calidad a seguir, así como una prueba de servicio general para la comprobación del correcto funcionamiento del robot. Por otra parte, se supondrá que el equipo necesario para los procesos y los materiales adquiridos a empresas externas han superado un control de calidad previo.

B.2. Normativa

En todo momento, se deberá cumplir la legislación española y europea, así como los estándares internacionales a los que está sujeto el proyecto. De este modo se deben seguir las siguientes normativas[24][25][26]:

- **Real Decreto 1215/97, de 18 de julio.** Se establecen las condiciones mínimas de salud y seguridad que deben cumplirse en el uso de puestos de trabajo.

- **Real Decreto 208/2005, de 25 de febrero.** Se trata la gestión de residuos de aparatos eléctricos y electrónicos.
- **Ley 12/1986, de 1 de abril.** Se definen las atribuciones de los Ingenieros técnicos.
- **Ley Orgánica 5/2002, de 19 de junio.** Se establecen las cualificaciones profesionales.
- **UNE-EN ISO/IEC 80000.** Se define una guía para el uso de magnitudes físicas y de medida, así como las fórmulas que las incluyen, en documentos de carácter técnico o educativo.
- **UNE-EN IEC 60027.** Se establece un estándar internacional para el uso de símbolos literales en electrotecnia.
- **UNE-EN IEC 60086.** Se indican estándares de baterías primarias en lo referente a dimensiones, nomenclatura, configuración y otros aspectos de las mismas, incluidos de seguridad y medioambientales.
- **UNE-EN IEC 60469.** Se definen términos para las señales de pulsos y características para la comunicación eficiente de información.
- **UNE-EN IEC 60749.** Se definen los métodos para las pruebas mecánicas y térmicas de los dispositivos de semiconductores.
- **UNE-EN IEC 61190.** Se establecen materiales de fijación para conjuntos electrónicos y requisitos para aleaciones de soldadura electrónica y soldaduras sólidas con y sin fundente para aplicaciones electrónicas.
- **UNE-EN IEC 61960.** Se especifican características y pruebas de servicio para baterías con células de ión de litio.
- **IPC 2221.** Se comentan los estándares de diseño de placas de circuitos electrónicos impresos.
- **UNE-EN ISO 898.** Se establecen las características del acero al carbono y el acero aleado con el que deben hacerse pernos, tornillos y bulones.
- **UNE-EN ISO 1043.** Se comenta la nomenclatura estandarizada de plásticos.
- **UNE-EN ISO 13482.** Se establecen especificaciones y guías para el diseño de robots y sistemas robóticos no industriales y de asistencia no médica.
- **UNE-EN ISO 19062.** Se especifican los métodos de preparación de muestras y de pruebas que deben usarse para determinar las propiedades del acrilonitrilo-butadieno-estireno (ABS) para moldeo y extrusión.
- **ISO 8373.** Se definen términos usados en relación con robots y sistemas robóticos operando en entornos industriales y no industriales.
- **ISO 9787.** Se definen y especifican sistemas de coordenadas para robots. También incluye nomenclatura sobre los movimientos básicos de robots.
- **ISO 19649.** Se definen términos relativos a robots móviles que operan sobre superficies sólidas en aplicaciones industriales y de servicio.
- **ISO DIN 13.** Se definen tamaños y métricas normalizadas para tornillos y tuercas.

B.3. Condiciones de los materiales

B.3.1. Placa perforada

Descripción

La placa placa base será de fibra de vidrio (FR-4) con resina epoxy que será resistente al fuego. Las soldaduras serán realizadas con una aleación de 60 % de estaño, 38 % de plomo y 2 % de cobre con adición de resina, formando estaño homologado para soldar. Además, los puntos de soldadura de la placa deben ser uniformes.

Control de calidad

Antes del uso de la placa, se asegurará que las pistas de soldadura que no han de estar unidas se encuentran separadas por una distancia mínima de 0.05 mm. Asimismo, se comprobará que estas pistan presentan una sección aproximadamente constante e igual a las demás.

B.3.2. Batería

Descripción

La batería será de polímero de iones de litio (LiPo), de dimensiones no superiores a 136 x 45 x 21 mm, y de masa no superior a 220 gramos. Asimismo, será recargable, con una tensión nominal mínima de 7.4 V y una capacidad no inferior a 4500 mAh.

Control de calidad

La batería cumplirá la normativa y restricciones definidas en la norma UNE-EN IEC 61960.

B.3.3. Piezas de la estructura

Descripción

Se realizarán mediante impresión 3D empleando plástico de tipo ABS.

Control de calidad

Se comprobará que no existan grietas en las piezas tras su fabricación, que podrían provocar roturas, y que la masa total de las piezas no excede 530 g.

El plástico ABS estará estandarizado según la norma UNE-EN ISO 19062.

B.3.4. Tornillería

Descripción

Los tornillos deben pertenecer a las métricas estandarizadas adecuadas en cada caso, según la aplicación que tengan dentro del proyecto.

Control de calidad

Los tornillos se adecuará al estándar definido por la norma ISO DIN 13, y el acero del que están formados seguirá la norma UNE-EN ISO 898.

B.4. Condiciones de ejecución

B.4.1. Circuito sobre la placa perforada

Descripción

Se utilizarán herramientas y guantes de protección mientras se manipule, se realicen las conexiones sobre la placa o siempre que sea preciso. La soldadura se llevará a cabo en un banco con capacidad de regulación de temperatura de soldado, siendo la temperatura máxima de 275°C. El soldado de la placa se realizará de la manera siguiente, se colocarán en primer lugar los pines hembra para introducir el microcontrolador, seguidamente se pondrán los pines macho para la alimentación y los servomotores, después se conectará el convertidor MP1584EN y finalmente se unirán los pines que deban estar conectados mediante soldaduras consecutivas o puentes.

Control de calidad

Se procederá al control de calidad de cada sección después de su soldadura, consistiendo en

una comprobación de la continuidad de cada sección y la realización de una prueba final incorporando la alimentación a la placa, donde se evaluará la tensión presente en los distintos puntos. Asimismo, las soldaduras y los materiales utilizados deberán de cumplir la normativa UNE-EN IEC 61190. Los circuitos integrados y semiconductores empleados estarán normalizados según JEDEC[26] y la norma UNE-EN 60749.

B.4.2. Estructura del robot

Descripción

El montaje del robot se realizará por partes, y además de seguir los pasos descritos en este apartado, se deberá consultar la guía de montaje del modelo Antroid[12]. En primer lugar, se posicionarán todos los servomotores en su posición intermedia, para después introducirlos en las piezas que los sostienen. Para realizar este paso se retirará la tapa que llevan por defecto, prestando especial cuidado a que no se desmonte la reductora mecánica que contienen, y se introducirán en dichas piezas. Una vez introducidas, se colocarán las tapas modificadas, a las cuales se les habrá introducido previamente un tornillo M4x16.

Una vez preparados los servomotores, deben colocarse los discos a los que se unirán los ejes de estos sobre las piezas correspondientes, la cubierta superior y los arcos intermedios de la extremidades. Para ello se usarán dos tornillos M2x8.25 por cada disco. Tras colocar los discos se procederá a unir las distintas articulaciones, empezando por las seis extremidades del robot. Se incorporarán las piezas con los discos a los ejes de los servomotores en la posición base de la extremidad y se emplearán los tornillos proporcionados por los servomotores para asegurar la unión de las piezas. Una vez se haya completado este proceso para todas los motores de las extremidades, se deberá emplear introducir una arandela en cada tornillo de la tapa de los mismos. Después de la arandela, se colocarán en su posición base y se empleará otra arandela y una tuerca autoblocante para completar la articulación. Tras repetir este proceso en las seis estremidades, estas se deberán unir al cuerpo.

Para realizar esta unión, se colocarán los ejes de los motores de la base de cada extremidad en su posición adecuada de la cubierta superior. Esto implica situar la extremidad con una orientación adecuada de 45^{o} para el caso de las delanteras y de -45^{o} para las traseras, mientras que las intermedias deben quedar perpendiculares al cuerpo del robot o con una orientación

de 0^{o}. Tras colocar las patas se asegurará la unión con los tornillos proporcionados por los servomotores, para posteriormente incorporar los separadores de la base en la cubierta superior. A continuación se introducirá una arandela en cada eje de los servomotores de la base de las extremidades, que ya se encontrarán unidos a la cubierta superior y se incorporará la cubierta inferior. Finalmente se introducirá otra arandela y la tuerca autoblocante en cada uno de los ejes de los servomotores, los cuales asomarán tras colocar la cubierta inferior.

Control de calidad

Para asegurar la correcta realización del montaje debe llevarse al robot a su posición base, empleando para ello un código de calibración. Tras realizar este paso, se comprobará que, tras añadir la batería y la electrónica, soporta su propio peso correctamente manteniendo las patas en posición vertical, es decir, sin que estas se doblen.

B.5. Prueba de servicio

La prueba de servicio del robot consistirá en emplear el código desarrollado para el microcontrolador Wroom ESP32, que deberá cargarse a la placa del robot, junto con la aplicación móvil desarrollada para evaluar el comportamiento del robot en una prueba real.

Se asegurará que se realiza la conexión con el robot de manera correcta y que el robot ejecuta los movimientos correspondientes a cada botón de la aplicación. También se comprobará que el robot no pueda acceder a la posición de reposo si no se encuentra detenido antes y que es capaz de realizar todos los movimientos y giros sin tropezar. Mientras realiza los movimientos se debe poder detener manualmente al robot si tropieza, para evitar daños en la estructura y los servomotores. Durante la ejecución de las pruebas debe también observarse detenidamente la posición de las patas durante el movimiento con el objetivo de detectar si se realiza algún movimiento inadecuado, en cuyo caso se debe tratar de solucionarlo mediante cambios en la calibración, del mismo modo que debe realizarse si el robot tropieza.

C

Código de Matlab

```
% Alumno: David Laseca Perez
%% Inicio
Tini = 0;
%T = 0.0005;
T =  0.001;

params = struct('h1', 17.8, 'l1', 55, 'l2', 70, 'l3', 166.66);

alturaBase = 184.46;

h = 30;
avance = 60;
ang = pi / 6;

RateLimiter = 3;
damping = 0.0001;
st = 1e4;
dm = 1e2;
sf = 0.3;
df = 0.25;

origenPata1 = struct('x',  77.8, 'y',  54.5);
origenPata2 = struct('x',  77.8, 'y', -54.5);
origenPata3 = struct('x',     0, 'y',  78.0);
origenPata4 = struct('x',     0, 'y', -78.0);
origenPata5 = struct('x', -77.8, 'y',  54.5);
origenPata6 = struct('x', -77.8, 'y', -54.5);

direccion = 0;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));

%% Posicion base
p0 = [struct('x',  88.3883, 'y', -88.3883, 'z',  alturaBase);
      struct('x',  88.3883, 'y', -88.3883, 'z', -alturaBase);
      struct('x',        0, 'y',     -125, 'z',  alturaBase);
      struct('x',        0, 'y',     -125, 'z', -alturaBase);
      struct('x', -88.3883, 'y', -88.3883, 'z',  alturaBase);
      struct('x', -88.3883, 'y', -88.3883, 'z', -alturaBase)];

%% Posiciones avance segun la direccion
```

```
[p1, p2] = puntosAvance(p0, avance, vectorDireccion);

%% Movimientos
% Configuracion inicial
q1 = [  pi/4, 0, 0];
q2 = [  pi/4, 0, 0];
q3 = [     0, 0, 0];
q4 = [     0, 0, 0];
q5 = [- pi/4, 0, 0];
q6 = [- pi/4, 0, 0];
t = 0;
% Avance recto
movs1 = moveLI(q1, p0(1), params, t, T, t + 1);
movs2 = moveLD(q2, p0(2), params, t, T, t + 1);
movs3 = moveLI(q3, p0(3), params, t, T, t + 1);
movs4 = moveLD(q4, p0(4), params, t, T, t + 1);
movs5 = moveLI(q5, p0(5), params, t, T, t + 1);
movs6 = moveLD(q6, p0(6), params, t, T, t + 1);
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, p0(2), params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, p0(3), params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, p0(6), params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
movs1 = [movs1; moveLI(q1, p1(1), params, t, T, t + 1)];
movs2 = [movs2; moveCD(q2, p1(2), h, params, t, T, t + 1)];
movs3 = [movs3; moveCI(q3, p1(3), h, params, t, T, t + 1)];
movs4 = [movs4; moveLD(q4, p1(4), params, t, T, t + 1)];
movs5 = [movs5; moveLI(q5, p1(5), params, t, T, t + 1)];
movs6 = [movs6; moveCD(q6, p1(6), h, params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
for i = 1:2    % TFinSecuencia = 3 + ifinal * 3
    movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
    movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
    movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
    movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
    movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
    movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
```

```
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
    movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
    movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
    movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
    movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
    movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
    movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
end % t = 9

% Avance derecha
direccion = -pi/2;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
paux2 = p1(2);
paux3 = p1(3);
paux6 = p1(6);
[p1, p2] = puntosAvance(p0, avance, vectorDireccion);
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
for i = 1:2
    movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
    movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
    movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
    movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
    movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
    movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
    movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
    movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
    movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
    movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
    movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
    movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
```

```
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
end % t = 16

% Avance hacia atras
direccion = pi;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
paux2 = p1(2);
paux3 = p1(3);
paux6 = p1(6);
[p1, p2] = puntosAvance(p0, avance, vectorDireccion);
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
for i = 1:2
    movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
    movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
    movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
    movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
    movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
    movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
    movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
    movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
    movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
    movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
    movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
    movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
end % t = 23

% Avance izquierda
direccion = pi/2;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
paux2 = p1(2);
paux3 = p1(3);
```

```
paux6 = p1(6);
[p1, p2] = puntosAvance(p0, avance, vectorDireccion);
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
for i = 1:2
    movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
    movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
    movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
    movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
    movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
    movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
    movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
    movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
    movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
    movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
    movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
    movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
end % t = 30

% Avance atras-izquierda
direccion = 3*pi/4;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
paux2 = p1(2);
paux3 = p1(3);
paux6 = p1(6);
[p1, p2] = puntosAvance(p0, avance, vectorDireccion);
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
```

```
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
for i = 1:2
    movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
    movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
    movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
    movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
    movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
    movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
    movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
    movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
    movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
    movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
    movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
    movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
    q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
    q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
    q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
    q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
    t = t + 1.5;
end % t = 37

% Avance arriba-izquierda
direccion = pi/4;
vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
paux2 = p1(2);
paux3 = p1(3);
paux6 = p1(6);
[p1, p2] = puntosAvance(p0, avance, vectorDireccion);
movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
for i = 1:2
    movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
    movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
    movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
    movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
    movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
    movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
    q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
    q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
```

```
346     q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
347     q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
348     q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
349     q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
350     t = t + 1.5;
351     movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
352     movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
353     movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
354     movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
355     movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
356     movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
357     q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
358     q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
359     q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
360     q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
361     q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
362     q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
363     t = t + 1.5;
364 end % t = 44
365
366 % Avance arriba-derecha
367 direccion = -pi/4;
368 vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
369 paux2 = p1(2);
370 paux3 = p1(3);
371 paux6 = p1(6);
372 [p1, p2] = puntosAvance(p0, avance, vectorDireccion);
373 movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
374 movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
375 movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
376 movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
377 movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
378 movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
379 q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
380 q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
381 q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
382 q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
383 q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
384 q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
385 t = t + 1;
386 for i = 1:2
387     movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
388     movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
389     movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
390     movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
391     movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
392     movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
393     q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
394     q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
395     q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
396     q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
397     q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
398     q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
399     t = t + 1.5;
400     movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
401     movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
402     movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
403     movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
404     movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
405     movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
406     q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
```

```
407     q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
408     q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
409     q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
410     q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
411     q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
412     t = t + 1.5;
413 end % t = 51
414
415 % Avance atras-derecha
416 direccion = -3*pi/4;
417 vectorDireccion = struct('x', cos(direccion), 'y', sin(direccion));
418 paux2 = p1(2);
419 paux3 = p1(3);
420 paux6 = p1(6);
421 [p1, p2] = puntosAvance(p0, avance, vectorDireccion);
422 movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1)];
423 movs2 = [movs2; moveLD(q2, paux2, params, t, T, t + 1)];
424 movs3 = [movs3; moveLI(q3, paux3, params, t, T, t + 1)];
425 movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1)];
426 movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1)];
427 movs6 = [movs6; moveLD(q6, paux6, params, t, T, t + 1)];
428 q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
429 q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
430 q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
431 q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
432 q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
433 q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
434 t = t + 1;
435 for i = 1:2
436     movs1 = [movs1; moveLI(q1, p2(1), params, t, T, t + 1.5)];
437     movs2 = [movs2; moveCD(q2, p2(2), h, params, t, T, t + 1.5)];
438     movs3 = [movs3; moveCI(q3, p2(3), h, params, t, T, t + 1.5)];
439     movs4 = [movs4; moveLD(q4, p2(4), params, t, T, t + 1.5)];
440     movs5 = [movs5; moveLI(q5, p2(5), params, t, T, t + 1.5)];
441     movs6 = [movs6; moveCD(q6, p2(6), h, params, t, T, t + 1.5)];
442     q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
443     q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
444     q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
445     q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
446     q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
447     q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
448     t = t + 1.5;
449     movs1 = [movs1; moveCI(q1, p1(1), h, params, t, T, t + 1.5)];
450     movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1.5)];
451     movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1.5)];
452     movs4 = [movs4; moveCD(q4, p1(4), h, params, t, T, t + 1.5)];
453     movs5 = [movs5; moveCI(q5, p1(5), h, params, t, T, t + 1.5)];
454     movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1.5)];
455     q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
456     q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
457     q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
458     q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
459     q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
460     q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
461     t = t + 1.5;
462 end % t = 58
463
464 % Giro hacia la derecha (q2, q3 y q6 empiezan en p1)
465 [p13, p14] = giroI(p0(1), -ang, origenPata1);
466 [p23, p24] = giroD(p0(2),  ang, origenPata2);
467 [p33, p34] = giroI(p0(3), -ang, origenPata3);
```

```
[p43, p44] = giroD(p0(4),  ang, origenPata4);
[p53, p54] = giroI(p0(5), -ang, origenPata5);
[p63, p64] = giroD(p0(6),  ang, origenPata6);

p3 = [p13; p23; p33; p43; p53; p63];
p4 = [p14; p24; p34; p44; p54; p64];

movs1 = [movs1; moveCI(q1, p3(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, p1(2), params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, p1(3), params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p3(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p3(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, p1(6), params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
movs1 = [movs1; moveGI(q1, -ang, origenPata1, params, t, T, t + 1.5)];
movs2 = [movs2; moveCD(q2, p3(2), h, params, t, T, t + 1.5)];
movs3 = [movs3; moveCI(q3, p3(3), h, params, t, T, t + 1.5)];
movs4 = [movs4; moveGD(q4,  ang, origenPata4, params, t, T, t + 1.5)];
movs5 = [movs5; moveGI(q5, -ang, origenPata5, params, t, T, t + 1.5)];
movs6 = [movs6; moveCD(q6, p3(6), h, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1.5;
movs1 = [movs1; moveCI(q1, p3(1), h, params, t, T, t + 1.5)];
movs2 = [movs2; moveGD(q2,  ang, origenPata2, params, t, T, t + 1.5)];
movs3 = [movs3; moveGI(q3, -ang, origenPata3, params, t, T, t + 1.5)];
movs4 = [movs4; moveCD(q4, p3(4), h, params, t, T, t + 1.5)];
movs5 = [movs5; moveCI(q5, p3(5), h, params, t, T, t + 1.5)];
movs6 = [movs6; moveGD(q6,  ang, origenPata6, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1.5;
movs1 = [movs1; moveGI(q1, -ang, origenPata1, params, t, T, t + 1.5)];
movs2 = [movs2; moveCD(q2, p3(2), h, params, t, T, t + 1.5)];
movs3 = [movs3; moveCI(q3, p3(3), h, params, t, T, t + 1.5)];
movs4 = [movs4; moveGD(q4,  ang, origenPata4, params, t, T, t + 1.5)];
movs5 = [movs5; moveGI(q5, -ang, origenPata5, params, t, T, t + 1.5)];
movs6 = [movs6; moveCD(q6, p3(6), h, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1.5;
% t = 63.5
```

```
% Giro hacia la izquierda (q2, q3 y q6 empiezan en p3)
movs1 = [movs1; moveCI(q1, p4(1), h, params, t, T, t + 1)];
movs2 = [movs2; moveLD(q2, p3(2), params, t, T, t + 1)];
movs3 = [movs3; moveLI(q3, p3(3), params, t, T, t + 1)];
movs4 = [movs4; moveCD(q4, p4(4), h, params, t, T, t + 1)];
movs5 = [movs5; moveCI(q5, p4(5), h, params, t, T, t + 1)];
movs6 = [movs6; moveLD(q6, p3(6), params, t, T, t + 1)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1;
movs1 = [movs1; moveGI(q1,  ang, origenPata1, params, t, T, t + 1.5)];
movs2 = [movs2; moveCD(q2, p4(2), h, params, t, T, t + 1.5)];
movs3 = [movs3; moveCI(q3, p4(3), h, params, t, T, t + 1.5)];
movs4 = [movs4; moveGD(q4, -ang, origenPata4, params, t, T, t + 1.5)];
movs5 = [movs5; moveGI(q5,  ang, origenPata5, params, t, T, t + 1.5)];
movs6 = [movs6; moveCD(q6, p4(6), h, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1.5;
movs1 = [movs1; moveCI(q1, p4(1), h, params, t, T, t + 1.5)];
movs2 = [movs2; moveGD(q2, -ang, origenPata2, params, t, T, t + 1.5)];
movs3 = [movs3; moveGI(q3,  ang, origenPata3, params, t, T, t + 1.5)];
movs4 = [movs4; moveCD(q4, p4(4), h, params, t, T, t + 1.5)];
movs5 = [movs5; moveCI(q5, p4(5), h, params, t, T, t + 1.5)];
movs6 = [movs6; moveGD(q6, -ang, origenPata6, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1.5;
movs1 = [movs1; moveGI(q1,  ang, origenPata1, params, t, T, t + 1.5)];
movs2 = [movs2; moveCD(q2, p4(2), h, params, t, T, t + 1.5)];
movs3 = [movs3; moveCI(q3, p4(3), h, params, t, T, t + 1.5)];
movs4 = [movs4; moveGD(q4, -ang, origenPata4, params, t, T, t + 1.5)];
movs5 = [movs5; moveGI(q5,  ang, origenPata5, params, t, T, t + 1.5)];
movs6 = [movs6; moveCD(q6, p4(6), h, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
t = t + 1.5;
movs1 = [movs1; moveCI(q1, p4(1), h, params, t, T, t + 1.5)];
movs2 = [movs2; moveGD(q2, -ang, origenPata2, params, t, T, t + 1.5)];
movs3 = [movs3; moveGI(q3,  ang, origenPata3, params, t, T, t + 1.5)];
movs4 = [movs4; moveCD(q4, p4(4), h, params, t, T, t + 1.5)];
movs5 = [movs5; moveCI(q5, p4(5), h, params, t, T, t + 1.5)];
movs6 = [movs6; moveGD(q6, -ang, origenPata6, params, t, T, t + 1.5)];
q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
```

```
590 q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
591 q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
592 q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
593 q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
594 t = t + 1.5;
595 movs1 = [movs1; moveGI(q1,  ang, origenPata1, params, t, T, t + 1.5)];
596 movs2 = [movs2; moveCD(q2, p4(2), h, params, t, T, t + 1.5)];
597 movs3 = [movs3; moveCI(q3, p4(3), h, params, t, T, t + 1.5)];
598 movs4 = [movs4; moveGD(q4, -ang, origenPata4, params, t, T, t + 1.5)];
599 movs5 = [movs5; moveGI(q5,  ang, origenPata5, params, t, T, t + 1.5)];
600 movs6 = [movs6; moveCD(q6, p4(6), h, params, t, T, t + 1.5)];
601 q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
602 q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
603 q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
604 q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
605 q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
606 q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
607 t = t + 1.5;
608 movs1 = [movs1; moveCI(q1, p4(1), h, params, t, T, t + 1.5)];
609 movs2 = [movs2; moveGD(q2, -ang, origenPata2, params, t, T, t + 1.5)];
610 movs3 = [movs3; moveGI(q3,  ang, origenPata3, params, t, T, t + 1.5)];
611 movs4 = [movs4; moveCD(q4, p4(4), h, params, t, T, t + 1.5)];
612 movs5 = [movs5; moveCI(q5, p4(5), h, params, t, T, t + 1.5)];
613 movs6 = [movs6; moveGD(q6, -ang, origenPata6, params, t, T, t + 1.5)];
614 q1 = [movs1(end,2), movs1(end,4), movs1(end,6)];
615 q2 = [movs2(end,2), movs2(end,4), movs2(end,6)];
616 q3 = [movs3(end,2), movs3(end,4), movs3(end,6)];
617 q4 = [movs4(end,2), movs4(end,4), movs4(end,6)];
618 q5 = [movs5(end,2), movs5(end,4), movs5(end,6)];
619 q6 = [movs6(end,2), movs6(end,4), movs6(end,6)];
620 t = t + 1.5;
621 % t = 73.5
622
623 %% Generar los movimientos para la simulacion
624 q11 = movs1(1:end,1:2);
625 q12 = movs1(1:end,3:4);
626 q13 = movs1(1:end,5:6);
627 q21 = movs2(1:end,1:2);
628 q22 = movs2(1:end,3:4);
629 q23 = movs2(1:end,5:6);
630 q31 = movs3(1:end,1:2);
631 q32 = movs3(1:end,3:4);
632 q33 = movs3(1:end,5:6);
633 q41 = movs4(1:end,1:2);
634 q42 = movs4(1:end,3:4);
635 q43 = movs4(1:end,5:6);
636 q51 = movs5(1:end,1:2);
637 q52 = movs5(1:end,3:4);
638 q53 = movs5(1:end,5:6);
639 q61 = movs6(1:end,1:2);
640 q62 = movs6(1:end,3:4);
641 q63 = movs6(1:end,5:6);
642
643 %% Fin de la simulacion
644 Tfin = t;
645
646 %% Funcion para crear la matriz de angulos y tiempos
647 function v = moveMotor(qObj, Tini, T, Tfin)
648     aux1 = [];
649     aux2 = [];
650     i = Tini;
```

```
651     while i < Tfin
652         aux1 = [aux1, i];
653         aux2 = [aux2, real(q0bj)];
654         i = i + T;
655     end
656     v = [aux1' aux2'];
657 end
658 
659 %% Funciones lado izquierdo
660 
661 function v = moveJI(pObj, params, Tini, T, Tfin)
662     [q1, q2, q3] = inverseKinI(pObj, params);
663     v = moveMotor(q1, Tini, T, Tfin);
664     v = [v, moveMotor(q2, Tini, T, Tfin)];
665     v = [v, moveMotor(q3, Tini, T, Tfin)];
666 end
667 
668 function v = moveLI(q0, pObj, params, Tini, T, Tfin)
669     p0 = directKinI(q0, params);
670     distEuc = sqrt((pObj.x - p0.x)^2 + (pObj.y - p0.y)^2 + (pObj.z - p0.z)^2);
671     if distEuc < 0.0000005 && distEuc > -0.0000005
672         vUnitario.x = 0;
673         vUnitario.y = 0;
674         vUnitario.z = 0;
675     else
676         vUnitario.x = (pObj.x - p0.x) / distEuc;
677         vUnitario.y = (pObj.y - p0.y) / distEuc;
678         vUnitario.z = (pObj.z - p0.z) / distEuc;
679     end
680 
681     a = -2 * distEuc / (Tfin - Tini)^3; % Trayectoria cubica
682     b = 3 * distEuc / (Tfin - Tini)^2;
683 
684     iteracionFinal = (Tfin - Tini) / T;
685     Tactual = Tini;
686     v = [];
687     i = 1;
688     while i <= iteracionFinal
689         t = i * T;
690         s = a * t^3 + b * t^2;
691         p.x = s * vUnitario.x + p0.x;
692         p.y = s * vUnitario.y + p0.y;
693         p.z = s * vUnitario.z + p0.z;
694         v = [v; moveJI(p, params, Tactual, T, Tactual + T)];
695         i = i + 1;
696         Tactual = Tactual + T;
697     end
698 end
699 
700 function v = moveCI(q0, pObj, h, params, Tini, T, Tfin)
701     p0 = directKinI(q0, params);
702     dist = sqrt((pObj.x - p0.x)^2 + (pObj.y - p0.y)^2);
703     vUnitario.x = (pObj.x - p0.x) / dist;
704     vUnitario.y = (pObj.y - p0.y) / dist;
705 
706     a = -2 * dist / (Tfin - Tini)^3; % Trayectoria cubica
707     b = 3 * dist / (Tfin - Tini)^2;
708 
709     A = - 4 * h;
710     B = - A;
```

```
712     iteracionFinal = (Tfin - Tini) / T;
713     Tactual = Tini;
714     v = [];
715     i = 1;
716     while i <= iteracionFinal
717         t = i * T;
718         s = a * t^3 + b * t^2;
719         p.x = s * vUnitario.x + p0.x;
720         p.y = s * vUnitario.y + p0.y;
721         p.z = p0.z - (A * (s / dist)^2 + B * s / dist);
722         v = [v; moveJI(p, params, Tactual, T, Tactual + T)];
723         i = i + 1;
724         Tactual = Tactual + T;
725     end
726 end
727 
728 function v = moveGI(q0, ang, origen, params, Tini, T, Tfin)
729     posPata0 = directKinI(q0, params);
730     posOrigen0.x = origen.x + posPata0.x;
731     posOrigen0.y = origen.y - posPata0.y;
732 
733     r = sqrt(posOrigen0.x^2 + posOrigen0.y^2);
734     alfa0 = atan2(posOrigen0.x, posOrigen0.y);
735 
736     a = -2 * ang / (Tfin - Tini)^3; % Trayectoria cubica
737     b = 3 * ang / (Tfin - Tini)^2;
738 
739     iteracionFinal = (Tfin - Tini) / T;
740     Tactual = Tini;
741     v = [];
742     i = 1;
743     while i <= iteracionFinal
744         t = i * T;
745         s = a * t^3 + b * t^2;
746 
747         alfa = alfa0 + s;
748         posOrigen.x = r * sin(alfa);
749         posOrigen.y = r * cos(alfa);
750 
751         posPata.x = posOrigen.x - origen.x;
752         posPata.y = origen.y - posOrigen.y;  % -(posOrigen.y - origen.y);
753         posPata.z = posPata0.z;
754 
755         v = [v; moveJI(posPata, params, Tactual, T, Tactual + T)];
756         i = i + 1;
757         Tactual = Tactual + T;
758     end
759 end
760 
761 function [p3, p4] = giroI(p0, ang, origen)
762     posOrigen0.x = origen.x + p0.x;
763     posOrigen0.y = origen.y - p0.y;
764 
765     r = sqrt(posOrigen0.x^2 + posOrigen0.y^2);
766     alfa0 = atan2(posOrigen0.x, posOrigen0.y);
767 
768     alfa3 = alfa0 - ang / 2;
769     posOrigen3.x = r * sin(alfa3);
770     posOrigen3.y = r * cos(alfa3);
771     p3.x = posOrigen3.x - origen.x;
772     p3.y = origen.y - posOrigen3.y;
```

```
773     p3.z = p0.z;
774
775     alfa4 = alfa0 + ang / 2;
776     posOrigen4.x = r * sin(alfa4);
777     posOrigen4.y = r * cos(alfa4);
778     p4.x = posOrigen4.x - origen.x;
779     p4.y = origen.y - posOrigen4.y;
780     p4.z = p0.z;
781 end
782
783 % Debido al montaje en la simulacion:
784 % . q1 toma sentido opuesto al que debe -> q1 * (-1)
785 % . q2 se toma el complementario -> pi/2 - q2
786 % . q3 coincide
787
788 function [q1, q2, q3] = inverseKinI(p, params)
789 % desde la base de cada brazo
790     p.x = real(p.x);
791     p.y = real(p.y);
792     p.z = real(p.z);
793     if p.x < 0.0000005 && p.x > -0.0000005
794         p.x = 0.00000001;
795     end
796     if p.y < 0.0000005 && p.y > -0.0000005
797         p.y = 0.00000001;
798     end
799     if p.z < 0.0000005 && p.z > -0.0000005
800         p.z = 0.00000001;
801     end
802     q1 = atan2(p.x, -p.y); % - porque es el lado izq del robot
803     r = sqrt(p.x^2 + p.y^2);
804     s = sqrt((r - params.l1)^2 + (p.z - params.h1)^2);
805     alfa = atan2(p.z - params.h1, r - params.l1);
806     beta = acos((s^2 + params.l2^2 - params.l3^2) / (2 * s * params.l2));
807     ganma = acos((params.l2^2 + params.l3^2 - s^2) / (2 * params.l2 * params.l3));
808     q2 = alfa - beta;
809     q3 = ganma - pi / 2;
810 end
811
812 function p = directKinI(q, params)
813 % desde la base de cada brazo
814     p.x = 0; p.y = 0;
815     s = sqrt(params.l2^2 + params.l3^2 - 2 * params.l2 * params.l3 * cos(q(3) + pi
    / 2));
816     beta = acos((s^2 + params.l2^2 - params.l3^2) / (2 * s * params.l2));
817     alfa = q(2) + beta;
818     p.z = params.h1 + s * sin(alfa);
819     p.y = -(params.l1 + s * cos(alfa)) * cos(q(1));
820     p.x = -p.y * tan(q(1));
821 end
822
823 %% Funciones lado derecho
824 function v = moveJD(pObj, params, Tini, T, Tfin)
825     [q1, q2, q3] = inverseKinD(pObj, params);
826     v = moveMotor(q1, Tini, T, Tfin);
827     v = [v, moveMotor(q2, Tini, T, Tfin)];
828     v = [v, moveMotor(q3, Tini, T, Tfin)];
829 end
830
831 function v = moveLD(q0, pObj, params, Tini, T, Tfin)
```

```
832     p0 = directKinD(q0, params);
833     distEuc = sqrt((pObj.x - p0.x)^2 + (pObj.y - p0.y)^2 + (pObj.z - p0.z)^2);
834     if distEuc < 0.0000005 && distEuc > -0.0000005
835         vUnitario.x = 0;
836         vUnitario.y = 0;
837         vUnitario.z = 0;
838     else
839         vUnitario.x = (pObj.x - p0.x) / distEuc;
840         vUnitario.y = (pObj.y - p0.y) / distEuc;
841         vUnitario.z = (pObj.z - p0.z) / distEuc;
842     end
843     a = -2 * distEuc / (Tfin - Tini)^3; % Trayectoria cubica
844     b = 3 * distEuc / (Tfin - Tini)^2;
845 
846     iteracionFinal = (Tfin - Tini) / T;
847     Tactual = Tini;
848     v = [];
849     i = 1;
850     while i <= iteracionFinal
851         t = i * T;
852         s = a * t^3 + b * t^2;
853         p.x = s * vUnitario.x + p0.x;
854         p.y = s * vUnitario.y + p0.y;
855         p.z = s * vUnitario.z + p0.z;
856         v = [v; moveJD(p, params, Tactual, T, Tactual + T)];
857         i = i + 1;
858         Tactual = Tactual + T;
859     end
860 end
861 
862 function v = moveCD(q0, pObj, h, params, Tini, T, Tfin)
863     p0 = directKinD(q0, params);
864     dist = sqrt((pObj.x - p0.x)^2 + (pObj.y - p0.y)^2);
865     vUnitario.x = (pObj.x - p0.x) / dist;
866     vUnitario.y = (pObj.y - p0.y) / dist;
867 
868     a = -2 * dist / (Tfin - Tini)^3; % Trayectoria cubica
869     b = 3 * dist / (Tfin - Tini)^2;
870 
871     A = - 4 * h;
872     B = - A;
873 
874     iteracionFinal = (Tfin - Tini) / T;
875     Tactual = Tini;
876     v = [];
877     i = 1;
878     while i <= iteracionFinal
879         t = i * T;
880         s = a * t^3 + b * t^2;
881 
882         p.x = s * vUnitario.x + p0.x;
883         p.y = s * vUnitario.y + p0.y;
884         p.z = p0.z + (A * (s / dist)^2 + B * s / dist);
885 
886         v = [v; moveJD(p, params, Tactual, T, Tactual + T)];
887         i = i + 1;
888         Tactual = Tactual + T;
889     end
890 end
891 
892 function v = moveGD(q0, ang, origen, params, Tini, T, Tfin)
```

```
893     posPata0 = directKinD(q0, params);
894     posOrigen0.x = origen.x + posPata0.x;
895     posOrigen0.y = origen.y + posPata0.y;
896 
897     r = sqrt(posOrigen0.x^2 + posOrigen0.y^2);
898     alfa0 = atan2(posOrigen0.x, -posOrigen0.y);
899 
900     a = -2 * ang / (Tfin - Tini)^3; % Trayectoria cubica
901     b = 3 * ang / (Tfin - Tini)^2;
902 
903     iteracionFinal = (Tfin - Tini) / T;
904     Tactual = Tini;
905     v = [];
906     i = 1;
907     while i <= iteracionFinal
908         t = i * T;
909         s = a * t^3 + b * t^2;
910 
911         alfa = alfa0 + s;
912         posOrigen.x = r * sin(alfa);
913         posOrigen.y = -r * cos(alfa);
914 
915         posPata.x = posOrigen.x - origen.x;
916         posPata.y = posOrigen.y - origen.y;
917         posPata.z = posPata0.z;
918 
919         v = [v; moveJD(posPata, params, Tactual, T, Tactual + T)];
920         i = i + 1;
921         Tactual = Tactual + T;
922     end
923 end
924 
925 function [p3, p4] = giroD(p0, ang, origen)
926     posOrigen0.x = origen.x + p0.x;
927     posOrigen0.y = origen.y + p0.y;
928 
929     r = sqrt(posOrigen0.x^2 + posOrigen0.y^2);
930     alfa0 = atan2(posOrigen0.x, -posOrigen0.y);
931 
932     alfa3 = alfa0 - ang / 2;
933     posOrigen3.x = r * sin(alfa3);
934     posOrigen3.y = -r * cos(alfa3);
935     p3.x = posOrigen3.x - origen.x;
936     p3.y = posOrigen3.y - origen.y;
937     p3.z = p0.z;
938 
939     alfa4 = alfa0 + ang / 2;
940     posOrigen4.x = r * sin(alfa4);
941     posOrigen4.y = -r * cos(alfa4);
942     p4.x = posOrigen4.x - origen.x;
943     p4.y = posOrigen4.y - origen.y;
944     p4.z = p0.z;
945 end
946 
947 % Debido al montaje en la simulacion:
948 % . q1 toma sentido opuesto al que debe -> q1 * (-1)
949 % . q2 se toma el complementario -> pi/2 - q2
950 % . q3 coincide
951 
952 function [q1, q2, q3] = inverseKinD(p, params)
953 % desde la base de cada brazo
```

```
954     p.x = real(p.x);
955     p.y = real(p.y);
956     p.z = real(p.z);
957     if p.x < 0.0000005 && p.x > -0.0000005
958         p.x = 0.00000001;
959     end
960     if p.y < 0.0000005 && p.y > -0.0000005
961         p.y = 0.00000001;
962     end
963     if p.z < 0.0000005 && p.z > -0.0000005
964         p.z = 0.00000001;
965     end
966     q1 = atan2(p.x, -p.y);
967     r = sqrt(p.x^2 + p.y^2);
968     s = sqrt((r - params.l1)^2 + (p.z + params.h1)^2);
969     alfa = atan2(abs(p.z + params.h1), r - params.l1);
970     beta = acos((s^2 + params.l2^2 - params.l3^2) / (2 * s * params.l2));
971     ganma = acos((params.l2^2 + params.l3^2 - s^2) / (2 * params.l2 * params.l3));
972     q2 = alfa - beta;
973     q3 = ganma - pi / 2;
974 end
975
976 function p = directKinD(q, params)
977 % desde la base de cada brazo
978     p.x = 0; p.y = 0;
979     s = sqrt(params.l2^2 + params.l3^2 - 2 * params.l2 * params.l3 * cos(q(3) + pi
    / 2));
980     beta = acos((s^2 + params.l2^2 - params.l3^2) / (2 * s * params.l2));
981     alfa = q(2) + beta;
982     p.z = -params.h1 - s * sin(alfa);
983     p.y = -(params.l1 + s * cos(alfa)) * cos(q(1));
984     p.x = -p.y * tan(q(1));
985 end
986
987 %% Puntos avance
988 function [p1, p2] = puntosAvance(p0, avance, vectorDireccion)
989     p1 = [struct('x', p0(1).x + avance * vectorDireccion.x, 'y', p0(1).y - avance *
     vectorDireccion.y, 'z', p0(1).z);
990           struct('x', p0(2).x - avance * vectorDireccion.x, 'y', p0(2).y - avance *
     vectorDireccion.y, 'z', p0(2).z);
991           struct('x', p0(3).x - avance * vectorDireccion.x, 'y', p0(3).y + avance *
     vectorDireccion.y, 'z', p0(3).z);
992           struct('x', p0(4).x + avance * vectorDireccion.x, 'y', p0(4).y + avance *
     vectorDireccion.y, 'z', p0(4).z);
993           struct('x', p0(5).x + avance * vectorDireccion.x, 'y', p0(5).y - avance *
     vectorDireccion.y, 'z', p0(5).z);
994           struct('x', p0(6).x - avance * vectorDireccion.x, 'y', p0(6).y - avance *
     vectorDireccion.y, 'z', p0(6).z)];
995
996     p2 = [struct('x', p0(1).x - avance * vectorDireccion.x, 'y', p0(1).y + avance *
     vectorDireccion.y, 'z', p0(1).z);
997           struct('x', p0(2).x + avance * vectorDireccion.x, 'y', p0(2).y + avance *
     vectorDireccion.y, 'z', p0(2).z);
998           struct('x', p0(3).x + avance * vectorDireccion.x, 'y', p0(3).y - avance *
     vectorDireccion.y, 'z', p0(3).z);
999           struct('x', p0(4).x - avance * vectorDireccion.x, 'y', p0(4).y - avance *
     vectorDireccion.y, 'z', p0(4).z);
1000          struct('x', p0(5).x - avance * vectorDireccion.x, 'y', p0(5).y + avance *
     vectorDireccion.y, 'z', p0(5).z);
1001          struct('x', p0(6).x + avance * vectorDireccion.x, 'y', p0(6).y + avance *
     vectorDireccion.y, 'z', p0(6).z)];
```

```
1002 end
```

D

Código del robot

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>

#define SERVICE_UUID        "beb5483e-36e1-4688-b7f5-ea07361b26a8"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

BLEServer *pServer;
BLEService *pService;
BLECharacteristic *pCharacteristic;

#include <math.h>
#include <Servo.h>
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();

#define I2C_SDA_PIN 21
#define I2C_SCL_PIN 22

#define FREQUENCY 50
#define MIN_PWM 140
#define MAX_PWM 560
#define PERIODO 50

#define QUARTER_PI PI / 4
#define THREE_QUARTER_PI 3 * PI / 4
#define DEG_TO_RAD 0.017453292519943295769236907684886
#define RAD_TO_DEG 57.295779513082320876798154814105

#define NumPatas 6
#define NumServosPata 3

#define h 30
#define avance 50         //60
#define giro PI / 6

```

```
38 double alturaBase = 150;
39
40 void BLE() {
41   BLEDevice::deinit(false);
42   BLEDevice::init("ESP32-BLE-Hexapod");
43   pServer = BLEDevice::createServer();
44   pService = pServer->createService(SERVICE_UUID);
45   pCharacteristic = pService->createCharacteristic(
46                               CHARACTERISTIC_UUID,
47                               BLECharacteristic::PROPERTY_READ |
48                               BLECharacteristic::PROPERTY_WRITE
49                               );
50   pCharacteristic->setValue("-");
51   pService->start();
52   //BLEAdvertising *pAdvertising = pServer->getAdvertising();
53   BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
54   pAdvertising->addServiceUUID(SERVICE_UUID);
55   pAdvertising->setScanResponse(true);
56   //pAdvertising->setMinPreferred(0x06);
57   pAdvertising->setMinPreferred(0x12);
58   BLEDevice::startAdvertising();
59 }
60
61 typedef struct {            // Estructura para los parametros de las
       patas
62   double h1;
63   double l1;
64   double l2;
65   double l3;
66 } RobotParams;
67
68 const RobotParams params = {17.8, 55, 70, 166.66};
69
70 typedef struct {            // Estructura para marcar el origen de
      las patas desde el sistema de referencia de la base
71   double x;
72   double y;
73 } OrigenPata;
74
75 OrigenPata origen[] = {{ 77.8,  54.5},
76                        { 77.8, -54.5},
77                        {    0,    78},
78                        {    0,   -78},
79                        {-77.8,  54.5},
80                        {-77.8, -54.5}};
81
82 typedef struct {            // Estructura para representar un punto
      del espacio
83   double x;
```

```
84    double y;
85    double z;
86  } Punto;
87
88  typedef struct {            // Estructura para la configuracion de
       cada servo de la placa
89    int offset;
90    int signo;
91    int min_pos;
92    int max_pos;
93  } ServoTipo1;
94
95  typedef struct {            // Estructura para la configuracion de
       cada servo de la controladora
96    uint8_t pin;
97    int offset;
98    int signo;
99    int min_pos;
100   int max_pos;
101 } ServoTipo2;
102
103 ServoTipo1 pata1[NumServosPata], pata4[NumServosPata], pata5[
       NumServosPata];
104
105 ServoTipo2 pata2[] = {{0,  90 - 20,  1,  -75, 90},
106                       {1,  90 - 15, -1,  -90, 35},
107                       {2, 135 - 25, -1,   15, 90}};
108
109 ServoTipo2 pata3[] = {{4, 90 - 15, -1,  -75, 90},
110                       {5, 90 - 15,  1,  -90, 35},
111                       {6, 90 - 20,  1,  -28, 28}};
112
113 ServoTipo2 pata6[] = {{ 8, 90 - 20,  1,  -75,  90},
114                       { 9, 90 -  4, -1,  -90,  35},
115                       {10, 45 - 25, -1,  -90, -15}};
116
117 Servo servosPata1[NumServosPata], servosPata4[NumServosPata],
       servosPata5[NumServosPata];
118
119 double q0[][NumServosPata] = {{0, 0,  45},
120                               {0, 0,  45},
121                               {0, 0,   0},
122                               {0, 0,   0},
123                               {0, 0, -45},
124                               {0, 0, -45}};
125
126 Punto p0[] = {{ 88.3883, -88.3883,  alturaBase},
127               { 88.3883, -88.3883, -alturaBase},
128               {       0,     -125,  alturaBase},
```

```
129                {       0,     -125, -alturaBase},
130                {-88.3883, -88.3883,  alturaBase},
131                {-88.3883, -88.3883, -alturaBase}};
132
133 Punto p1[] = {{p0[0].x + avance, -88.3883,  alturaBase},
134                {p0[1].x - avance, -88.3883, -alturaBase},
135                {p0[2].x - avance,     -125,  alturaBase},
136                {p0[3].x + avance,     -125, -alturaBase},
137                {p0[4].x + avance, -88.3883,  alturaBase},
138                {p0[5].x - avance, -88.3883, -alturaBase}};
139
140 Punto p2[] = {{p0[0].x - avance, -88.3883,  alturaBase},
141                {p0[1].x + avance, -88.3883, -alturaBase},
142                {p0[2].x + avance,     -125,  alturaBase},
143                {p0[3].x - avance,     -125, -alturaBase},
144                {p0[4].x - avance, -88.3883,  alturaBase},
145                {p0[5].x + avance, -88.3883, -alturaBase}};
146
147 float direccion = 0;
148
149 Punto pGiro1[] = {{ 119.7078,  -40.5068,  alturaBase},
150                   { 119.7078,  -40.5068, -alturaBase},
151                   { -52.5403, -118.0829,  alturaBase},
152                   { -52.5403, -118.0829, -alturaBase},
153                   { -45.7434, -126.5321,  alturaBase},
154                   { -45.7434, -126.5321, -alturaBase}};
155
156 Punto pGiro2[] = {{   45.7434, -126.5321,  alturaBase},
157                   {   45.7434, -126.5321, -alturaBase},
158                   {   52.5403, -118.0829,  alturaBase},
159                   {   52.5403, -118.0829, -alturaBase},
160                   { -119.7078,  -40.5068,  alturaBase},
161                   { -119.7078,  -40.5068, -alturaBase}};
162
163 Punto pAparcado[] = {{0.00, -172.85,  65.65},
164                      {0.00, -172.85, -65.65},
165                      {0.00, -172.85,  65.65},
166                      {0.00, -172.85, -60.65},
167                      {0.00, -172.85,  70.65},
168                      {0.00, -172.85, -65.65}};
169
170 Punto pLevantarse[] = {{0.00, -172.85,  65.65 + 90},
171                         {0.00, -172.85, -65.65 - 90},
172                         {0.00, -172.85,  65.65 + 90},
173                         {0.00, -172.85, -60.65 - 90},
174                         {0.00, -172.85,  70.65 + 90},
175                         {0.00, -172.85, -65.65 - 90}};
176
177 char movimiento = 'I';
```

```
178 String accion, accionPrevia;
179 int paso = 0;
180
181 void goTo(Punto p[]);
182 void cambiarSentidoGiro();
183 void moveAbsJ(double qT[][NumServosPata], double T);
184 void moveJ(Punto pT[], double T);
185 void moveL(Punto pT[], double T);
186 void moveAvance(Punto pT[], double T);
187 void moveGiro(Punto pT[], double T, int sentido);
188 void writeServo2(ServoTipo2 motor, int angle);
189 void inverseKinI(Punto pT, double qT[]);
190 void directKinI(double q0[], Punto *pT);
191 void inverseKinD(Punto pT, double qT[]);
192 void directKinD(double q0[], Punto *pT);
193
194 void setup() {
195   pinMode(33, OUTPUT);
196   pinMode(32, OUTPUT);
197   pinMode(23, OUTPUT);
198   pinMode(27, OUTPUT);
199   pinMode(26, OUTPUT);
200   pinMode(25, OUTPUT);
201   pinMode(14, OUTPUT);
202   pinMode(13, OUTPUT);
203   pinMode(12, OUTPUT);
204
205   pwm.begin();
206   pwm.setPWMFreq(FREQUENCY);
207   yield();
208
209   Serial.begin(115200);
210   delay(1000);
211   Serial.println("Empezando");
212
213   BLE();
214
215   pata1[0].offset = 90 - 8;   pata1[0].signo = -1;
216   pata1[0].min_pos = -75;     pata1[0].max_pos = 90;
217   pata1[1].offset = 90 - 5;   pata1[1].signo =  1;
218   pata1[1].min_pos = -90;     pata1[1].max_pos = 35;
219   pata1[2].offset = 45 - 0;   pata1[2].signo =  1;
220   pata1[2].min_pos =   15;    pata1[2].max_pos = 90;
221
222   pata4[0].offset = 90 - 0;   pata4[0].signo =  1;
223   pata4[0].min_pos = -75;     pata4[0].max_pos = 90;
224   pata4[1].offset = 90 + 5;   pata4[1].signo = -1;
225   pata4[1].min_pos = -90;     pata4[1].max_pos = 35;
226   pata4[2].offset = 90 - 0;   pata4[2].signo = -1;
```

```
227  pata4[2].min_pos = -28;     pata4[2].max_pos = 28;
228
229  pata5[0].offset =  90 - 0;  pata5[0].signo = -1;
230  pata5[0].min_pos = -75;     pata5[0].max_pos = 90;
231  pata5[1].offset =  90 + 0;  pata5[1].signo =  1;
232  pata5[1].min_pos = -90;     pata5[1].max_pos = 35;
233  pata5[2].offset = 135 - 0;  pata5[2].signo =  1;
234  pata5[2].min_pos = -90;     pata5[2].max_pos =  -15;
235
236  servosPata1[0].attach(33);
237  servosPata1[1].attach(32);
238  servosPata1[2].attach(23);
239
240  servosPata4[0].attach(25);
241  servosPata4[1].attach(26);
242  servosPata4[2].attach(27);
243
244  servosPata5[0].attach(14);
245  servosPata5[1].attach(12);
246  servosPata5[2].attach(13);
247
248  delay(1000);
249  servosPata1[0].write(constrain( 0, pata1[0].min_pos, pata1[0].
      max_pos) * pata1[0].signo +  pata1[0].offset);
250  servosPata1[1].write(constrain( 0, pata1[1].min_pos, pata1[1].
      max_pos) * pata1[1].signo +  pata1[1].offset);
251  servosPata1[2].write(constrain(45, pata1[2].min_pos, pata1[2].
      max_pos) * pata1[2].signo +  pata1[2].offset);
252  delay(1000);
253  writeServo2(pata2[0],  0);
254  writeServo2(pata2[1],  0);
255  writeServo2(pata2[2],  45);
256  delay(1000);
257  writeServo2(pata3[0], 0);
258  writeServo2(pata3[1], 0);
259  writeServo2(pata3[2], 0);
260  delay(1000);
261  servosPata4[0].write(constrain(0, pata4[0].min_pos, pata4[0].
      max_pos) * pata4[0].signo +  pata4[0].offset);
262  servosPata4[1].write(constrain(0, pata4[1].min_pos, pata4[1].
      max_pos) * pata4[1].signo +  pata4[1].offset);
263  servosPata4[2].write(constrain(0, pata4[2].min_pos, pata4[2].
      max_pos) * pata4[2].signo +  pata4[2].offset);
264  delay(1000);
265  servosPata5[0].write(constrain(  0, pata5[0].min_pos, pata5[0].
      max_pos) * pata5[0].signo +  pata5[0].offset);
266  servosPata5[1].write(constrain(  0, pata5[1].min_pos, pata5[1].
      max_pos) * pata5[1].signo +  pata5[1].offset);
267  servosPata5[2].write(constrain(-45, pata5[2].min_pos, pata5[2].
```

```
    max_pos) * pata5[2].signo +  pata5[2].offset);
268   delay(1000);
269   writeServo2(pata6[0],   0);
270   writeServo2(pata6[1],   0);
271   writeServo2(pata6[2], -45);
272   delay(1000);
273
274   moveJ(p0, 2000);
275   delay(500);
276
277   Serial.println("Empezando loop");
278 }
279
280 void loop() {
281   std::string value = pCharacteristic->getValue();
282   accionPrevia = accion;
283   accion = value.c_str();
284   Serial.println(accion);
285
286   if (accion == "A") {              // Avanzar hacia delante
287     if (accionPrevia != accion) {
288       direccion = 0;
289       puntosAvance();
290       goTo(p1);
291     }
292     moveAvance(p2, 1500);      // 3500
293     delay(500);
294     moveAvance(p1, 1500);
295     delay(500);
296   } else if (accion == "B") {       // Avanzar hacia atras
297     if (accionPrevia != accion) {
298       direccion = PI;
299       puntosAvance();
300       goTo(p1);
301     }
302     moveAvance(p2, 1500);
303     delay(500);
304     moveAvance(p1, 1500);
305     delay(500);
306   } else if (accion == "C") {       // Avanzar hacia la derecha
307     if (accionPrevia != accion) {
308       direccion = -HALF_PI;
309       puntosAvance();
310       goTo(p1);
311     }
312     moveAvance(p2, 1500);
313     delay(500);
314     moveAvance(p1, 1500);
315     delay(500);
```

```
316   } else if (accion == "D") {       // Avanzar hacia la izquierda
317     if (accionPrevia != accion) {
318       direccion = HALF_PI;
319       puntosAvance();
320       goTo(p1);
321     }
322     moveAvance(p2, 1500);
323     delay(500);
324     moveAvance(p1, 1500);
325     delay(500);
326   } else if (accion == "E") {       // Avanzar hacia delante-derecha
327     if (accionPrevia != accion) {
328       direccion = -QUARTER_PI;
329       puntosAvance();
330       goTo(p1);
331     }
332     moveAvance(p2, 1500);
333     delay(500);
334     moveAvance(p1, 1500);
335     delay(500);
336   } else if (accion == "F") {       // Avanzar hacia atras-derecha
337     if (accionPrevia != accion) {
338       direccion = -THREE_QUARTER_PI;
339       puntosAvance();
340       goTo(p1);
341     }
342     moveAvance(p2, 1500);
343     delay(500);
344     moveAvance(p1, 1500);
345     delay(500);
346   } else if (accion == "G") {       // Avanzar hacia atras-izquierda
347     if (accionPrevia != accion) {
348       direccion = THREE_QUARTER_PI;
349       puntosAvance();
350       goTo(p1);
351     }
352     moveAvance(p2, 1500);
353     delay(500);
354     moveAvance(p1, 1500);
355     delay(500);
356   } else if (accion == "H") {       // Avanzar hacia delante-
     izquierda
357     if (accionPrevia != accion) {
358       direccion = QUARTER_PI;
359       puntosAvance();
360       goTo(p1);
361     }
362     moveAvance(p2, 1500);
363     delay(500);
```

```
364     moveAvance(p1, 1500);
365     delay(500);
366   } else if (accion == "I") {        // Detenerse
367     if (accionPrevia != accion) {
368       goTo(p0);
369     }
370   } else if (accion == "J") {        // Girar a la derecha
371     if (accionPrevia == "K") {
372       cambiarSentidoGiro();
373     } else if (accionPrevia != accion) {
374       goTo(pGiro1);
375     }
376     moveGiro(pGiro2, 2000, 1);
377     delay(500);
378     moveGiro(pGiro1, 2000, 1);
379     delay(500);
380   } else if (accion == "K") {        // Girar a la izquierda
381     if (accionPrevia == "J") {
382       cambiarSentidoGiro();
383     } else if (accionPrevia != accion) {
384       goTo(pGiro1);
385       movimiento = 'D';
386     }
387     moveGiro(pGiro2, 2000, -1);
388     delay(500);
389     moveGiro(pGiro1, 2000, -1);
390     delay(500);
391   } else if (accion == "L") {        // Aparcar
392     if (accionPrevia != accion) {
393       goTo(pLevantarse);
394       delay(500);
395       moveL(pAparcado, 3000);
396     }
397   } else if (accion == "M") {        // Levantarse
398     if (accionPrevia == "L") {
399       moveL(pLevantarse, 4000);
400     }
401   } else if (accion == "Z") {        // Reiniciar BLE
402     delay(1000);
403     BLE();
404     pCharacteristic->setValue("-");
405   }
406   delay(200);
407 }
408
409 void goTo(Punto p[]) {
410   movimiento = 'I';
411   Punto paux[NumPatas];
412   directKinI(q0[0], &paux[0]);
```

```
413   directKinD(q0[3], &paux[3]);
414   directKinI(q0[4], &paux[4]);
415   paux[1].x = p[1].x;
416   paux[1].y = p[1].y;
417   paux[1].z = p[1].z;
418   paux[2].x = p[2].x;
419   paux[2].y = p[2].y;
420   paux[2].z = p[2].z;
421   paux[5].x = p[5].x;
422   paux[5].y = p[5].y;
423   paux[5].z = p[5].z;
424   moveAvance(paux, 2000);
425   delay(500);
426   moveAvance(p, 2000);
427   delay(500);
428 }
429
430 void cambiarSentidoGiro(){
431   if (movimiento == 'I') { movimiento = 'D'; }
432   else { movimiento = 'I'; }
433 }
434
435 void moveAbsJ(double qT[][NumServosPata], double T) {  // Mueve el
      robot entre dos configuraciones de angulos
436   double a[NumPatas][NumServosPata], b[NumPatas][NumServosPata];  //
        c no se incluye al ser nulas todas sus componentes
437   double d[][NumServosPata] = {{q0[0][0], q0[0][1], q0[0][2]},
438                                {q0[1][0], q0[1][1], q0[1][2]},
439                                {q0[2][0], q0[2][1], q0[2][2]},
440                                {q0[3][0], q0[3][1], q0[3][2]},
441                                {q0[4][0], q0[4][1], q0[4][2]},
442                                {q0[5][0], q0[5][1], q0[5][2]}};
443
444   double T2 = T * T, T3 = T2 * T, q[NumPatas][NumServosPata];
445   int i, j;
446
447   for (i = 0; i < NumPatas; i++) {  // Se calculan los parametros a
       y b para cada articulacion
448     for (j = 0; j < NumServosPata; j++) {
449       a[i][j] = -2 * (qT[i][j] - q0[i][j]) / T3;
450       b[i][j] = 3 * (qT[i][j] - q0[i][j]) / T2;
451     }
452   }
453
454   int itFinal = T / PERIODO;  // Se define la cantidad de
       iteraciones necesarias, sabiendo que cada una durara PERIODO ms
455   double t1, t2, t3;  // Se declaran aqui las variables para no
       tener que repetir su declaracion
456   unsigned long t;
```

```
457
458   for (int itActual = 0; itActual < itFinal; itActual++) {  // El
       bucle calcula las siguientes posiciones de los servos para seguir
       la trayectoria cubica
459     t = millis();
460     t1 = itActual * PERIODO;
461     t2 = t1 * t1;
462     t3 = t2 * t1;
463
464     for (i = 0; i < NumPatas; i++) {
465       for (j = 0; j < NumServosPata; j++) {
466         q[i][j] = a[i][j] * t3 + b[i][j] * t2 + d[i][j];
467       }
468     }
469     for (i = 0; i < NumServosPata; i++) {  // Se mueven los motores
470       servosPata1[i].write(constrain(round(q[0][i]), pata1[i].
       min_pos, pata1[i].max_pos) * pata1[i].signo +  pata1[i].offset);
471       servosPata4[i].write(constrain(round(q[3][i]), pata4[i].
       min_pos, pata4[i].max_pos) * pata4[i].signo +  pata4[i].offset);
472       servosPata5[i].write(constrain(round(q[4][i]), pata5[i].
       min_pos, pata5[i].max_pos) * pata5[i].signo +  pata5[i].offset);
473     }
474     delay(1);
475     for (i = 0; i < NumServosPata; i++) {
476       writeServo2(pata2[i], round(q[1][i]));
477       writeServo2(pata3[i], round(q[2][i]));
478       writeServo2(pata6[i], round(q[5][i]));
479     }
480     delay(PERIODO - (millis() - t));  // Se espera el tiempo
       necesario para alcanzar los PERIODO ms de cada iteracion
481   }
482 }
483
484 void moveJ(Punto pT[], double T) {  // Mueve el robot a partir de
       las posiciones actuales de los servos y un punto final
485   double qT[NumPatas][NumServosPata];  // Se crea un vector donde se
       almacenaran los valores finales de las articulaciones para
       llevar a cabo el movimiento
486   // Llamada a la funcion que realiza dicho calculo
487   inverseKinI(pT[0], qT[0]);
488   inverseKinD(pT[1], qT[1]);
489   inverseKinI(pT[2], qT[2]);
490   inverseKinD(pT[3], qT[3]);
491   inverseKinI(pT[4], qT[4]);
492   inverseKinD(pT[5], qT[5]);
493   moveAbsJ(qT, T);  // Mueve el robot a la configuracion deseada
494   for (int i = 0; i < NumPatas; i++) {
495     for (int j = 0; j < NumServosPata; j++) {
496       q0[i][j] = qT[i][j];
```

```
497     }
498   }
499 }
500
501 void moveL(Punto pT[], double T) {   // Mueve los extremos de las
      patas linealmente a partir de las posiciones actuales de los
      servos y los punto finales
502   double a[NumPatas], b[NumPatas], distEuc;
503   double T2 = T * T, T3 = T2 * T;
504   Punto p0[NumPatas], pUnitario[NumPatas], pSiguiente[NumPatas];  //
       Se define una serie de posiciones para el desarrollo de la
      funcion
505   // Se calcula el punto actual a partir de los angulos actuales
506   directKinI(q0[0], &p0[0]);
507   directKinD(q0[1], &p0[1]);
508   directKinI(q0[2], &p0[2]);
509   directKinD(q0[3], &p0[3]);
510   directKinI(q0[4], &p0[4]);
511   directKinD(q0[5], &p0[5]);
512   int i;
513   for (i = 0; i < NumPatas; i++) {
514     distEuc = sqrt((pT[i].x - p0[i].x) * (pT[i].x - p0[i].x) + (pT[i
      ].y - p0[i].y) * (pT[i].y - p0[i].y) + (pT[i].z - p0[i].z) * (pT[
      i].z - p0[i].z));  // Se calcula una sola vez la distancia
      euclidea
515     if (distEuc < 0.000001) { distEuc = 0.0000001; }
516     a[i] = -2 * distEuc / T3;  // Se obtienen los valores de a y b
      para la trayectoria cubica
517     b[i] = 3 * distEuc / T2;
518     // Se calcula el vector unitario de la direccion
519     pUnitario[i].x = (pT[i].x - p0[i].x) / distEuc;
520     pUnitario[i].y = (pT[i].y - p0[i].y) / distEuc;
521     pUnitario[i].z = (pT[i].z - p0[i].z) / distEuc;
522   }
523   int itFinal = T / PERIODO;  // Se declaran una serie de variables
      para el movimiento
524   double s, t1, t2, t3;
525
526   for(int itActual = 1; itActual <= itFinal; itActual++) {  // Se
      comienza por 1 para inciar realizando movimiento y alcanzar
      completamente la posicion final
527     t1 = itActual * PERIODO;  // Se obtienen el tiempo actual y su
      cuadrado
528     t2 = t1 * t1;
529     t3 = t2 * t1;
530     for (i = 0; i < NumPatas; i++) {
531       s = a[i] * t3 + b[i] * t2;
532       // Se genera el punto que se alcanzara en el siguiente
      movimiento
```

```
533       pSiguiente[i].x = s * pUnitario[i].x + p0[i].x;
534       pSiguiente[i].y = s * pUnitario[i].y + p0[i].y;
535       pSiguiente[i].z = s * pUnitario[i].z + p0[i].z;
536     }
537     moveJ(pSiguiente, PERIODO);  // Se lleva al robot a dicho punto
538   }
539 }
540
541 void moveAvance(Punto pT[], double T) {   // Mueve el robot
     generando un paso en la direccion establecida
542   double a[NumPatas], b[NumPatas], distEuc[NumPatas]
543   double T2 = T * T, T3 = T2 * T;
544   Punto p0[NumPatas], pUnitario[NumPatas], pSiguiente[NumPatas];  //
       Se define una serie de posiciones para el desarrollo de la
     funcion
545   directKinI(q0[0], &p0[0]);
546   directKinD(q0[1], &p0[1]);
547   directKinI(q0[2], &p0[2]);
548   directKinD(q0[3], &p0[3]);
549   directKinI(q0[4], &p0[4]);
550   directKinD(q0[5], &p0[5]);
551   int i, n1, n2, n3;
552   // Se analiza que patas deben realizar cada movimiento
553   if (movimiento == 'I') {
554     n1 = 0;
555     n2 = 3;
556     n3 = 4;
557     movimiento = 'D';
558   } else {
559     n1 = 1;
560     n2 = 2;
561     n3 = 5;
562     movimiento = 'I';
563   }
564
565   for (i = 0; i < NumPatas; i++) {
566     if (i == n1 || i == n2 || i == n3) {
567       distEuc[i] = sqrt((pT[i].x - p0[i].x) * (pT[i].x - p0[i].x) +
      (pT[i].y - p0[i].y) * (pT[i].y - p0[i].y) + (pT[i].z - p0[i].z) *
       (pT[i].z - p0[i].z));  // Se calcula una sola vez la distancia
     euclidea
568       if (distEuc[i] < 0.000001) { distEuc[i] = 0.0000001; }
569       a[i] = -2 * distEuc[i] / T3;  // Se obtienen los valores de a
     y b para la trayectoria cubica
570       b[i] = 3 * distEuc[i] / T2;
571       // Se calcula el vector unitario de la direccion
572       pUnitario[i].x = (pT[i].x - p0[i].x) / distEuc[i];
573       pUnitario[i].y = (pT[i].y - p0[i].y) / distEuc[i];
574       pUnitario[i].z = (pT[i].z - p0[i].z) / distEuc[i];
```

```
575     } else {
576       distEuc[i] = sqrt((pT[i].x - p0[i].x) * (pT[i].x - p0[i].x) +
    (pT[i].y - p0[i].y) * (pT[i].y - p0[i].y));    // Se calcula una
    sola vez la distancia euclidea
577       if (distEuc[i] < 0.000001) { distEuc[i] = 0.0000001; }
578       a[i] = -2 * distEuc[i] / T3;
          // Se obtienen los valores de a y b para la trayectoria cubica
579       b[i] = 3 * distEuc[i] / T2;
580       // Se calcula el vector unitario de la direccion sobre el
    plano XY
581       pUnitario[i].x = (pT[i].x - p0[i].x) / distEuc[i];
582       pUnitario[i].y = (pT[i].y - p0[i].y) / distEuc[i];
583     }
584   }
585   int itFinal = T / PERIODO;
586   double s, t1, t2, t3;
587   double A = - 4 * h, B = -A;
588   for(int itActual = 1; itActual <= itFinal; itActual++) {
589     t1 = itActual * PERIODO;
590     t2 = t1 * t1;
591     t3 = t2 * t1;
592     for (i = 0; i < NumPatas; i++) {
593       s = a[i] * t3 + b[i] * t2;
594       // Se genera el punto que se alcanzara en el siguiente
    movimiento segun la trayectoria que debe realizar cada pata
595       pSiguiente[i].x = s * pUnitario[i].x + p0[i].x;
596       pSiguiente[i].y = s * pUnitario[i].y + p0[i].y;
597       if (i == n1 || i == n2 || i == n3) {
598         pSiguiente[i].z = s * pUnitario[i].z + p0[i].z;
599       } else {
600         s = s / distEuc[i];
601         pSiguiente[i].z = pT[i].z + ((i % 2 == 0) ? -1 : 1) * (A * s
    * s + B * s);
602       }
603     }
604     moveJ(pSiguiente, PERIODO);  // Se lleva al robot a dicho punto
605   }
606 }
607
608 void moveGiro(Punto pT[], double T, int sentido) {  // Mueve el
    robot realizando un paso de giro en el sentido establecido
609   double a[NumPatas], b[NumPatas], distEuc[NumPatas], ang0[NumPatas
    ];
610   double T2 = T * T, T3 = T2 * T;
611   Punto p0[NumPatas], pUnitario[NumPatas], pSiguiente[NumPatas],
    posOrigen[NumPatas];  // Se definen una serie de posiciones para
    el desarrollo de la funcion
612   directKinI(q0[0], &p0[0]);
613   directKinD(q0[1], &p0[1]);
```

```
614   directKinI(q0[2], &p0[2]);
615   directKinD(q0[3], &p0[3]);
616   directKinI(q0[4], &p0[4]);
617   directKinD(q0[5], &p0[5]);
618   int i, n1, n2, n3;
619   // Se analiza que patas deben realizar cada movimiento
620   if (movimiento == 'I') {
621     n1 = 0;
622     n2 = 3;
623     n3 = 4;
624     movimiento = 'D';
625   } else {
626     n1 = 1;
627     n2 = 2;
628     n3 = 5;
629     movimiento = 'I';
630   }
631   for (i = 0; i < NumPatas; i++) {
632     if (i == n1 || i == n2 || i == n3) {
633       // Se obtienen las posiciones respecto al origen del robot
634       posOrigen[i].x = origen[i].x + p0[i].x;
635       posOrigen[i].y = origen[i].y + p0[i].y * (i % 2 == 0 ? -1:1);
636       // Se obtiene la longitud en el plano XY que separa el punto
      del origen
637       distEuc[i] = sqrt(posOrigen[i].x * posOrigen[i].x + posOrigen[
      i].y * posOrigen[i].y);
638       ang0[i] = atan2(posOrigen[i].x, -posOrigen[i].y);
639
640       a[i] = -2 * (PI / 6) / T3;
641       b[i] = 3 * (PI / 6) / T2;
642     } else {
643       distEuc[i] = sqrt((pT[i].x - p0[i].x) * (pT[i].x - p0[i].x) +
      (pT[i].y - p0[i].y) * (pT[i].y - p0[i].y));  // Se calcula una
      sola vez la distancia euclidea
644       if (distEuc[i] < 0.000001) { distEuc[i] = 0.0000001; }
645       a[i] = -2 * distEuc[i] / T3;
646       b[i] = 3 * distEuc[i] / T2;
647     // Se calcula el vector unitario de la direccion en el plano XY
648       pUnitario[i].x = (pT[i].x - p0[i].x) / distEuc[i];
649       pUnitario[i].y = (pT[i].y - p0[i].y) / distEuc[i];
650     }
651   }
652   int itFinal = T / PERIODO;
653   double s, t1, t2, t3;
654   double A = - 4 * h, B = -A;
655   for(int itActual = 1; itActual <= itFinal; itActual++) {
656     t1 = itActual * PERIODO;
657     t2 = t1 * t1;
658     t3 = t2 * t1;
```

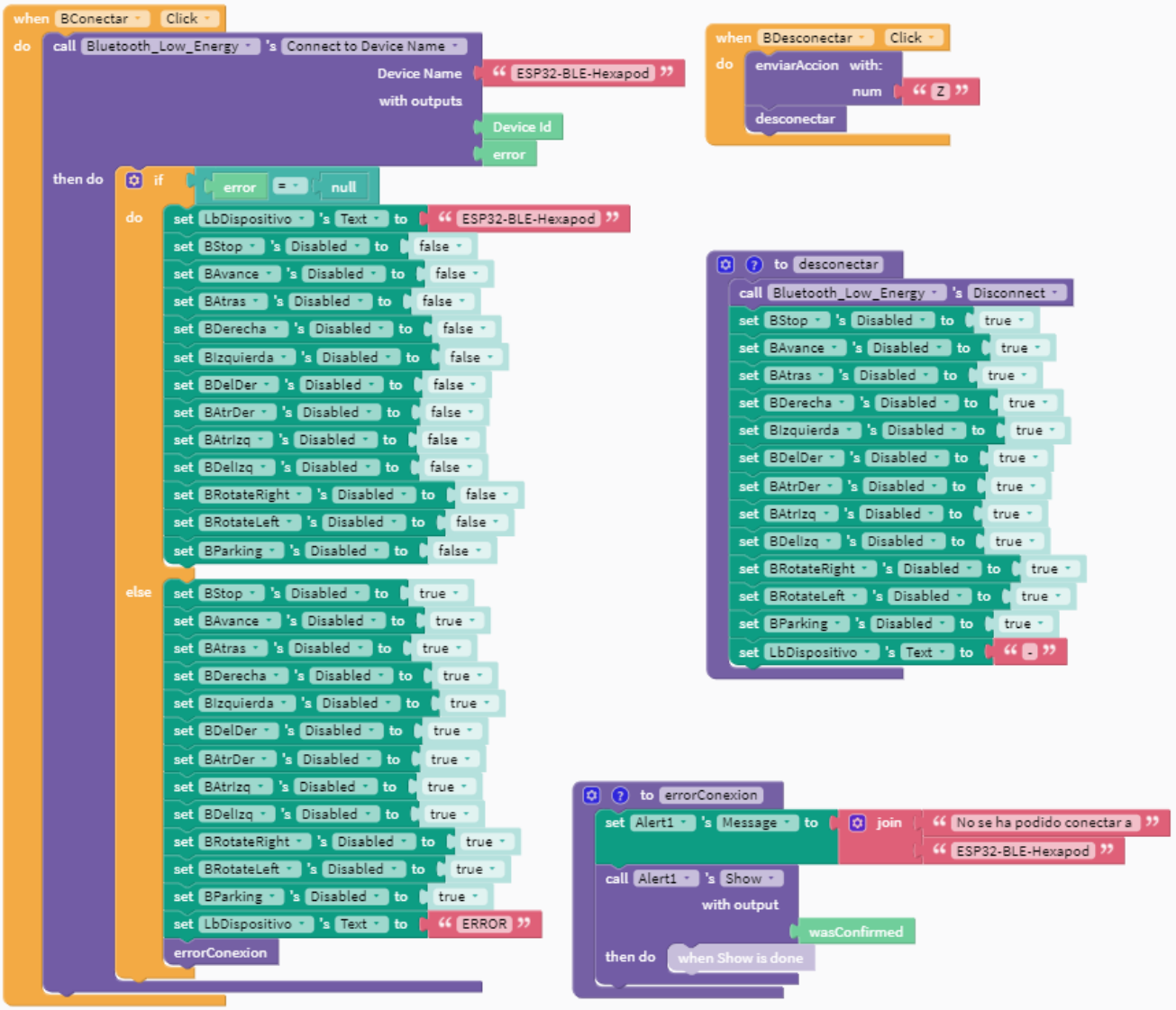
```
659     for (i = 0; i < NumPatas; i++) {
660       s = a[i] * t3 + b[i] * t2;
661       // Se genera el punto que se alcanzara en el siguiente
    movimiento segun la trayectoria que debe realizar cada pata
662       if (i == n1 || i == n2 || i == n3) {
663         s = ang0[i] + s * sentido;
664         posOrigen[i].x = distEuc[i] * sin(s);
665         posOrigen[i].y = -distEuc[i] * cos(s);
666         pSiguiente[i].x = posOrigen[i].x - origen[i].x;
667         pSiguiente[i].y = ((i % 2 == 0) ? -posOrigen[i].y + origen[i
    ].y : posOrigen[i].y - origen[i].y);
668         pSiguiente[i].z = pT[i].z;
669       } else {
670         pSiguiente[i].x = s * pUnitario[i].x + p0[i].x;
671         pSiguiente[i].y = s * pUnitario[i].y + p0[i].y;
672         s = s / distEuc[i];
673         pSiguiente[i].z = pT[i].z + ((i % 2 == 0) ? -1 : 1) * (A * s
     * s + B * s);
674       }
675     }
676     moveJ(pSiguiente, PERIODO);  // Se lleva al robot a dicho punto
677   }
678 }
679
680 void writeServo2(ServoTipo2 motor, int angle) {   // Mueve el
    servomotor al angulo recibido como parametro
681   int pulse_width;
682   angle = constrain(angle, motor.min_pos, motor.max_pos);
683   pulse_width = map(angle * motor.signo + motor.offset, 0, 180,
     MIN_PWM, MAX_PWM);
684   pwm.setPWM(motor.pin, 0, pulse_width);
685 }
686
687 void inverseKinI(Punto pT, double qT[]) {
688   // Para evitar posibles problemas en los calculos, se evitan las
     coordenadas de 0 en todos los ejes
689   if(pT.x < 0.005 && pT.x > -0.005){ pT.x = 0.0001; }
690   if(pT.y < 0.005 && pT.y > -0.005){ pT.y = 0.0001; }
691   if(pT.z < 0.005 && pT.z > -0.005){ pT.z = 0.0001; }
692   double r, s, alfa, beta, ganma;
693   qT[2] = atan2(pT.x, -pT.y) * RAD_TO_DEG;
694   r = sqrt(pT.x * pT.x + pT.y * pT.y);
695   s = sqrt((r - params.l1) * (r - params.l1) + (pT.z - params.h1) *
     (pT.z - params.h1));
696   alfa = atan2(pT.z - params.h1, r - params.l1);
697   beta = acos((s * s + params.l2 * params.l2 - params.l3 * params.l3
     ) / (2 * s * params.l2));
698   ganma = acos((params.l2 * params.l2 + params.l3 * params.l3 - s *
     s) / (2 * params.l2 * params.l3));
```

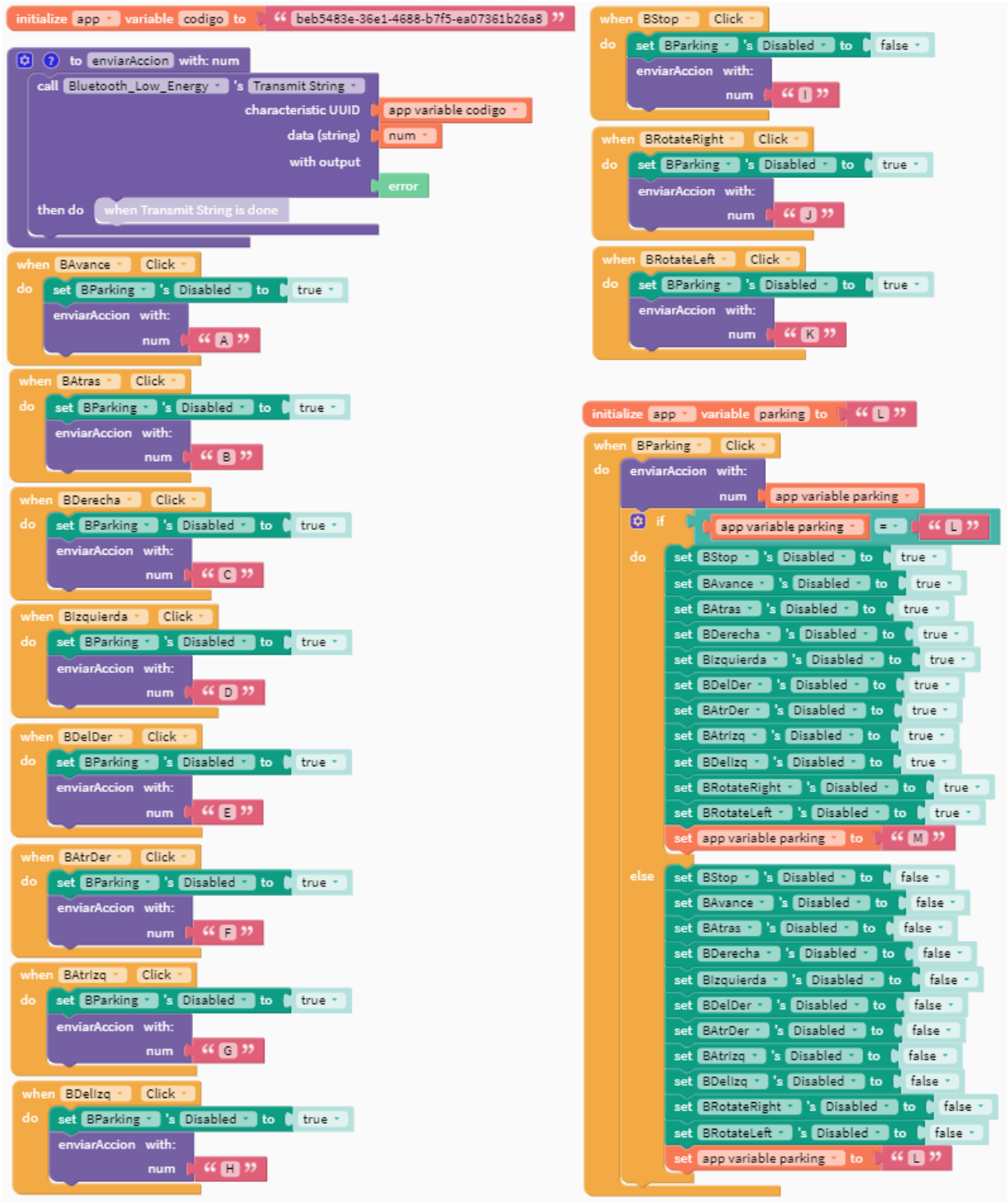
```
699   qT[1] = (alfa - beta) * RAD_TO_DEG;
700   qT[0] = (ganma - HALF_PI) * RAD_TO_DEG;
701 }
702
703 void directKinI(double q0[], Punto *pT) {
704   double s, beta, alfa;
705   q0[0] *= DEG_TO_RAD;  // Se convierten los angulos a radianes para
        trabajar con ellos
706   q0[1] *= DEG_TO_RAD;
707   q0[2] *= DEG_TO_RAD;
708   s = sqrt(params.l2 * params.l2 + params.l3 * params.l3 - 2 *
       params.l2 * params.l3 * cos(q0[0] + HALF_PI));
709   beta = acos((s * s + params.l2 * params.l2 - params.l3 * params.l3
       ) / (2 * s * params.l2));
710   alfa = q0[1] + beta;
711   pT->z = params.h1 + s * sin(alfa);
712   pT->y = -(params.l1 + s * cos(alfa)) * cos(q0[2]);
713   pT->x = -pT->y * tan(q0[2]);
714   q0[0] *= RAD_TO_DEG;  // Se vuelven a convertir los angulos a
       grados, ya que las funciones implementadas los esperan con estas
       unidades
715   q0[1] *= RAD_TO_DEG;
716   q0[2] *= RAD_TO_DEG;
717 }
718
719 void inverseKinD(Punto pT, double qT[]) {
720   // Para evitar posibles problemas en los calculos, se evitan las
       coordenadas de 0 en todos los ejes
721   if(pT.x < 0.005 && pT.x > -0.005){ pT.x = 0.0001; }
722   if(pT.y < 0.005 && pT.y > -0.005){ pT.y = 0.0001; }
723   if(pT.z < 0.005 && pT.z > -0.005){ pT.z = 0.0001; }
724   double r, s, alfa, beta, ganma;
725   qT[2] = atan2(pT.x, -pT.y) * RAD_TO_DEG;
726   r = sqrt(pT.x * pT.x + pT.y * pT.y);
727   s = sqrt((r - params.l1) * (r - params.l1) + (pT.z + params.h1) *
       (pT.z + params.h1));
728   alfa = atan2(abs(pT.z + params.h1), r - params.l1);
729   beta = acos((s * s + params.l2 * params.l2 - params.l3 * params.l3
       ) / (2 * s * params.l2));
730   ganma = acos((params.l2 * params.l2 + params.l3 * params.l3 - s *
       s) / (2 * params.l2 * params.l3));
731   qT[1] = (alfa - beta) * RAD_TO_DEG;
732   qT[0] = (ganma - HALF_PI) * RAD_TO_DEG;
733 }
734
735 void directKinD(double q0[], Punto *pT) {
736   double s, beta, alfa;
737   q0[0] *= DEG_TO_RAD;  // Se convierten los angulos a radianes para
        trabajar con ellos
```

```
738   q0[1] *= DEG_TO_RAD;
739   q0[2] *= DEG_TO_RAD;
740   s = sqrt(params.l2 * params.l2 + params.l3 * params.l3 - 2 *
       params.l2 * params.l3 * cos(q0[0] + HALF_PI));
741   beta = acos((s * s + params.l2 * params.l2 - params.l3 * params.l3
       ) / (2 * s * params.l2));
742   alfa = q0[1] + beta;
743   pT->z = -params.h1 - s * sin(alfa);
744   pT->y = -(params.l1 + s * cos(alfa)) * cos(q0[2]);
745   pT->x = -pT->y * tan(q0[2]);
746   q0[0] *= RAD_TO_DEG;  // Se vuelven a convertir los angulos a
       grados, ya que las funciones implementadas los esperan con estas
       unidades
747   q0[1] *= RAD_TO_DEG;
748   q0[2] *= RAD_TO_DEG;
749 }
750
751 void puntosAvance() {
752   double dirx = cos(direccion), diry = sin(direccion);
753   int i;
754   p1[0].x = p0[0].x + avance * dirx; p1[0].y = p0[0].y - avance *
       diry;  p1[0].z =  alturaBase;
755   p1[1].x = p0[1].x - avance * dirx; p1[1].y = p0[1].y - avance *
       diry;  p1[1].z = -alturaBase;
756   p1[2].x = p0[2].x - avance * dirx; p1[2].y = p0[2].y + avance *
       diry;  p1[2].z =  alturaBase;
757   p1[3].x = p0[3].x + avance * dirx; p1[3].y = p0[3].y + avance *
       diry;  p1[3].z = -alturaBase;
758   p1[4].x = p0[4].x + avance * dirx; p1[4].y = p0[4].y - avance *
       diry;  p1[4].z =  alturaBase;
759   p1[5].x = p0[5].x - avance * dirx; p1[5].y = p0[5].y - avance *
       diry;  p1[5].z = -alturaBase;
760
761   p2[0].x = p0[0].x - avance * dirx; p2[0].y = p0[0].y + avance *
       diry;  p2[0].z =  alturaBase;
762   p2[1].x = p0[1].x + avance * dirx; p2[1].y = p0[1].y + avance *
       diry;  p2[1].z = -alturaBase;
763   p2[2].x = p0[2].x + avance * dirx; p2[2].y = p0[2].y - avance *
       diry;  p2[2].z =  alturaBase;
764   p2[3].x = p0[3].x - avance * dirx; p2[3].y = p0[3].y - avance *
       diry;  p2[3].z = -alturaBase;
765   p2[4].x = p0[4].x - avance * dirx; p2[4].y = p0[4].y + avance *
       diry;  p2[4].z =  alturaBase;
766   p2[5].x = p0[5].x + avance * dirx; p2[5].y = p0[5].y + avance *
       diry;  p2[5].z = -alturaBase;
767 }
```

E

Código de la aplicación

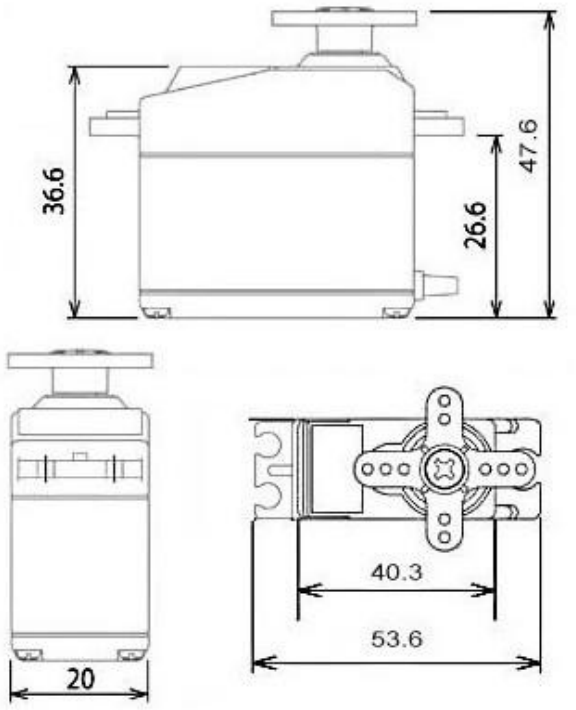



initialize app variable codigo to beb5483e-36e1-4688-b7f5-ea07361b26a8
to enviarAccion with: num
call Bluetooth_Low_Energy 's Transmit String
characteristic UUID app variable codigo
data (string) num
with output
error
then do when Transmit String is done
when BAvance Click
do set BParking 's Disabled to true
enviarAccion with: num A
when BAtras Click
do set BParking 's Disabled to true
enviarAccion with: num B
when BDerecha Click
do set BParking 's Disabled to true
enviarAccion with: num C
when BIzquierda Click
do set BParking 's Disabled to true
enviarAccion with: num D
when BDelDer Click
do set BParking 's Disabled to true
enviarAccion with: num E
when BAtrDer Click
do set BParking 's Disabled to true
enviarAccion with: num F
when BAtrIzq Click
do set BParking 's Disabled to true
enviarAccion with: num G
when BDelIzq Click
do set BParking 's Disabled to true
enviarAccion with: num H
when BStop Click
do set BParking 's Disabled to false
enviarAccion with: num I
when BRotateRight Click
do set BParking 's Disabled to true
enviarAccion with: num J
when BRotateLeft Click
do set BParking 's Disabled to true
enviarAccion with: num K
initialize app variable parking to L
when BParking Click
do enviarAccion with: num app variable parking
if app variable parking = L
do set BStop 's Disabled to true
set BAvance 's Disabled to true
set BAtras 's Disabled to true
set BDerecha 's Disabled to true
set BIzquierda 's Disabled to true
set BDelDer 's Disabled to true
set BAtrDer 's Disabled to true
set BAtrIzq 's Disabled to true
set BDelIzq 's Disabled to true
set BRotateRight 's Disabled to true
set BRotateLeft 's Disabled to true
set app variable parking to M
else set BStop 's Disabled to false
set BAvance 's Disabled to false
set BAtras 's Disabled to false
set BDerecha 's Disabled to false
set BIzquierda 's Disabled to false
set BDelDer 's Disabled to false
set BAtrDer 's Disabled to false
set BAtrIzq 's Disabled to false
set BDelIzq 's Disabled to false
set BRotateRight 's Disabled to false
set BRotateLeft 's Disabled to false
set app variable parking to L

F

Hojas de características de los componentes empleados

MG996R High Torque
Metal Gear Dual Ball Bearing Servo



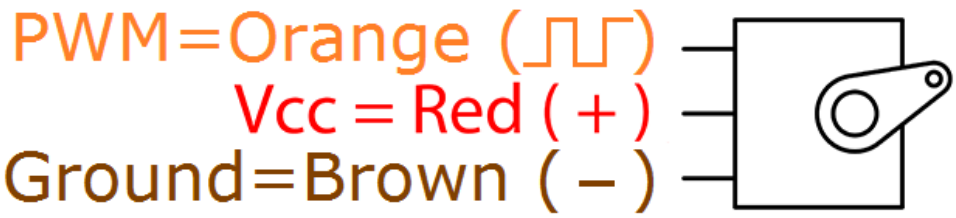
This High-Torque MG996R Digital Servo features metal gearing resulting in extra high 10kg stalling torque in a tiny package. The MG996R is essentially an upgraded version of the famous MG995 servo, and features upgraded shock-proofing and a redesigned PCB and IC control system that make it much more accurate than its predecessor. The gearing and motor have also been upgraded to improve dead bandwith and centering. The unit comes complete with 30cm wire and 3 pin 'S' type female header connector that fits most receivers, including Futaba, JR, GWS, Cirrus, Blue Bird, Blue Arrow, Corona, Berg, Spektrum and Hitec.

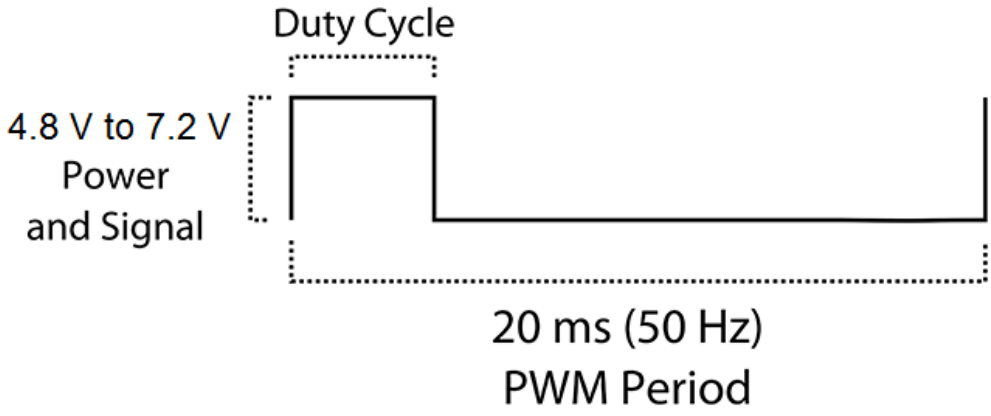
This high-torque standard servo can rotate approximately 120 degrees (60 in each direction). You can use any servo code, hardware or library to control these servos, so it's great for beginners who want to make stuff move without building a motor controller with feedback & gear box, especially since it will fit in small places. The MG996R Metal Gear Servo also comes with a selection of arms and hardware to get you set up nice and fast!

Specifications

- Weight: 55 g
- Dimension: 40.7 x 19.7 x 42.9 mm approx.
- Stall torque: 9.4 kgf·cm (4.8 V), 11 kgf·cm (6 V)
- Operating speed: 0.17 s/60º (4.8 V), 0.14 s/60º (6 V)

- Operating voltage: 4.8 V a 7.2 V
- Running Current 500 mA – 900 mA (6V)
- Stall Current 2.5 A (6V)
- Dead band width: 5 µs
- Stable and shock proof double ball bearing design
- Temperature range: 0 ºC – 55 ºC





PCA9685

16-channel, 12-bit PWM Fm+ I²C-bus LED controller

Rev. 4 — 16 April 2015 Product data sheet

1. General description

The PCA9685 is an I²C-bus controlled 16-channel LED controller optimized for Red/Green/Blue/Amber (RGBA) color backlighting applications. Each LED output has its own 12-bit resolution (4096 steps) fixed frequency individual PWM controller that operates at a programmable frequency from a typical of 24 Hz to 1526 Hz with a duty cycle that is adjustable from 0 % to 100 % to allow the LED to be set to a specific brightness value. All outputs are set to the same PWM frequency.

Each LED output can be off or on (no PWM control), or set at its individual PWM controller value. The LED output driver is programmed to be either open-drain with a 25 mA current sink capability at 5 V or totem pole with a 25 mA sink, 10 mA source capability at 5 V. The PCA9685 operates with a supply voltage range of 2.3 V to 5.5 V and the inputs and outputs are 5.5 V tolerant. LEDs can be directly connected to the LED output (up to 25 mA, 5.5 V) or controlled with external drivers and a minimum amount of discrete components for larger current or higher voltage LEDs.

The PCA9685 is in the new Fast-mode Plus (Fm+) family. Fm+ devices offer higher frequency (up to 1 MHz) and more densely populated bus operation (up to 4000 pF).

Although the PCA9635 and PCA9685 have many similar features, the PCA9685 has some unique features that make it more suitable for applications such as LCD or LED backlighting and Ambilight:

- The PCA9685 allows staggered LED output on and off times to minimize current surges. The on and off time delay is independently programmable for each of the 16 channels. This feature is not available in PCA9635.
- The PCA9685 has 4096 steps (12-bit PWM) of individual LED brightness control. The PCA9635 has only 256 steps (8-bit PWM).
- When multiple LED controllers are incorporated in a system, the PWM pulse widths between multiple devices may differ if PCA9635s are used. The PCA9685 has a programmable prescaler to adjust the PWM pulse widths of multiple devices.
- The PCA9685 has an external clock input pin that will accept user-supplied clock (50 MHz max.) in place of the internal 25 MHz oscillator. This feature allows synchronization of multiple devices. The PCA9635 does not have external clock input feature.
- Like the PCA9635, PCA9685 also has a built-in oscillator for the PWM control. However, the frequency used for PWM control in the PCA9685 is adjustable from about 24 Hz to 1526 Hz as compared to the typical 97.6 kHz frequency of the PCA9635. This allows the use of PCA9685 with external power supply controllers. All bits are set at the same frequency.
- The Power-On Reset (POR) default state of LEDn output pins is LOW in the case of PCA9685. It is HIGH for PCA9635.

The active LOW Output Enable input pin ($\overline{OE}$) allows asynchronous control of the LED outputs and can be used to set all the outputs to a defined I²C-bus programmable logic state. The $\overline{OE}$ can also be used to externally 'pulse width modulate' the outputs, which is useful when multiple devices need to be dimmed or blinked together using software control.

Software programmable LED All Call and three Sub Call I²C-bus addresses allow all or defined groups of PCA9685 devices to respond to a common I²C-bus address, allowing for example, all red LEDs to be turned on or off at the same time or marquee chasing effect, thus minimizing I²C-bus commands. Six hardware address pins allow up to 62 devices on the same bus.

The Software Reset (SWRST) General Call allows the master to perform a reset of the PCA9685 through the I²C-bus, identical to the Power-On Reset (POR) that initializes the registers to their default state causing the outputs to be set LOW. This allows an easy and quick way to reconfigure all device registers to the same condition via software.

2. Features and benefits

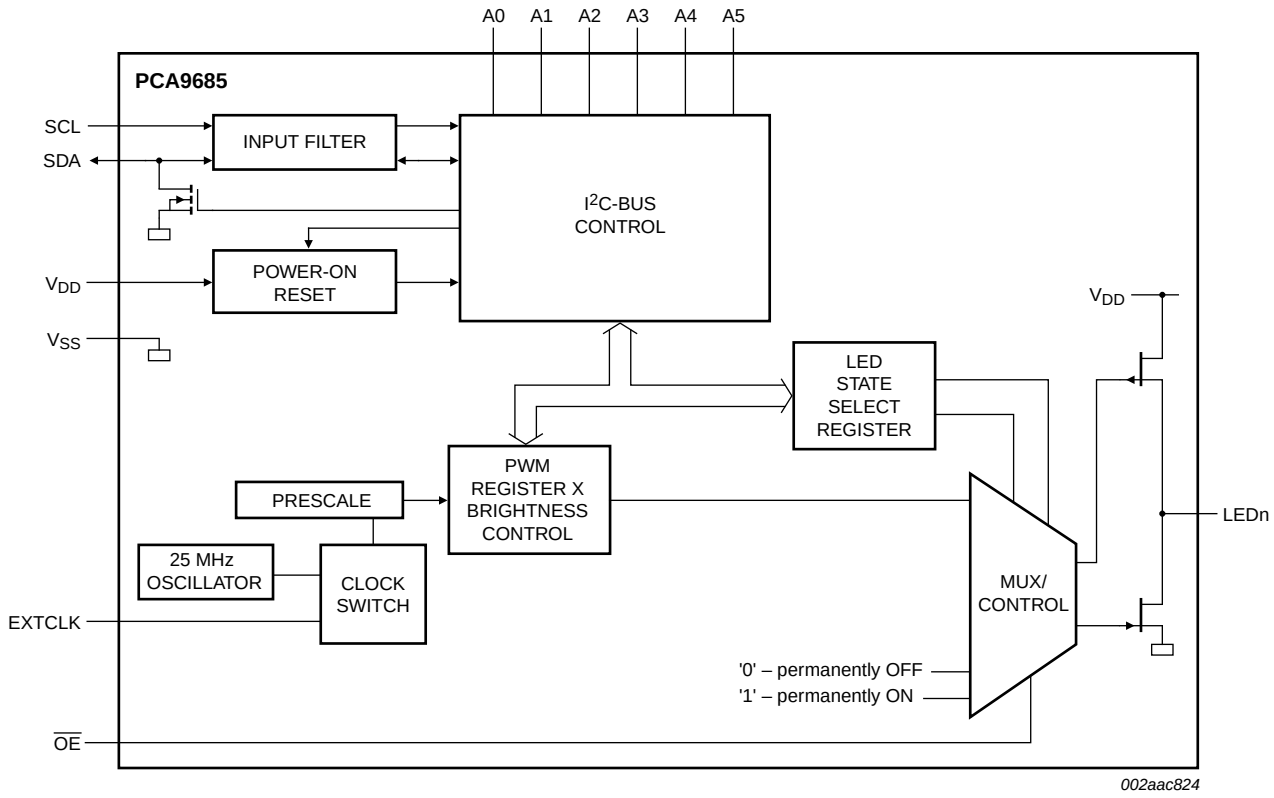
- 16 LED drivers. Each output programmable at:
  - Off
  - On
  - Programmable LED brightness
  - Programmable LED turn-on time to help reduce EMI
- 1 MHz Fast-mode Plus compatible I²C-bus interface with 30 mA high drive capability on SDA output for driving high capacitive buses
- 4096-step (12-bit) linear programmable brightness per LED output varying from fully off (default) to maximum brightness
- LED output frequency (all LEDs) typically varies from 24 Hz to 1526 Hz (Default of 1Eh in PRE_SCALE register results in a 200 Hz refresh rate with oscillator clock of 25 MHz.)
- Sixteen totem pole outputs (sink 25 mA and source 10 mA at 5 V) with software programmable open-drain LED outputs selection (default at totem pole). No input function.
- Output state change programmable on the Acknowledge or the STOP Command to update outputs byte-by-byte or all at the same time (default to 'Change on STOP').
- Active LOW Output Enable ($\overline{OE}$) input pin. LEDn outputs programmable to logic 1, logic 0 (default at power-up) or 'high-impedance' when $\overline{OE}$ is HIGH.
- 6 hardware address pins allow 62 PCA9685 devices to be connected to the same I²C-bus
- Toggling $\overline{OE}$ allows for hardware LED blinking
- 4 software programmable I²C-bus addresses (one LED All Call address and three LED Sub Call addresses) allow groups of devices to be addressed at the same time in any combination (for example, one register used for 'All Call' so that all the PCA9685s on the I²C-bus can be addressed at the same time and the second register used for three different addresses so that $1/3$ of all devices on the bus can be addressed at the same time in a group). Software enable and disable for these I²C-bus address.
- Software Reset feature (SWRST General Call) allows the device to be reset through the I²C-bus

- 25 MHz typical internal oscillator requires no external components
- External 50 MHz (max.) clock input
- Internal power-on reset
- Noise filter on SDA/SCL inputs
- Edge rate control on outputs
- No output glitches on power-up
- Supports hot insertion
- Low standby current
- Operating power supply voltage range of 2.3 V to 5.5 V
- 5.5 V tolerant inputs
- −40 °C to +85 °C operation
- ESD protection exceeds 2000 V HBM per JESD22-A114, 200 V MM per JESD22-A115 and 1000 V CDM per JESD22-C101
- Latch-up testing is done to JEDEC Standard JESD78 which exceeds 100 mA
- Packages offered: TSSOP28, HVQFN28

3. Applications

- RGB or RGBA LED drivers
- LED status information
- LED displays
- LCD backlights
- Keypad backlights for cellular phones or handheld devices

5. Block diagram



Remark: Only one LED output shown for clarity.

Fig 1. Block diagram of PCA9685

6. Pinning information

6.1 Pinning

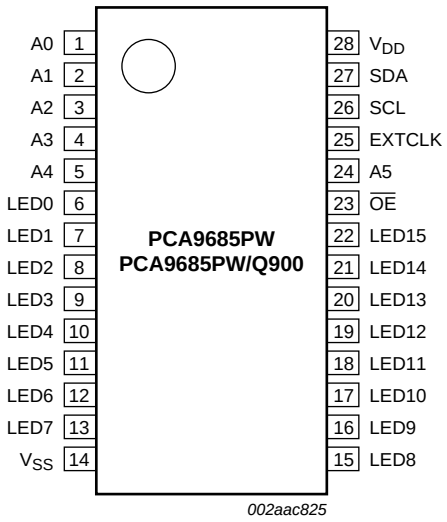


Fig 2. Pin configuration for TSSOP28

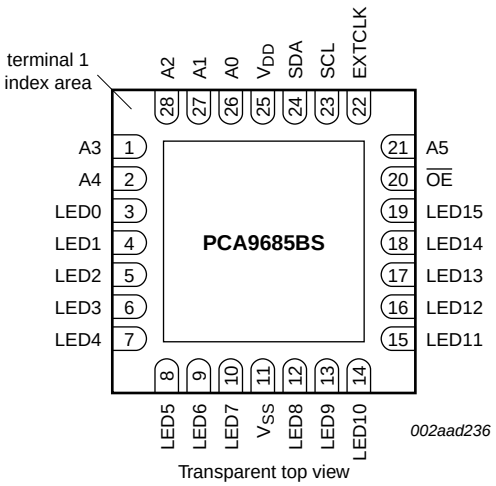


Fig 3. Pin configuration for HVQFN28

6.2 Pin description

Table 3. Pin description

Symbol	Pin		Type	Description
	TSSOP28	HVQFN28		
A0	1	26	I	address input 0
A1	2	27	I	address input 1
A2	3	28	I	address input 2
A3	4	1	I	address input 3
A4	5	2	I	address input 4
LED0	6	3	O	LED driver 0
LED1	7	4	O	LED driver 1
LED2	8	5	O	LED driver 2
LED3	9	6	O	LED driver 3
LED4	10	7	O	LED driver 4
LED5	11	8	O	LED driver 5
LED6	12	9	O	LED driver 6
LED7	13	10	O	LED driver 7
V_{SS}	14	11[1]	power supply	supply ground
LED8	15	12	O	LED driver 8
LED9	16	13	O	LED driver 9
LED10	17	14	O	LED driver 10
LED11	18	15	O	LED driver 11

Table 3. Pin description *...continued*

Symbol	Pin		Type	Description
	TSSOP28	HVQFN28		
LED12	19	16	O	LED driver 12
LED13	20	17	O	LED driver 13
LED14	21	18	O	LED driver 14
LED15	22	19	O	LED driver 15
$\overline{OE}$	23	20	I	active LOW output enable
A5	24	21	I	address input 5
EXTCLK	25	22	I	external clock input[2]
SCL	26	23	I	serial clock line
SDA	27	24	I/O	serial data line
V_{DD}	28	25	power supply	supply voltage

[1] HVQFN28 package die supply ground is connected to both V_{SS} pin and exposed center pad. V_{SS} pin must be connected to supply ground for proper device operation. For enhanced thermal, electrical, and board level performance, the exposed pad needs to be soldered to the board using a corresponding thermal pad on the board and for proper heat conduction through the board, thermal vias need to be incorporated in the PCB in the thermal pad region.

[2] This pin must be grounded when this feature is not used.

7. Functional description

Refer to Figure 1 “Block diagram of PCA9685”.

7.1 Device addresses

Following a START condition, the bus master must output the address of the slave it is accessing.

There are a maximum of 64 possible programmable addresses using the 6 hardware address pins. Two of these addresses, Software Reset and LED All Call, cannot be used because their default power-up state is ON, leaving a maximum of 62 addresses. Using other reserved addresses, as well as any other subcall address, will reduce the total number of possible addresses even further.

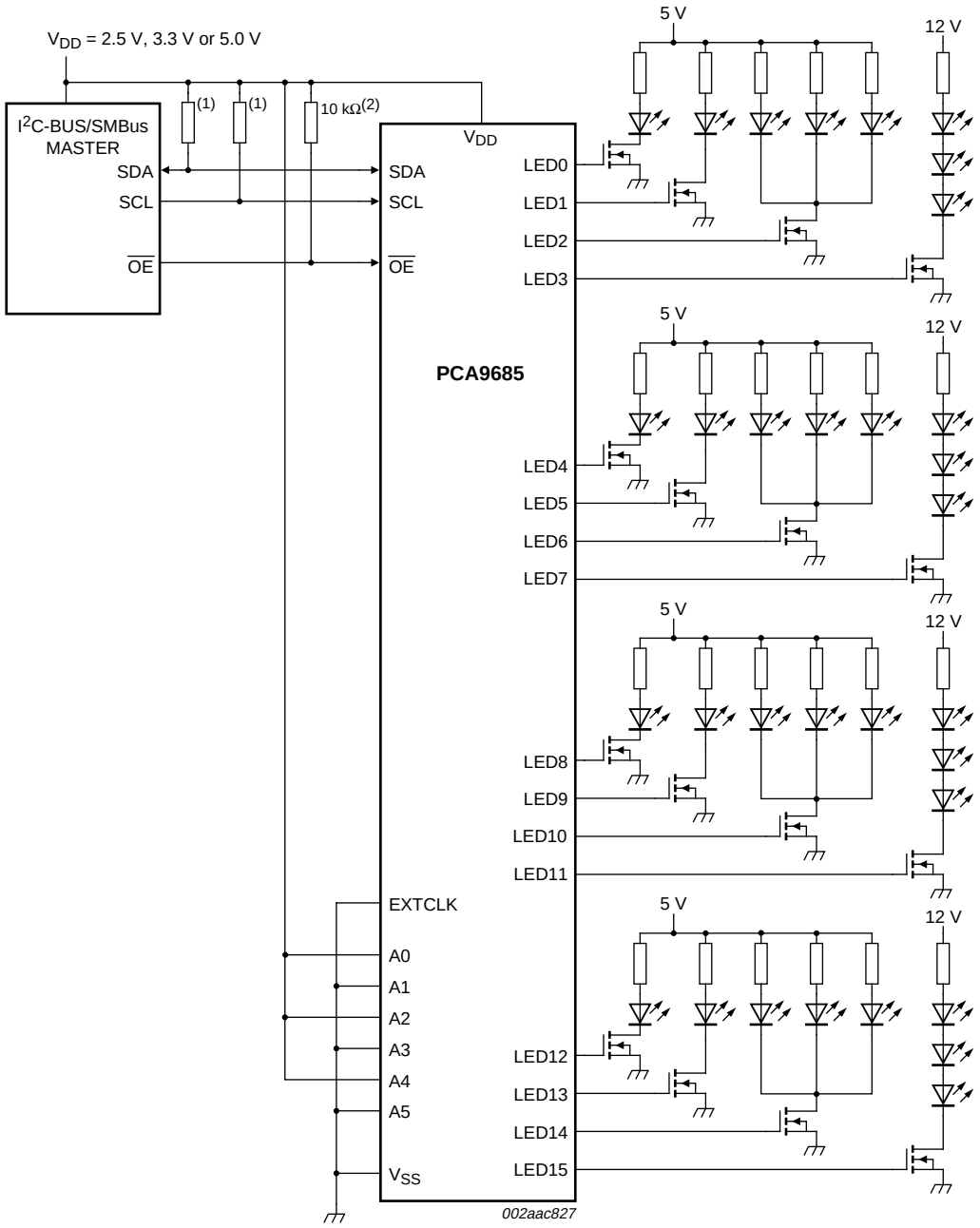
7.1.1 Regular I²C-bus slave address

The I²C-bus slave address of the PCA9685 is shown in Figure 4. To conserve power, no internal pull-up resistors are incorporated on the hardware selectable address pins and they must be pulled HIGH or LOW.

Remark: Using reserved I²C-bus addresses will interfere with other devices, but only if the devices are on the bus and/or the bus will be open to other I²C-bus systems at some later date. In a closed system where the designer controls the address assignment these addresses can be used since the PCA9685 treats them like any other address. The LED All Call, Software Reset and PCA9564 or PCA9665 slave address (if on the bus) can never be used for individual device addresses.

- PCA9685 LED All Call address (1110 000) and Software Reset (0000 0110) which are active on start-up

10. Application design-in information



I²C-bus address = 1010 101x.

All 16 of the LEDn outputs configurable as either open-drain or totem pole. Mixing of configuration is not possible.

Remark: Set INVRT = 0, OUTDRV = 1, OUTNE = 01 (MODE2 register bits)

(1) Resistor value should be chosen by referencing section 7 of *UM10204, "I²C-bus specification and user manual"*.

(2) $\overline{OE}$ requires pull-up resistor if control signal from the master is open-drain.

Fig 26. Typical application

11. Limiting values

Table 13. Limiting values

In accordance with the Absolute Maximum Rating System (IEC 60134).

Symbol	Parameter	Conditions	Min	Max	Unit
V_{DD}	supply voltage		–0.5	+6.0	V
$V_{I/O}$	voltage on an input/output pin		V_{SS} – 0.5	5.5	V
$I_{O(LEDn)}$	output current on pin LEDn		-	25	mA
I_{SS}	ground supply current		-	400	mA
P_{tot}	total power dissipation		-	400	mW
T_{stg}	storage temperature		–65	+150	°C
T_{amb}	ambient temperature	operating	–40	+85	°C

12. Static characteristics

Table 14. Static characteristics

V_{DD} = 2.3 V to 5.5 V; V_{SS} = 0 V; T_{amb} = –40 °C to +85 °C; unless otherwise specified.

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
Supply						
V_{DD}	supply voltage		2.3	-	5.5	V
I_{DD}	supply current	operating mode; no load; f_{SCL} = 1 MHz; V_{DD} = 2.3 V to 5.5 V	-	6	10	mA
I_{stb}	standby current	no load; f_{SCL} = 0 Hz; V_I = V_{DD} or V_{SS}; V_{DD} = 2.3 V to 5.5 V	-	2.2	15.5	µA
V_{POR}	power-on reset voltage	no load; V_I = V_{DD} or V_{SS} [1]	-	1.70	2.0	V
Input SCL; input/output SDA						
V_{IL}	LOW-level input voltage		–0.5	-	+0.3V_{DD}	V
V_{IH}	HIGH-level input voltage		0.7V_{DD}	-	5.5	V
I_{OL}	LOW-level output current	V_{OL} = 0.4 V; V_{DD} = 2.3 V	20	28	-	mA
		V_{OL} = 0.4 V; V_{DD} = 5.0 V	30	40	-	mA
I_L	leakage current	V_I = V_{DD} or V_{SS}	–1	-	+1	µA
C_i	input capacitance	V_I = V_{SS}	-	6	10	pF
LED driver outputs						
I_{OL}	LOW-level output current	V_{OL} = 0.5 V; V_{DD} = 2.3 V to 4.5 V [2]	12	25	-	mA
$I_{OL(tot)}$	total LOW-level output current	V_{OL} = 0.5 V; V_{DD} = 4.5 V [2]	-	-	400	mA
I_{OH}	HIGH-level output current	open-drain; V_{OH} = V_{DD}	–10	-	+10	µA
V_{OH}	HIGH-level output voltage	I_{OH} = –10 mA; V_{DD} = 2.3 V	1.6	-	-	V
		I_{OH} = –10 mA; V_{DD} = 3.0 V	2.3	-	-	V
		I_{OH} = –10 mA; V_{DD} = 4.5 V	4.0	-	-	V
I_{OZ}	OFF-state output current	3-state; V_{OH} = V_{DD} or V_{SS}	–10	-	+10	µA
C_o	output capacitance		-	5	8	pF

Table 14. Static characteristics *...continued*

V_{DD} = 2.3 V to 5.5 V; V_{SS} = 0 V; T_{amb} = −40 °C to +85 °C; unless otherwise specified.

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
Address inputs; $\overline{OE}$ input; EXTCLK						
V_{IL}	LOW-level input voltage		−0.5	-	+0.3V_{DD}	V
V_{IH}	HIGH-level input voltage		0.7V_{DD}	-	5.5	V
I_{LI}	input leakage current		−1	-	+1	µA
C_i	input capacitance		-	3	5	pF

[1] V_{DD} must be lowered to 0.2 V in order to reset part.

[2] Each bit must be limited to a maximum of 25 mA and the total package limited to 400 mA due to internal busing limits.

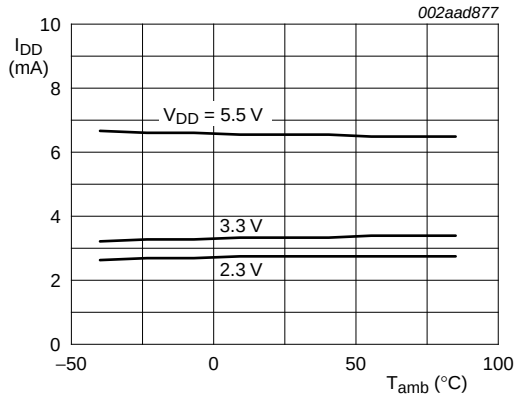


Fig 28. I_{DD} typical values with OSC on and f_{SCL} = 1 MHz versus temperature

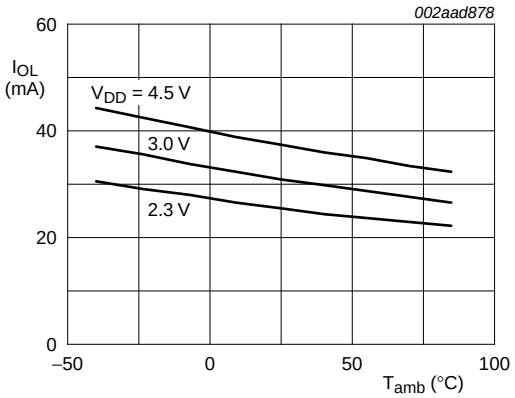


Fig 29. I_{OL} typical drive (LEDn outputs) versus temperature

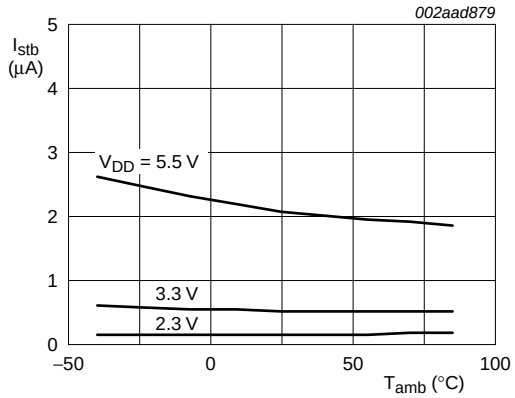


Fig 30. Standby supply current versus temperature

YEP 20A HV 2-12S SBEC with Selectable Voltage Output 5.0V / 5.5V / 6.0V / 7.0V / 9.0V

Known for their excellent quality, value and performance, this 20A HV SBEC by YEP is a must have for any model where clean, reliable power for your electronics is required. Offering a wide input voltage range of 6~50V, the YEP 20A SBEC can be used with just about any application from small models all the way up to 800 size helis or large gas airplanes.

The voltage output is selectable by jumper for 5V, 5.5V, 6V, 7V or 9V operation. Additionally, this unit features dual output leads which not only provides power redundancy, it also spreads the current load over two sets of leads to ensure you are not putting too many Amps through one lead.

Features:

- Switching BEC
- High voltage input (2~12S lipoly)
- Convenient jumper selectable voltage output of 5V/5.5V/6V/7V/9V (no programming required)
- High current handling of up to 20A
- Dual output leads offering power redundancy
- Integrated heatsink

Specification:

- Max Cont Current: 20A
- Input Voltage: 2-12 cells li-XX or 6-35 cells Ni-MH/Ni-CD battery
- Output Voltage: Jumper selectable 5V/5.5V/6V/7V/9V (no programming required)
- PCB Size: 57x26mm
- Weight: 42g

MP1584

3A, 1.5MHz, 28V Step-Down Converter

The Future of Analog IC Technology

DESCRIPTION

The MP1584 is a high frequency step-down switching regulator with an integrated internal high-side high voltage power MOSFET. It provides 3A output with current mode control for fast loop response and easy compensation.

The wide 4.5V to 28V input range accommodates a variety of step-down applications, including those in an automotive input environment. A 100μA operational quiescent current allows use in battery-powered applications.

High power conversion efficiency over a wide load range is achieved by scaling down the switching frequency at light load condition to reduce the switching and gate driving losses.

The frequency foldback helps prevent inductor current runaway during startup and thermal shutdown provides reliable, fault tolerant operation.

By switching at 1.5MHz, the MP1584 is able to prevent EMI (Electromagnetic Interference) noise problems, such as those found in AM radio and ADSL applications.

The MP1584 is available in a thermally enhanced SOIC8E package.

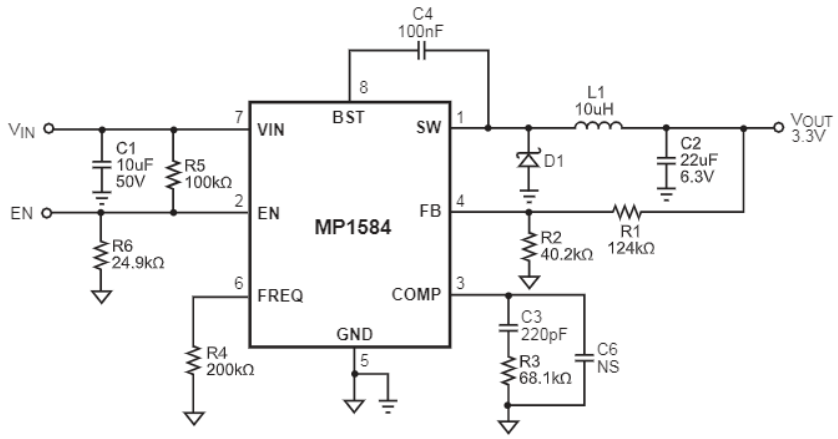
FEATURES

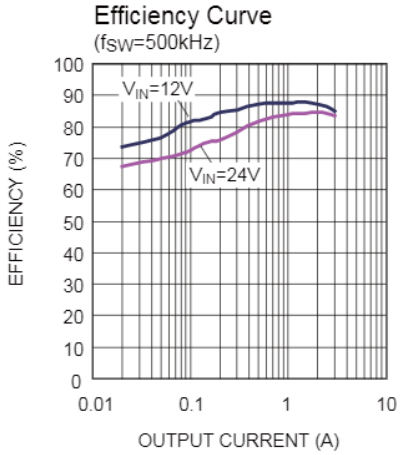
- Wide 4.5V to 28V Operating Input Range
- Programmable Switching Frequency from 100kHz to 1.5MHz
- High-Efficiency Pulse Skipping Mode for Light Load
- Ceramic Capacitor Stable
- Internal Soft-Start
- Internally Set Current Limit without a Current Sensing Resistor
- Available in SOIC8E Package.

APPLICATIONS

- High Voltage Power Conversion
- Automotive Systems
- Industrial Power Systems
- Distributed Power Systems
- Battery Powered Systems

TYPICAL APPLICATION



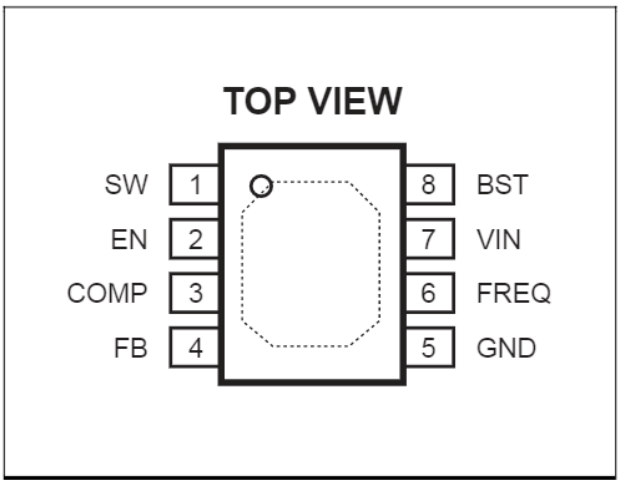


ORDERING INFORMATION

Part Number*	Package	Top Marking	Free Air Temperature (T_A)
MP1584EN	SOIC8E	MP1584EN	–20°C to +85°C

* For Tape & Reel, add suffix –Z (e.g. MP1584EN–Z);
For RoHS Compliant Packaging, add suffix –LF. (e.g. MP1584EN–LF–Z)

PACKAGE REFERENCE



ABSOLUTE MAXIMUM RATINGS (1)

Supply Voltage (V_{IN}) ..... –0.3V to +30V
Switch Voltage (V_{SW}) ..... –0.3V to V_{IN} + 0.3V
BST to SW ..... –0.3V to +6V
All Other Pins ..... –0.3V to +6V
Continuous Power Dissipation (T_A = +25°C)(2) 2.5W
Junction Temperature ..... 150°C
Lead Temperature ..... 260°C
Storage Temperature ..... –65°C to +150°C

Recommended Operating Conditions (3)

Supply Voltage V_{IN} ..... 4.5V to 28V
Output Voltage V_{OUT} ..... 0.8V to 25V
Operating Junct. Temp (T_J) ..... –20°C to +125°C

Thermal Resistance (4)

	θ_{JA}	θ_{JC}	
SOIC8E	50	10	°C/W

Notes:

1) Exceeding these ratings may damage the device.
2) The maximum allowable power dissipation is a function of the maximum junction temperature T_J(MAX), the junction-to-ambient thermal resistance θ_{JA}, and the ambient temperature T_A. The maximum allowable continuous power dissipation at any ambient temperature is calculated by $P_D(MAX)=(T_J(MAX)-T_A)/\theta_{JA}$. Exceeding the maximum allowable power dissipation will cause excessive die temperature, and the regulator will go into thermal shutdown. Internal thermal shutdown circuitry protects the device from permanent damage.
3) The device is not guaranteed to function outside of its operating conditions.
4) Measured on JESD51-7, 4-layer PCB.

ELECTRICAL CHARACTERISTICS

V_{IN} = 12V, V_{EN} = 2.5V, V_{COMP} = 1.4V, T_A= +25°C, unless otherwise noted.

Parameter	Symbol	Condition	Min	Typ	Max	Units
Feedback Voltage	V_{FB}	4.5V < V_{IN} < 28V	0.776	0.8	0.824	V
Upper Switch On Resistance	$R_{DS(ON)}$	V_{BST} – V_{SW} = 5V		150		mΩ
Upper Switch Leakage		V_{EN} = 0V, V_{SW} = 0V, V_{IN} = 28V		1		μA
Current Limit			4.0	4.7		A
COMP to Current Sense Transconductance	G_{CS}			9		A/V
Error Amp Voltage Gain (5)				200		V/V
Error Amp Transconductance		I_{COMP} = ±3μA	40	60	80	μA/V
Error Amp Min Source current		V_{FB} = 0.7V		5		μA
Error Amp Min Sink current		V_{FB} = 0.9V		–5		μA
VIN UVLO Threshold			2.7	3.0	3.3	V
VIN UVLO Hysteresis				0.35		V
Soft-Start Time (5)		0V < V_{FB} < 0.8V		1.5		ms
Oscillator Frequency		R_{FREQ} = 100kΩ		900		kHz
Shutdown Supply Current		V_{EN} = 0V		12	20	μA
Quiescent Supply Current		No load, V_{FB} = 0.9V		100	125	μA
Thermal Shutdown				150		°C
Thermal Shutdown Hysteresis				15		°C
Minimum Off Time (5)				100		ns
Minimum On Time (5)				100		ns
EN Up Threshold			1.35	1.5	1.65	V
EN Hysteresis				300		mV

Note:

5) Guaranteed by design.

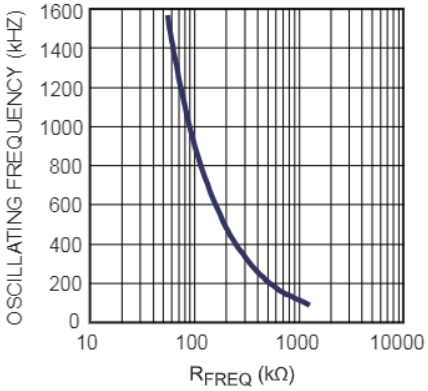
PIN FUNCTIONS

SOIC Pin #	Name	Description
1	SW	Switch Node. This is the output from the high-side switch. A low forward drop Schottky diode to ground is required. The diode must be close to the SW pins to reduce switching spikes.
2	EN	Enable Input. Pulling this pin below the specified threshold shuts the chip down. Pulling it up above the specified threshold or leaving it floating enables the chip.
3	COMP	Compensation. This node is the output of the error amplifier. Control loop frequency compensation is applied to this pin.
4	FB	Feedback. This is the input to the error amplifier. The output voltage is set by a resistive divider connected between the output and GND which scales down V_{OUT} equal to the internal +0.8V reference.
5	GND Exposed Pad	Ground. It should be connected as close as possible to the output capacitor to shorten the high current switch paths. Connect exposed pad to GND plane for optimal thermal performance.
6	FREQ	Switching Frequency Program Input. Connect a resistor from this pin to ground to set the switching frequency.
7	VIN	Input Supply. This supplies power to all the internal control circuitry, both BS regulators and the high-side switch. A decoupling capacitor to ground must be placed close to this pin to minimize switching spikes.
8	BST	Bootstrap. This is the positive power supply for the internal floating high-side MOSFET driver. Connect a bypass capacitor between this pin and SW pin.

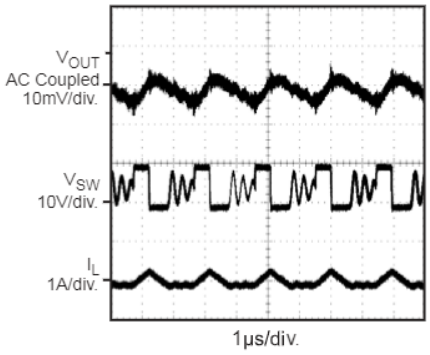
TYPICAL PERFORMANCE CHARACTERISTICS

V_{IN} = 12V, V_{OUT}=5V, C1 = 10µF, C2 = 22µF, L1= 10µH, T_A = +25°C, unless otherwise noted.

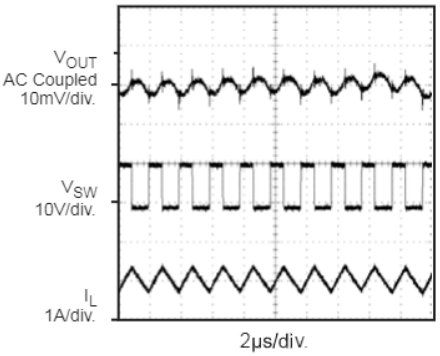
Oscillating Frequency vs. R_{freq}



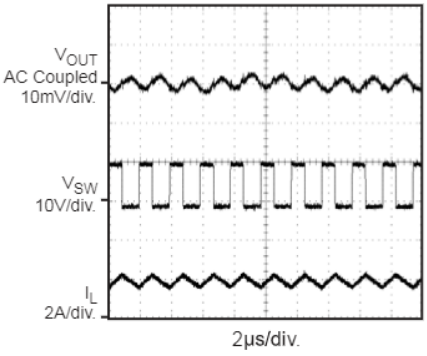
Steady State
I_{OUT}=0.1A, f_{SW}=500kHz



Steady State
I_{OUT}=1A, f_{SW}=500kHz



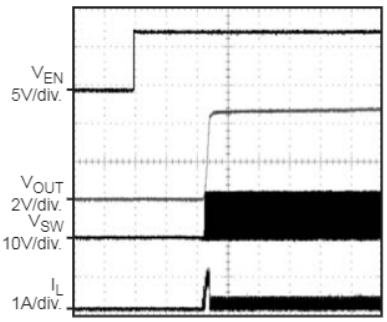
Steady State
I_{OUT}=2A, f_{SW}=500kHz



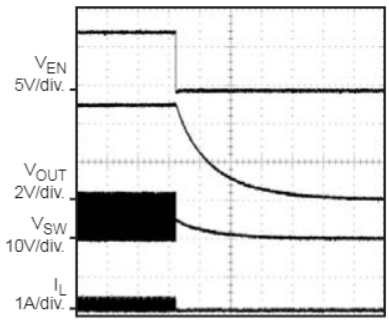
TYPICAL PERFORMANCE CHARACTERISTICS *(continued)*

V_{IN} = 12V, C1 = 10µF, C2 = 22µF, L1 = 10µH, f_{SW}=500kHz, and T_A = +25°C, unless otherwise noted.

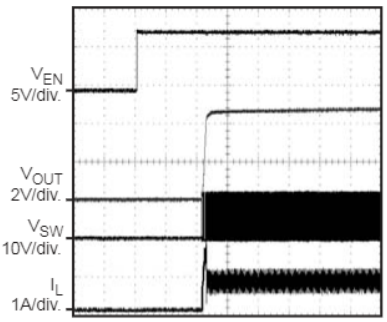
Startup
I_{OUT} = 0.1A



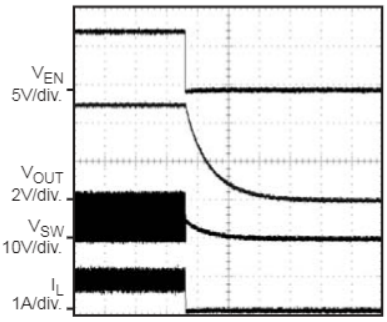
Shutdown
I_{OUT} = 0.1A



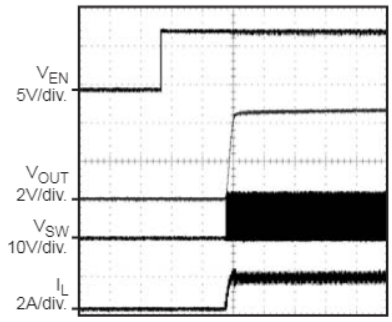
Startup
I_{OUT} = 1A



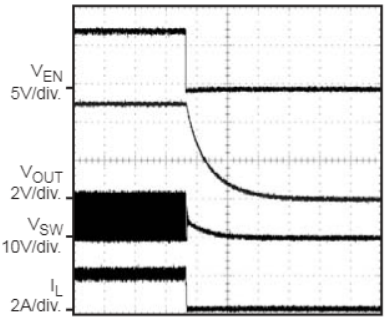
Shutdown
I_{OUT} = 1A



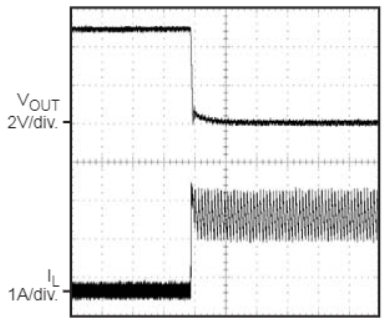
Startup
I_{OUT} = 2A



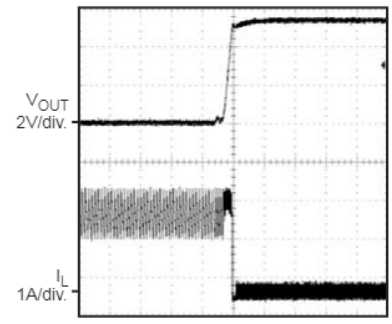
Shutdown
I_{OUT} = 2A



Short Circuit Entry
I_{OUT} = 0.1A to Short



Short Circuit Recovery
I_{OUT} = Short to 0.1A



BLOCK DIAGRAM

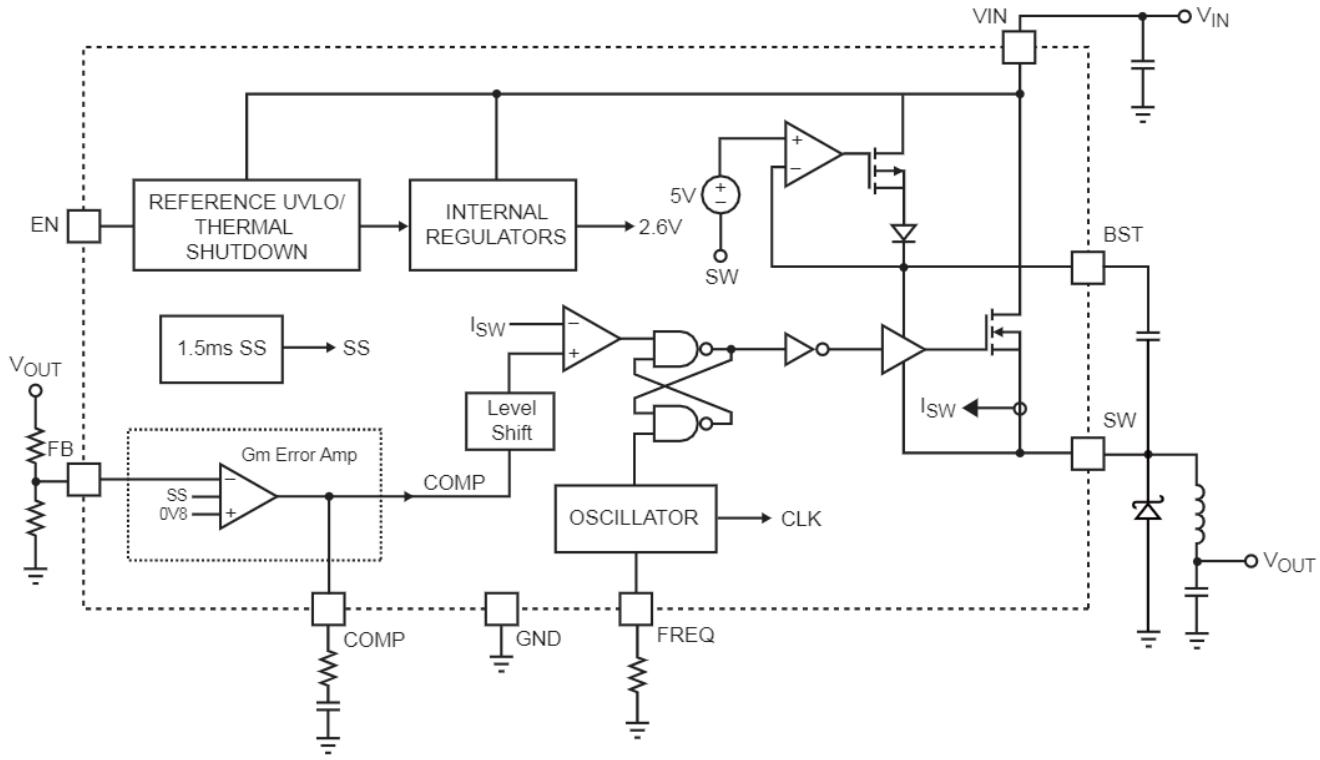


Figure 1—Functional Block Diagram

OPERATION

The MP1584 is a variable frequency, non-synchronous, step-down switching regulator with an integrated high-side high voltage power MOSFET. It provides a highly efficient solution with current mode control for fast loop response and easy compensation. It features a wide input voltage range, internal soft-start control and precision current limiting. Its very low operational quiescent current makes it suitable for battery powered applications.

PWM Control

At moderate to high output current, the MP1584 operates in a fixed frequency, peak current control mode to regulate the output voltage. A PWM cycle is initiated by the internal clock. The power MOSFET is turned on and remains on until its current reaches the value set by the COMP voltage. When the power switch is off, it remains off for at least 100ns before the next cycle starts. If, in one PWM period, the current in the power MOSFET does not reach the COMP set current value, the power MOSFET remains on, saving a turn-off operation.

TYPICAL APPLICATION CIRCUITS

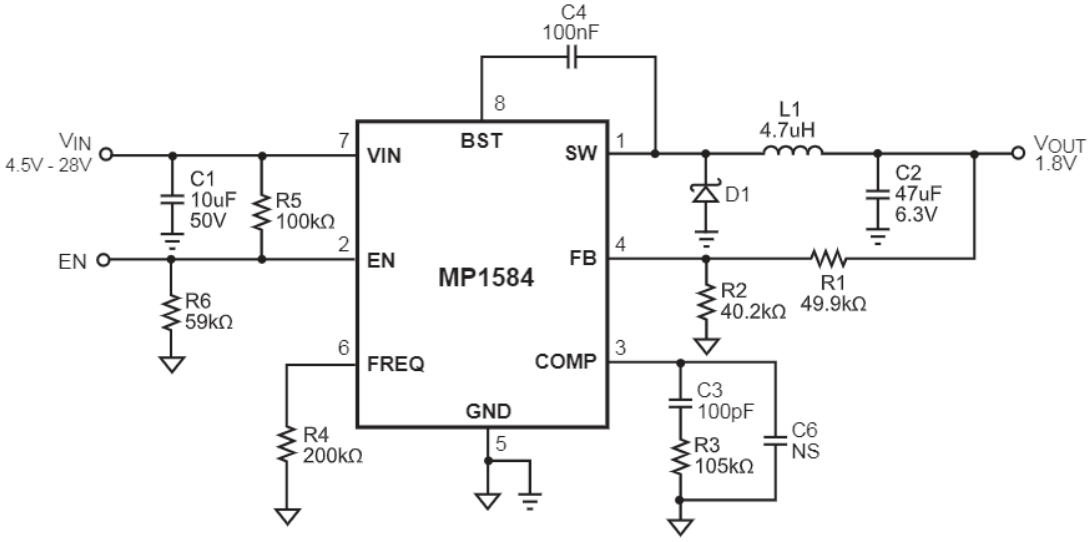


Figure 3—1.8V Output Typical Application Schematic

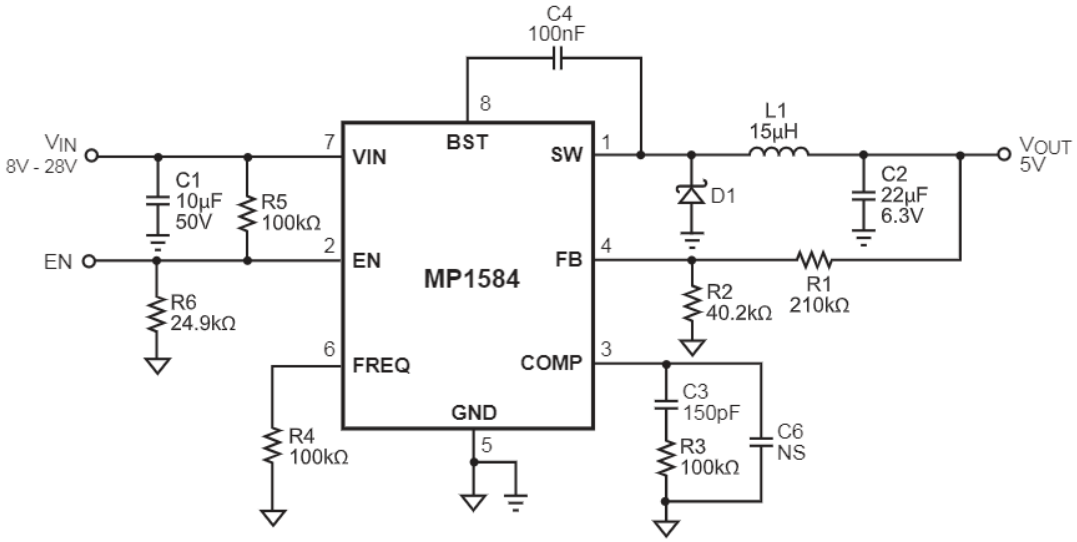


Figure 4—5V Output Typical Application Schematic