ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y TECNOLOGÍA

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Trabajo de Fin de Grado

GENERACIÓN DINÁMICAS DE SECUENCIAS PARA ROBOTS PARALELOS

PLATAFORMA DE STEWART GOUGH

Autores:

Facundo Esteban Scozzina

Ángel Daniel Delgado González

Tutores:

Jonay Tomás Toledo Carrillo

Antonio Luis Morell González

Julio, 2016

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

1. Introducción

1.1 Objetivos del Trabajo de Fin de Grado

1.1.1 Resumen

Este proyecto parte del trabajo de fin de grado "Sistema de posicionamiento XYZ" redactado por Kevin Mínguez Olivero en el que se llevó a cabo la creación de una plataforma de Stewart Gough, que es un sistema robótico de posicionamiento espacial, capaz de colocarse en cualquier punto XYZ y con una inclinación prestablecida dentro de su espacio de trabajo. Dicho proyecto constó de la programación del robot; el diseño, fabricación e instalación de la electrónica necesaria adicional; así como el diseño, creación y ensamblaje del propio sistema robótico.

Partiendo de esta base, el presente trabajo de fin de grado tiene como objetivo el estudio y mejora de algunos aspectos del robot, que se han considerado interesantes y útiles para la aplicación. Los hitos que se han decidido llevar a cabo son los siguientes:

- Creación de una simulación del movimiento real de la plataforma robótica.
- Estudio y realización de una gráfica tridimensional explicativa del espacio de trabajo del robot.
- Unificación de los códigos originales en uno sólo que será procesado completamente en Arduino.
- Cambio del Arduino Mega 2560 del que disponía la plataforma por Arduino Due, así como las modificaciones de código asociadas a dicha sustitución.
- Implementación de un método de realimentación basado en el funcionamiento simultáneo de un controlador de posición y otro de velocidad, con el objetivo de mejorar el movimiento de los actuadores durante sus trayectorias.

1.1.2 Abstract

This project started from the final degree project "Sistema de posicionamiento XYZ" written by Kevin Mínguez Olivero in which carried out the creation of Stewart Gough platform, that it's a spatial positioning robotic system, able to placed at any XYZ point and with a preset inclination within his workspace. This project consists of the robot programming; the design, manufacturing and installation of the necessary electronic; as well as the design, creation and assembly of the robotic system.

On this basis, the present final degree project aims to the study and improvement of some robot aspects, which are considered interesting and useful for the application. The milestones that are have decided to carry out are the following:

- Creation of a real movement simulation of the robotic platform.
- Study and realization of an explanatory three dimensional graph of the robot workspace.
- Unification of the original codes into only one that will be fully processed in Arduino.

- Change of the Arduino Mega 2560 that was integrated in the platform by an Arduino Due, as well as code modifications associated with that replacement.
- Implementation of a feedback method based on the simultaneous operation of a position controller and other of velocity, with the aim of improving actuator's movements during his trajectories.

1.2 Robot paralelo y plataforma de Stewart-Gough

Un mecanismo robótico es un sistema de cuerpos rígidos, llamados enlaces o links, conectados entre sí por articulaciones que pueden ser prismáticas (movimiento lineal) o de rotación (movimiento rotatorio). Si varios miembros están interconectados siguiendo un único camino, se dice que forman una cadena cinemática en serie, mientras que si algún miembro está conectado a otro mediante dos o más caminos diferentes se dice que forman una cadena cinemática en paralelo.

Sabiendo lo anterior, se puede definir un robot paralelo como una cadena cinemática en paralelo conformada por dos cuerpos (una base y un efector final), unidos entre sí por múltiples cadenas cinemáticas en serie. Un robot es capaz de interaccionar con el medio que le rodea mediante el efector final, y para ello es necesario describir su posición y orientación en el espacio mediante lo que se conoce como la pose del robot. Por lo tanto, para definir completamente la pose el efector final serán necesarias tres coordenadas de traslación y tres rotaciones, y por esta razón se dice que el robot posee seis grados de libertad.

La plataforma de Stewart-Gough es un tipo robot paralelo que debe su nombre a Gough, quien estableció las primeras nociones de un robot con estas catacterísticas y construyó el primer prototipo en 1955, y a Stewart quien propuso una topología similar para implementar un simulador de vuelo.

Este tipo de robot posee seis actuadores prismáticos y dos bases, una fija y otra móvil. Los actuadores están unidos a ambas bases siguiendo un diseño concreto y con ellos se consigue el total control de la plataforma móvil, tanto su posición final como su inclinación, además de permitir el movimiento longitudinal, rotacional así como movimientos combinados que aportan al efector final una gran variedad de movimientos y ángulos.

Estas características lo convierten en un sistema que puede ser empleado para diversas aplicaciones sobre todo en el ámbito industrial como simuladores de conducción y vuelo (aplicación más empleada para estos robots), robots manipuladores que pueden llegar a tener una gran precisión frente a otro tipo de manipuladores, sistema de acoplamiento de bajo impacto para vehículos espaciales (empleado en la NASA para facilitar el acoplamiento entre vehículos espaciales), en grandes telescopios para realizar movimientos lentos de algunos elementos que necesitan mucha precisión, operaciones para montar y desmontar, manipulación de piezas y aplicaciones de soldadura por puntos con retroalimentación de fuerza, etc.

En nuestro caso en particular, la plataforma será empleada para investigación y pruebas por parte del departamento de Ingeniería Informática y de Sistemas de la facultad.

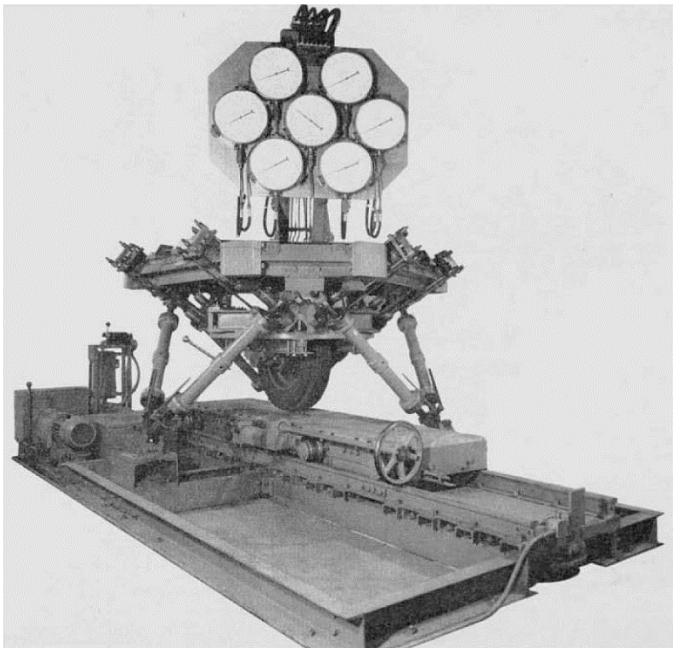


Figura 1. Primer prototipo de plataforma de Gough

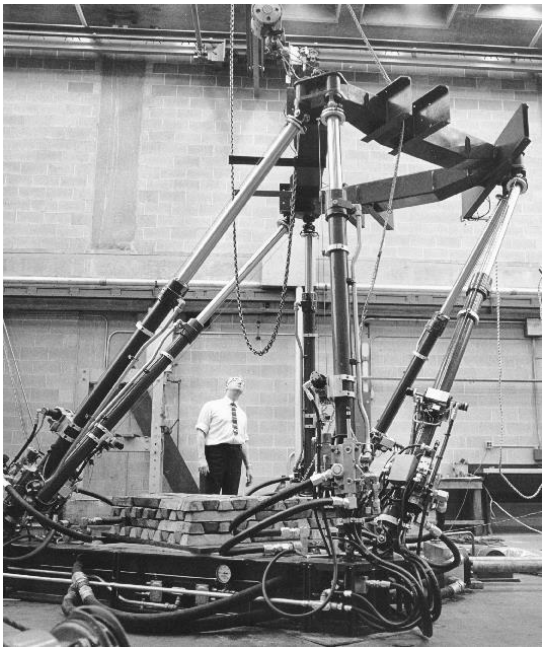


Figura 2. Primer prototipo de simulador de vuelo de Stewart

2. Diseño y fabricación de la plataforma Stewart

El Diseño y fabricación de la Plataforma Stewart fue realizado en un proyecto anterior de la Universidad de la Laguna. A continuación se presentan los datos del proyecto en cuestión.

- Título: Sistema de Posicionamiento XYZ.
- Titulación: Ingeniería Industrial Electrónica y Automática.
- Autor: Kevin Mínguez Olivero.
- Fecha: Septiembre, 2014.

2.1 Diseño de la plataforma

Existen varios diseños para la plataforma Stewart dependiendo, entre otras cosas, de la situación de los actuadores en el sistema, el tamaño de las bases y su geometría… Para el presente proyecto se ha optado por un diseño en el que sus bases siguen una forma triangular.

2.1.1 Bases fija y móvil

La base inferior, o fija, tiene un tamaño ligeramente superior a la base superior o móvil. Además ambas bases están dispuestas de manera inversa, es decir, en la proyección en planta del sistema podremos ver como los vértices de cada triángulo coincidirán con un lado del triángulo opuesto. A continuación se muestra una imagen donde podremos ver claramente esta configuración descrita.

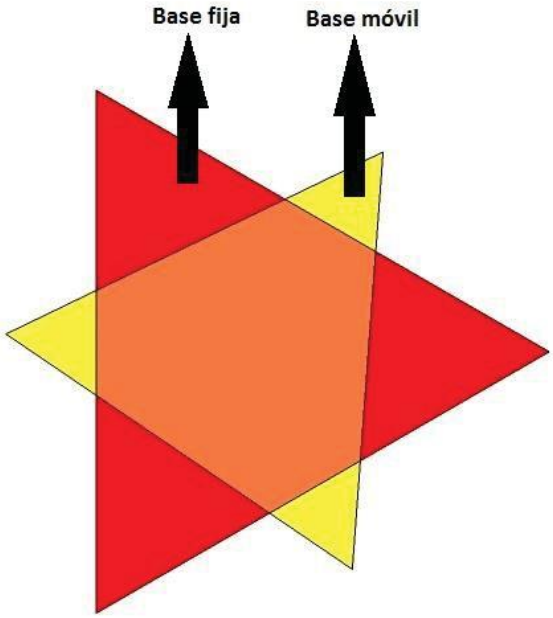


Figura 3. Forma y configuración de bases escogida

A pesar de ser esta la configuración base del diseño, debido a ciertas dificultades a la hora del ensamblaje, el diseño final ha sido ligeramente modificado para facilitar la instalación de las articulaciones de cada actuador. En el diseño final han sido recortadas las esquinas de cada triángulo resultando cada una de las bases en un polígono hexagonal no equilátero. A continuación se muestra el diseño final de las bases de la plataforma, donde el polígono azul se correspondería con la base móvil y el amarillo con la base fija:

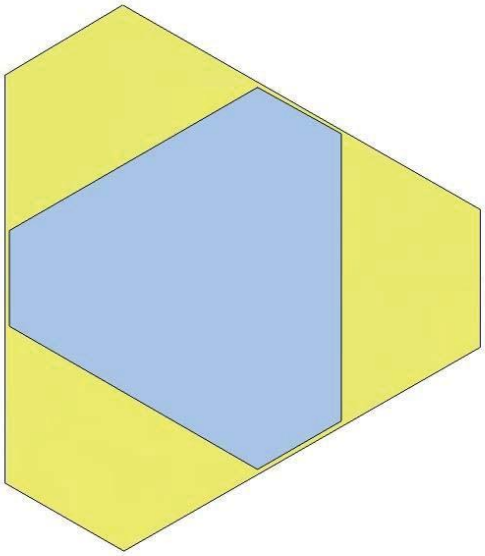


Figura 4. Forma final empleada en las bases

Finalmente, debe aclararse que, por la forma que presentan las articulaciones usadas en la base inferior y por la manera de insertarlas en ella, ha tenido que diseñarse esta base de un tamaño menor que el que aparece en la imagen anterior. Esto no modificaría el diseño expuesto ya que los puntos de anclaje de cada actuador seguirían en el lugar anteriormente indicados, formando, al unir todos estos puntos de anclaje, una base imaginaria equivalente al polígono mayor de la imagen previa. La siguiente imagen aclarará este aspecto:

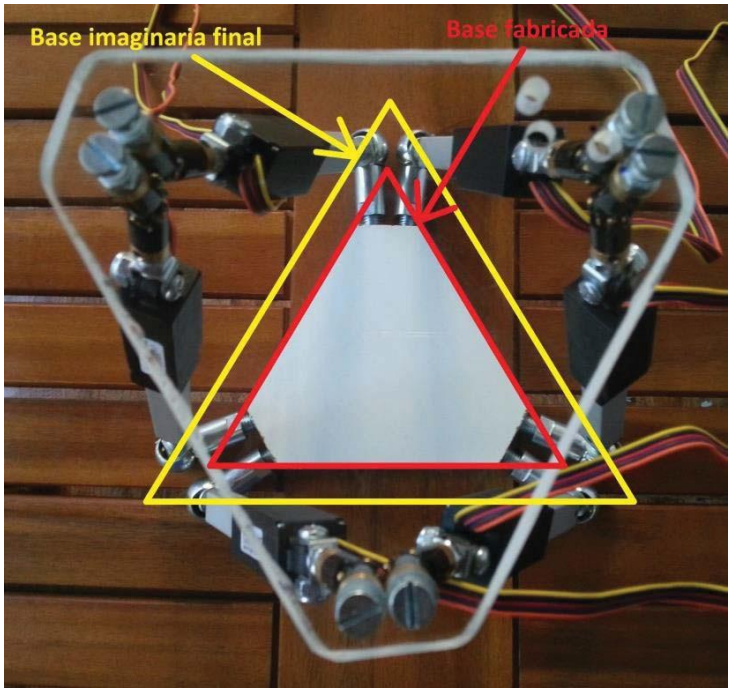


Figura 5. Base inferior imaginaria

Cabe aclarar que el posterior estudio de la cinemática inversa del robot haría uso de los puntos de anclaje, indistintamente del tamaño de la base, por lo que esta ligera modificación de diseño no sería relevante.

Como podemos ver, el lado resultante del corte del triángulo ha sido empleado para instalar lateralmente las articulaciones inferiores de cada motor. En cuanto a la base móvil, las articulaciones superiores fueron instaladas de forma vertical y no lateralmente.

A continuación se presentan las dimensiones empleadas para ambas bases, inicialmente, de igual tamaño, así como las dimensiones de la base imaginaria que se formaría a partir de los puntos de anclaje:

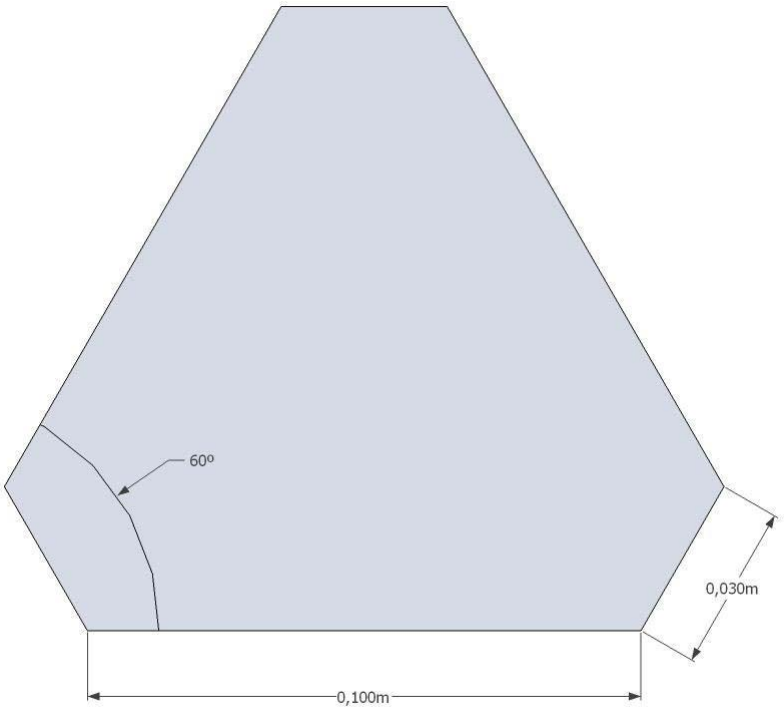


Figura 6. Dimensiones usadas para las bases

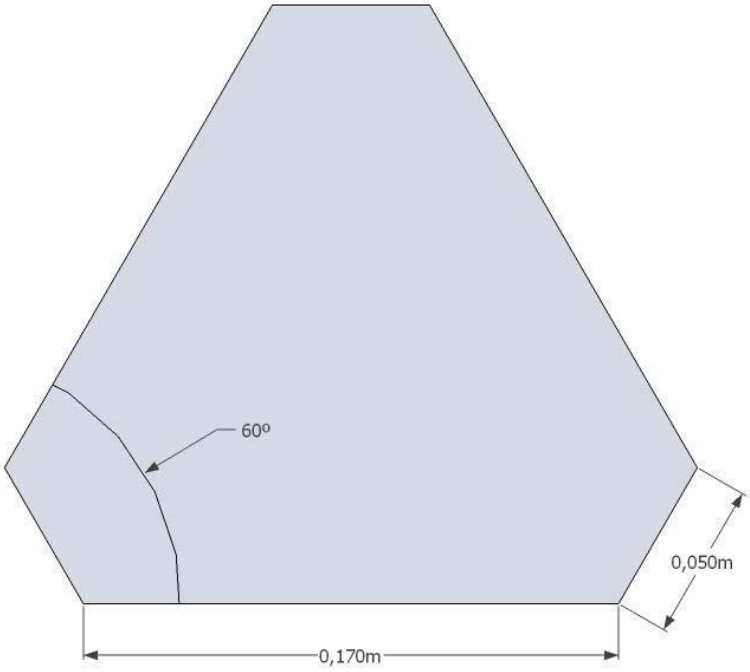


Figura 7. Dimensiones de la base inferior imaginaria

2.1.2 Actuadores

Los actuadores escogidos para la plataforma han sido 6 motores lineales Firgelli de la serie L16 de 50 mm de recorrido. Estos motores tienen una velocidad máxima de 32 mm/s (sin carga) y una fuerza máxima de 50 N. En cuanto a sus características eléctricas, se recomienda una alimentación de entre 0 y 15 VDC y tiene una corriente de bloqueo de 650 mA (con alimentación de 12 VDC). Los planos con las dimensiones de los motores serán proporcionados en los anexos.

Figura 8. Motor lineal Firgelli, serie L16

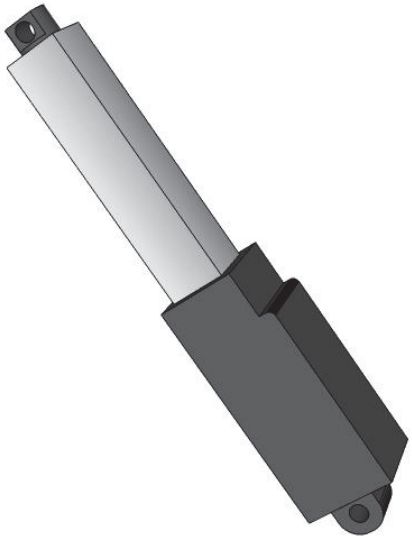


Figura 9. Diseño 3D (SketchUp) de motor empleado

La ventaja de estos motores lineales es que tienen incorporado un potenciómetro cuyo cursor es solidario con la parte móvil del actuador. Esto es lo que nos proporciona la información necesaria para realizar el control del sistema, transformando la señal de voltaje de dicho potenciómetro en su valor de posición en cada momento mediante la placa Arduino.

Los motores están anclados a cada una de las bases formando cierto ángulo con ambas y de forma inversa entre dos motores contiguos. Esta configuración de los actuadores es la que ha influido en la forma y situación de las bases, descrita anteriormente. A continuación se muestra una imagen donde se aclara la configuración de los actuadores en el diseño:

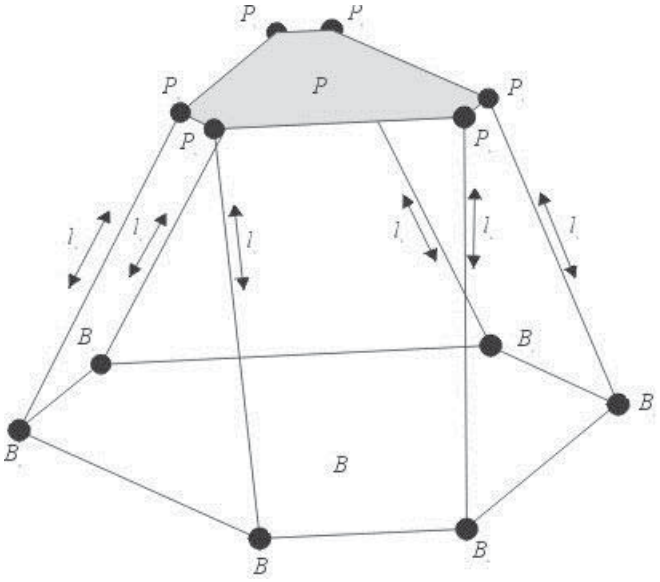


Figura 10. Ubicación de actuadores y articulaciones

2.1.3 Articulaciones

Las articulaciones escogidas para anclar los actuadores a ambas bases han sido articulaciones universales o cardan para la base superior y rótulas para la inferior. La combinación de estas articulaciones permite a los motores moverse en un amplio rango de ángulos permitiendo la libertad necesaria a la base móvil.

Los cardanes empleados en este caso cuentan con una junta simple y su águlo máximo de funcionamiento es de 45º.

Figura 11. Articulación universal o cardan

En cuanto a las rótulas empleadas, cuentan con un ángulo de trabajo de 30º en torno a su eje longitudinal, y de 36º en torno a su eje transversal.

Figura 12. Articulación tipo rótula

2.1.4 Diseño 3D de la plataforma Stewart

A continuación se mostrará una representación 3D del diseño final. En esta representación las articulaciones están simplificadas como esferas. Además de esto, y como se comentó en el apartado 2.2.1, en este diseño se ha usado la base inferior imaginaria, por lo que, tanto el anclaje de los actuadores como la forma de la base inferior resultante es un poco diferente a la empleada en el diseño físico final. A pesar de esto, esta representación nos sirve para apreciar el diseño general del robot así como la situación de los actuadores y las bases. El diseño está realizado mediante la herramienta de diseño gráfico y modelado 3D SketchUp, perteneciente en la actualidad a la empresa Trimble.

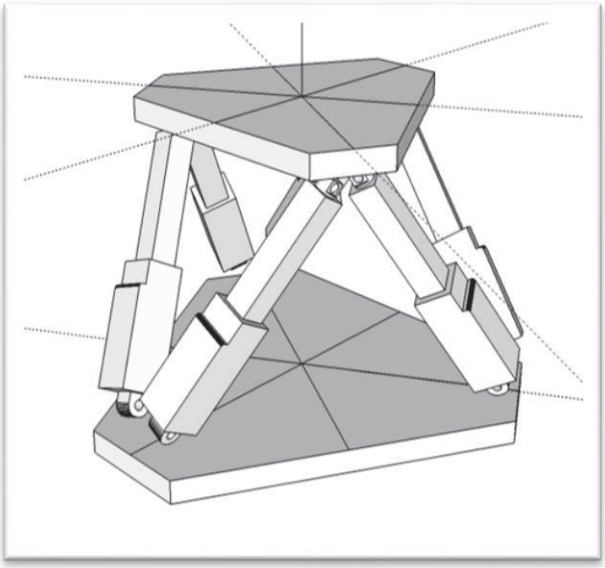


Figura 13. Diseño 3D básico, plataforma Stewart

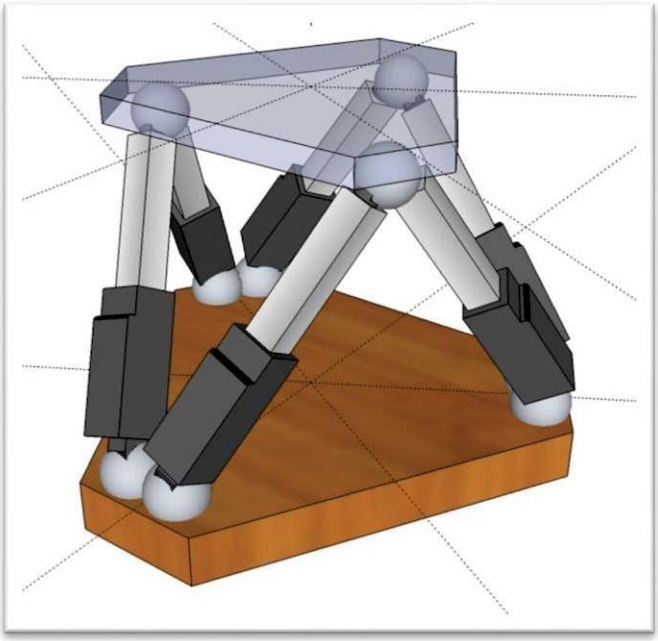


Figura 14. Diseño 3D final simplificado, plataforma Stewar

2.2 Fabricación de la plataforma

El proceso de fabricación ha consistido en la preparación de las bases, en el ensamblaje de las articulaciones en estas y, finalmente, en la unión con los actuadores, consiguiendo la conexión entre ambas bases. Los pasos realizados en este proceso fueron los siguientes:

- La base inferior se fabricó mediante un tablero de madera DM de 2 cm de grosor. Esta se cortó siguiendo el diseño mencionado en el apartado anterior.
- Lo mismo se realizó para la base superior, pero esta fue obtenida de una lámina de metacrilato de 1 cm de espesor.
- El siguiente paso consistió en unir las articulaciones tipo rótula a la plataforma inferior. Estas se insertaron en la madera lateralmente en los lados más pequeños del polígono que forma la base tras ser cortada según el diseño.
- En la base superior se realizaron también las perforaciones donde irían los tornillos que sujetarían los cardanes y, tras esto, se ensamblaron estas seis articulaciones. En este caso también se situaron en los lados más pequeños de la base pero de manera vertical.
- Finalmente se acoplaron los seis actuadores en sus correspondientes articulaciones, consiguiendo la conexión final entre las bases fija y móvil.

A continuación se aporta una imagen con la ubicación de las articulaciones en ambas bases:

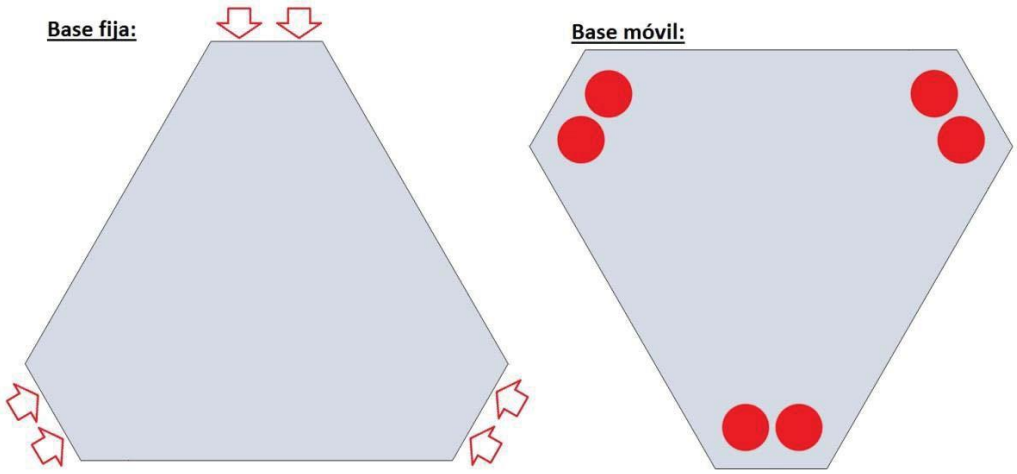


Figura 15. Ubicación de las articulaciones en las base

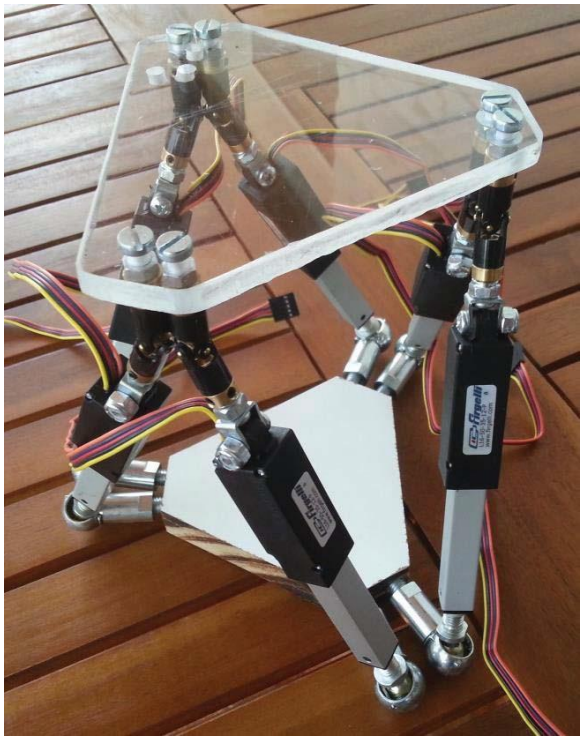


Figura 16. Plataforma robótica Stewart final

3. Diseño y fabricación de la placa PCB para electrónica adicional del sistema

El Diseño y fabricación de la Placa PCB fue realizado en un proyecto anterior de la Universidad de la Laguna. A continuación se presentan los datos del proyecto en cuestión.

- Título: Sistema de Posicionamiento XYZ.
- Titulación: Ingeniería Industrial Electrónica y Automática.
- Autor: Kevin Mínguez Olivero.
- Fecha: Septiembre, 2014.

Como ya fue comentado en la introducción del proyecto, para la realización del mismo se ha empleado la placa Arduino Mega 2560, con la que se ha implementado el control del sistema. La placa Arduino puede ser, en proyectos simples, suficiente para proporcionar la alimentación y realizar los cálculos necesarios mediante su microcontrolador. Sin embargo, para esta aplicación era necesario un nivel de voltaje superior al que puede proporcionar la placa para la alimentación de los motores. Aparte de esto, era preciso hacer uso de varios componentes externos para proporcionar al sistema:

- Alimentación independiente de la obtenida a través de un ordenador, y que además fuera única y suficiente para el funcionamiento de la plataforma.
- Una conexión sencilla y robusta al Arduino. Para ello se buscaba diseñar una placa de tipo shield.
- Una alternativa a la conexión USB de la Arduino, empleando un RS232 con el que aportar una salida serial a la misma.
- Capacidad para controlar de manera simultánea e independiente los 6 motores.

Por ello, ha sido necesario realizar el diseño y fabricación de una placa PCB que albergara la electrónica de potencia y algunos otros componentes necesarios para cumplir estos requerimientos.

Sería importante remarcar que antes del diseño de dicha placa hubo un proceso de pruebas previas en el laboratorio. En estas pruebas se testearon independientemente algunos de los subcircuitos que formarían la placa final, como los puentes en H o el circuito de alimentación (de los que se hablará con más profundidad en el siguiente apartado) para comprobar su funcionamiento y la configuración correcta

que deberían tener al ser instalados en la placa. Estos test iniciales también fueron muy útiles para plantear la correcta programación del código para Arduino.

Las pruebas fueron desarrolladas en protoboards con algunos motores (tanto convencionales como lineales) así como haciendo uso de los instrumentos del laboratorio, como fuentes de alimentación, osciloscopios y multímetros.

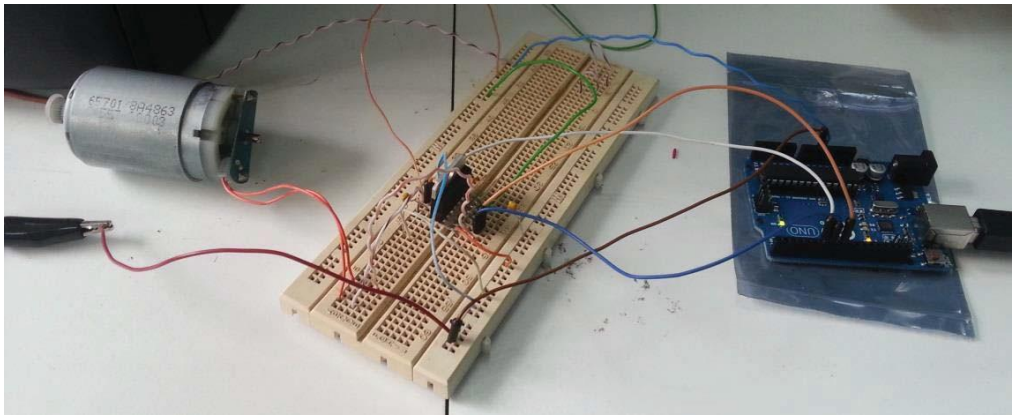


Figura 17. Primeras pruebas previas al diseño de la placa PCB

3.1 Diseño esquemático del circuito

Para el diseño de la placa PCB han sido empleadas algunas utilidades del software OrCAD de la empresa Cadence Design Systems, Específicamente se ha utilizado el módulo Capture para realizar el diseño esquemático y el módulo Layout para realizar el diseño final de la placa PCB.

En primer lugar se realizó el diseño esquemático en Capture de la electrónica que formaría la placa:

- En cuanto a la electrónica de potencia, el circuito comienza con un conector que recibiría la alimentación del mismo. La alimentación debe ser de entre 10-15 V de corriente continua, dependiendo de la alimentación que se quiera proporcionar a los motores. Este voltaje
  puede ser proporcionado mediante una fuente de voltaje, un transformador o una pila, si fuera preciso. Durante el desarrollo de las pruebas en el laboratorio se usó inicialmente una fuente de alimentación y posteriormente se sustituyó esta por un pequeño transformador.
  La tierra del sistema parte desde el terminal correspondiente del conector hasta todos los puntos donde sea preciso.
- Desde el conector inicial se alimentaría a los motores a través de varios L298 (de los cuales se hablará a continuación). Además de alimentar a los actuadores, este voltaje inicial es enviado a un regulador de tensión ajustable LM2575 configurado para emitir un voltaje de salida de 5V. Este voltaje de salida del regulador será el encargado de alimentar a toda la

lógica del sistema así como a la placa Arduino a través de su pin de 5V.

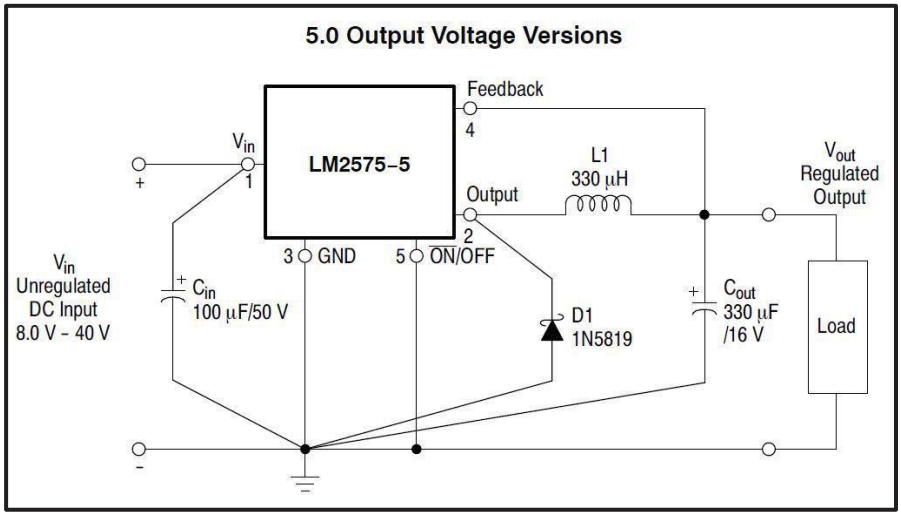


Figura 18. Configuración LM2575 para voltaje de salida de 5V

- Otro componente crucial para el funcionamiento del sistema incluido en el diseño fue el L298, componente que cuenta con 2 puentes en H.

  Un puente en H es un circuito que permite controlar la dirección y velocidad de un motor modificando su señal de voltaje de alimentación, empleando para ello, señales lógicas. El cambio de dirección se consigue cambiando las señales lógicas enviadas a cuatro puertas lógicas controladas de dos en dos por una única señal. Cada una de estas puertas lógicas, cuando emiten un '1' lógico activan un transistor que cierra una parte del circuito de alimentación del motor. Sabiendo esto, controlar la dirección del motor solo se basa en controlar la señal que se envía a cada puerta para cerrar el circuito en un sentido o en otro. Por otra parte, en el componente existe un pin 'Enable' que abre o cierra el circuito general. Actuando sobre este pin con una señal PWM*, es posible regular la velocidad, variando el ciclo de dicha señal.

  Tal como se comentó al principio, cada L298 incluye dos puentes en H, por lo que fue preciso incluir tres de estos encapsulados para controlar los seis actuadores.

  : Una señal **PWM (Pulse Width Modulation) es una señal formada por pulsos cuyo ancho es variable. Esto se suele emplear para modular señales analógicas mediante una señal digital. Dependiendo de la anchura de los pulsos, el voltaje contínuo de la señal analógica asociada también varía. Por ejemplo, si el tiempo en el que cada pulso está en alta se corresponde con el 40% del periodo total, el voltaje resultante al medir la señal modulada será el 40% del voltaje máximo.*

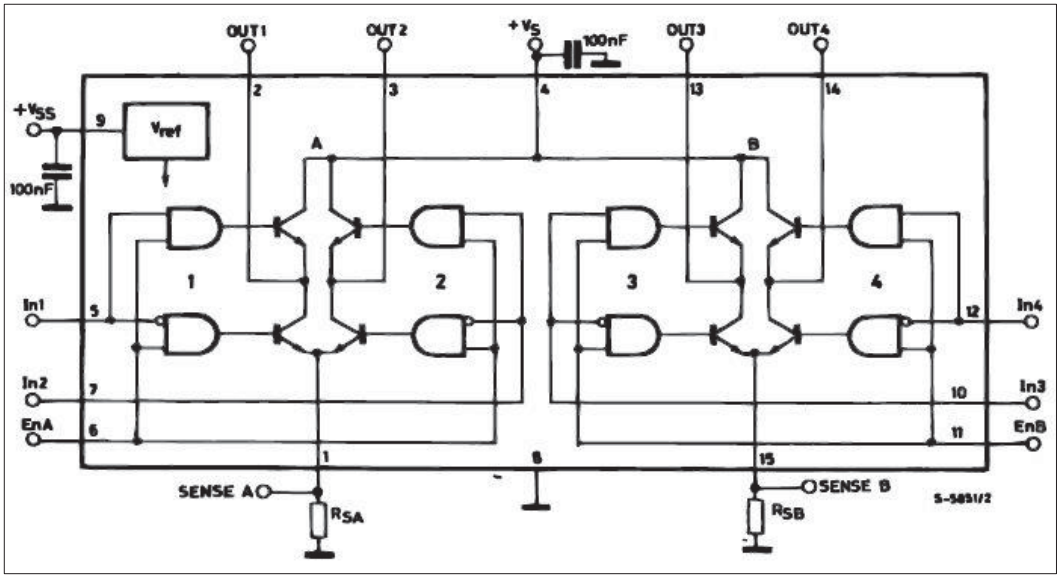


Figura 19. Composición de un L298 (2 puentes en H)

- Teniendo en cuenta que la placa Arduino Mega solo cuenta con un puerto USB para realizar una conexión sencilla con otros dispositivos, se ha decidido incluir en el diseño un circuito ST232 mediante el que se podrá transformar las señales emitidas por la placa, a través de los pines TX y RX, en el protocolo de comunicación serial RS232, incrementando de esta forma las posibilidades de comunicación del sistema, adaptando, por ejemplo un cable serial de 9 pines. Esta funcionalidad extra no será necesaria para el presente proyecto pero es una buena forma de facilitar una posterior mejora del mismo, si se diera el caso, aportando una alternativa a la comunicación por USB.
- Las señales necesarias que se deben aportar o se deben recibir de los motores se han centralizado en 6 filas -una por motor- de 5 pines cada una, con el fin de facilitar la instalación y desinstalación de los mismos. Los conectores de los motores cuentan con 5 cables que tienen las siguientes funciones:
  - Naranja: Referencia negativa del potenciómetro incluido (en nuestro caso, conectado a tierra).
  - Morado: Cursor del potenciómetro. Este cable nos proporcionará la información necesaria para saber la posición del motor.
  - Rojo: Alimentación (máx. 15V).
  - Negro: Tierra.
  - Amarillo: Referencia positiva del potenciómetro (en este caso, 5V)
- Por último indicar que se ha asociado un condensador de desacoplo a cada circuito integrado, que se colocará cerca de sus pines de alimentación y tierra.

Una vez incorporados los componentes y tras realizar las conexiones entre ellos, se pasa a generar, en Capture, los ficheros necesarios para el posterior diseño de la PCB en Layout.

A continuación se muestra el esquemático del circuito diseñado, aunque también será añadido al proyecto como un anexo para poder revisarlo con mayor claridad:

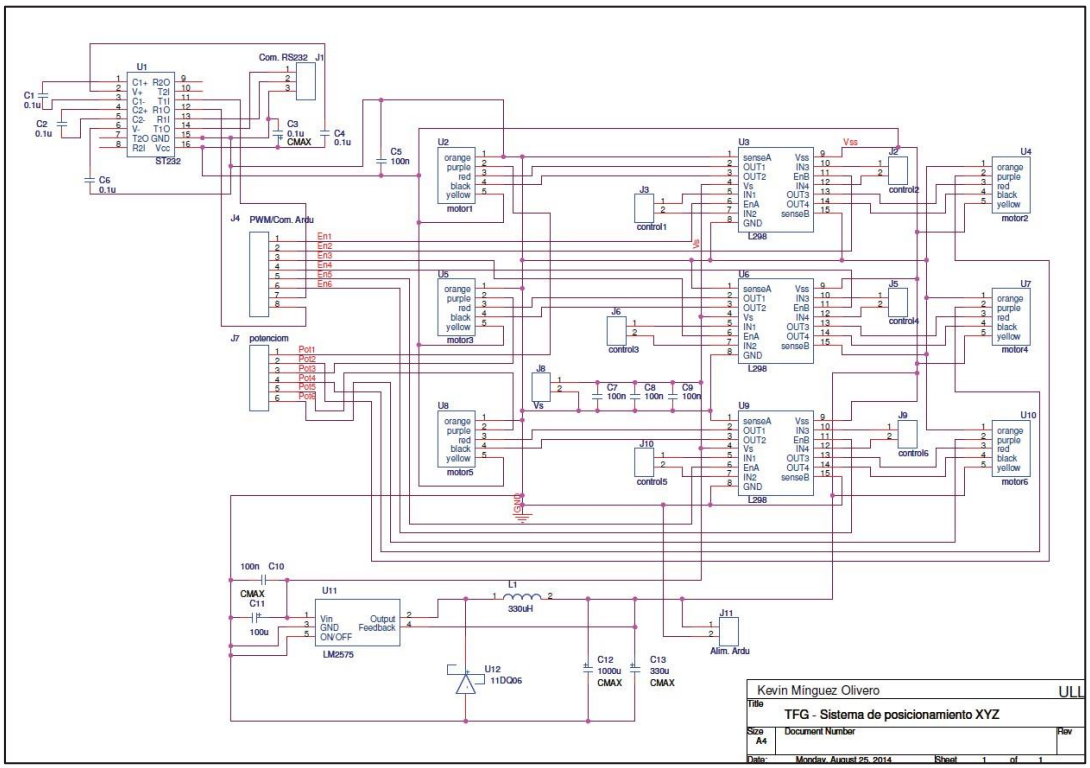


Figura 20. Diseño esquemático de la placa PCB

3.2 Diseño de la placa PCB

Una vez finalizado el diseño esquemático del circuito, el siguiente paso consiste en realizar el diseño de la placa PCB. Este proceso incluye la creación de footprints de los componentes si fuera necesario, la colocación de los componentes, el trazado de las pistas (ruteo), la asignación de pads y taladros y generación de ficheros para la fabricación. Es importante nombrar que la placa realizada cuenta con dos capas para el trazado de pistas, la capa de soldadura y la capa de componentes, así como que en ella se ha empleado un montaje de componentes tipo THT (*Through- Hole Technology*).

En el proyecto se decidió realizar esta placa como un shield para Arduino, ya que ambas placas compartían muchos de sus pines. Un shield es una placa diseñada teniendo en cuenta los pines existentes en la Arduino para colocar los suyos en la misma posición y poder conectar ambas placas a presión, en lugar de mediante cableado. Realizar esta placa como un shield simplificó en gran medida la interconexión, teniendo solamente que conectar la alimentación, los motores y el cable RS232, si este fuera a utilizarse. Además se consiguió una mayor compactación en el diseño, con la comodidad que esto conlleva a la hora del transporte y del montaje y desmontaje de la circuitería.

Como fue mencionado anteriormente, el programa empleado para el diseño de la placa fue el Layout. Una vez generados los ficheros finales del esquemático en Capture, estos se asocian a un nuevo archivo en Layout. Al hacerlo, aparecen en el espacio de trabajo del programa todos los componentes necesarios para fabricar la placa, así como todas las conexiones entre los pines de cada uno, aunque de manera 'virtual'. Estas conexiones 'virtuales' sirven para saber el origen y destino de cada pista a la hora de comenzar el trazado de las mismas. Puede ocurrir que al vincular el fichero de Capture con Layout algunos de los componentes no tengan asociada su huella (footprint) o directamente que esta no exista. En este caso es necesario buscar la huella dentro de la librería que incluye el programa para asociarla o crear manualmente la huella del componente mediante un gestor de librerías. En el proyecto actual fue necesario crear algunos footprints como el del L298 o el LM2575 para poder emplearlos en el programa.

Una vez se han asociado todas las huellas a sus componentes correctamente, se debe pasar a escoger el tamaño de la placa. Como esta tenía que ser un shield para Arduino se trató de no sobrepasar en gran medida las dimensiones de esta, tratando de conseguir un diseño más compacto.

Tras escoger el tamaño de la placa se pasó a distribuir los componentes en la misma teniendo en cuenta ciertos criterios en la medida de lo posible:

- Pines compartidos con Arduino en la posición exacta que tienen en dicha placa.
- Distancia mínima de borde de placa a componente o pista 2mm.
- Evitar una concentración demasiado elevada de taladros en cada zona.
- Colocar los componentes con mayores conexiones lo más cerca posible entre sí.
- Condensadores de desacoplo lo más cerca posible de los pines de alimentación y tierra del circuito integrado al que están asociados.
- Conector de alimentación en el borde de la placa para facilitar su conexión.
- Conectores de los motores lo más accesibles posible.

Después de colocar las huellas de los componentes siguiendo estas reglas se pasa al trazado de las pistas, usando como ayuda las conexiones 'virtuales' antes mencionadas, proporcionadas por el programa. Como se comentó al inicio, en este caso se optó por realizar una PCB bicapa, pudiendo emplear tanto la capa de soldadura como la capa de componentes para trazar pistas. Para el trazado de pistas también se siguieron una serie de reglas de diseño:

- Capas posibles para el trazado de pistas: BOTTOM y TOP
- Separación mínima entre pistas de 0,5 mm.
- Separación mínima entre pista y pad 0,254 mm.
- Hacer las pistas lo más anchas posibles donde sea posible (hasta 1 mm de ancho).
- Anchura mínima de pistas de 0,4 mm.

- Pistas con el menor recorrido posible.
- Ángulos máximos de 45º en las pistas.

Una vez finalizado el ruteo de la totalidad de las pistas siguiendo las reglas descritas anteriormente, el siguiente paso consiste en modificar el tamaño de los pads y taladros para adaptarlos a las características de la máquina encargada de la mecanización de la placa. Los tamaños empleados en los pads y taladros fueron los siguientes:

Componentes	Diámetro Pads	Diámetro taladros
Conector de alimentación	2	1,3
Pines de conexión con Arduino	1,2	1
Circuitos integrados	1,2	1
Condensadores de gran capacidad	1,2	1
Resto de componentes	1	0,8

Figura 21. Tabla de tamaño de pads y taladros de la placa PCB

Con el ajuste del tamaño de los pads y taladros solo queda generar los ficheros necesarios para el mecanizado de la placa. Mediante el software Layout es posible generar varios tipos de ficheros para la creación de placas. En nuestro caso se generaron los ficheros GERBER de las capas necesarias para la fabricación. Estos ficheros incluyeron: el trazado de pistas de las capas TOP y BOT, la máscara de soldadura de TOP y BOT, la serigrafía de TOP, y el fichero de taladros.

A continuación se muestra una imagen obtenida del programa, donde se pueden apreciar las pistas de ambas capas (rojas para la capa BOT y azules para la capa TOP), la colocación de los componentes, la serigrafía y los pads y taladros:

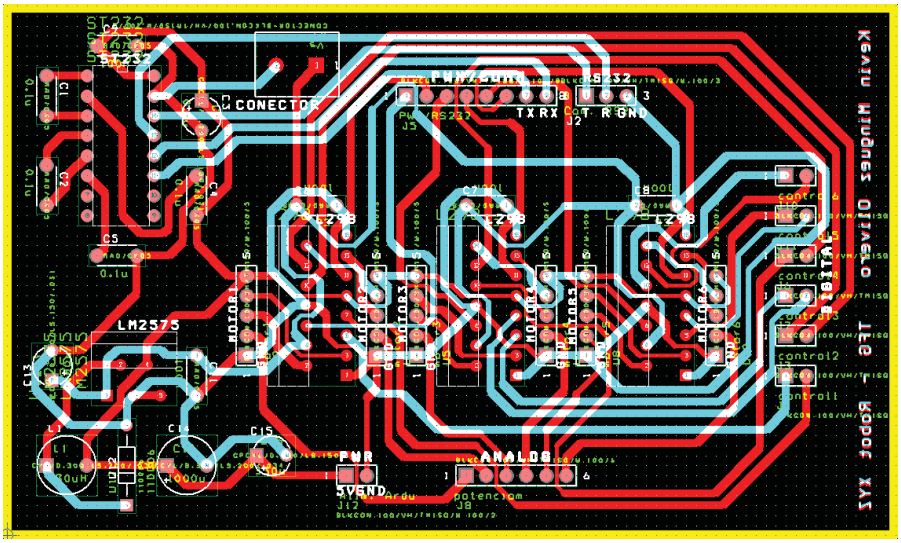


Figura 22. Diseño físico de la placa PCB

3.3 Fabricación de la placa PCB

La fabricación de la placa ha sido llevada a cabo por el Servicio de Electrónica de la Universidad de La Laguna. El proceso de fabricación de la placa ha sido mediante mecanizado con impresora-fresadora. Estas máquinas permiten el trazado de las pistas, el taladrado y contorneado de la placa tras suministrar a la misma los ficheros de postprocesado antes mencionados.

En el proceso de fabricación se aplicaron, también, las capas de máscara de soldadura y el metalizado de los taladros, imprescindible para el correcto contacto de los pads entre las capas BOT y TOP.

Tras recibir la placa terminada desde el Servicio de Electrónica se procedió al montaje y soldadura manual de los componentes. Como ya se había comentado, los componentes utilizados fueron de tipo THD o pasante. Una cuestión a nombrar es que la mayoría de los componentes fueron colocados sobre la capa TOP (con las patillas en la capa BOT) pero los pines de conexión con la Arduino tuvieron que ser soldados de manera inversa para facilitar el encaje entre ambas placas.

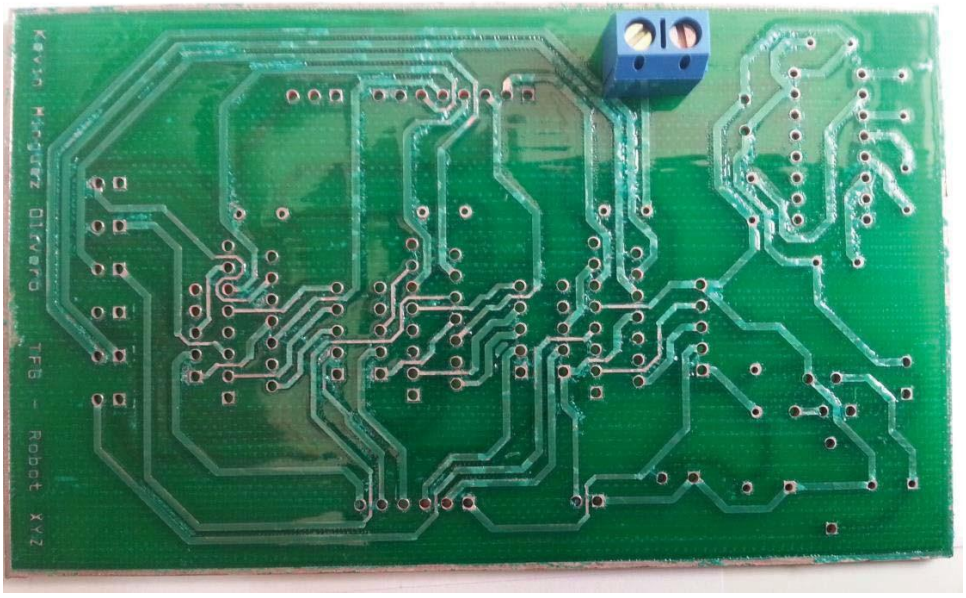


Figura 23. Placa PCB sin componentes (capa BOT)

[En la imagen el conector está soldado a la inversa. Esto se corrigió posteriormente]

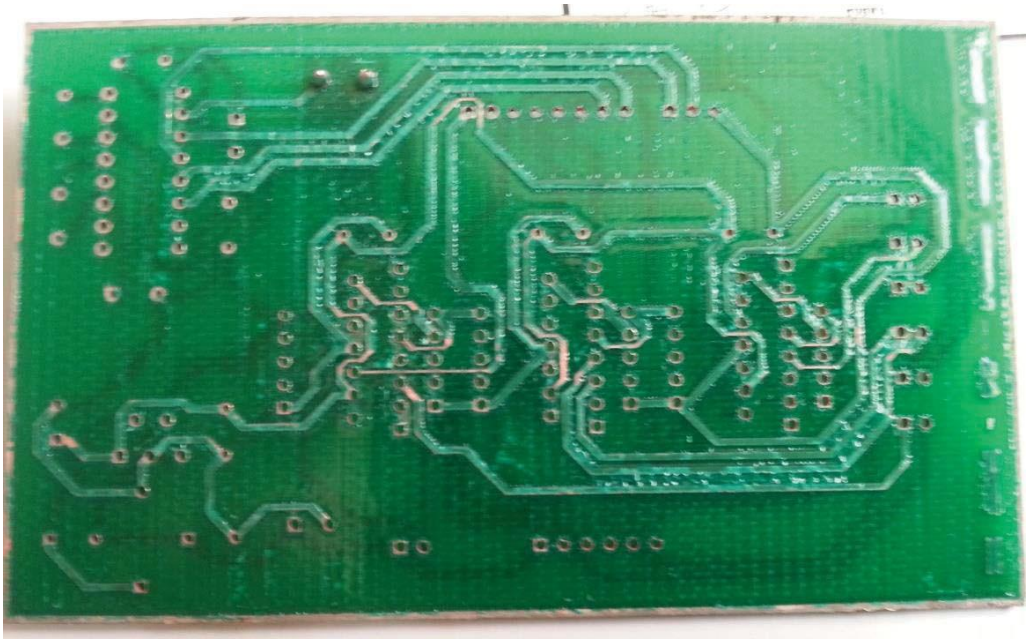


Figura 24. Placa PCB sin componentes (capa TOP)

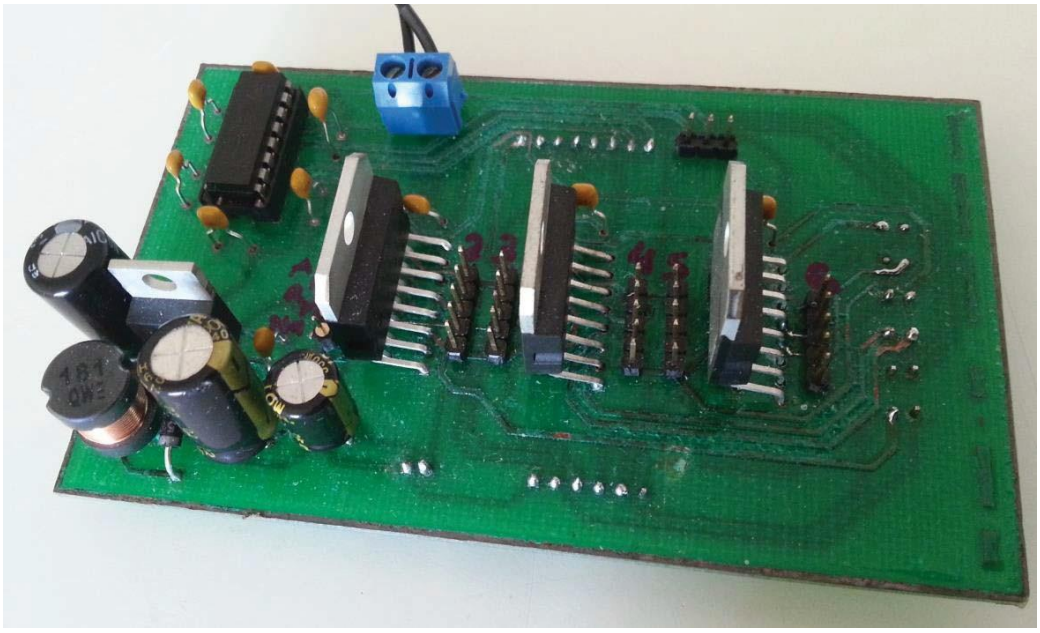


Figura 25. Placa PCB final (capa TOP)

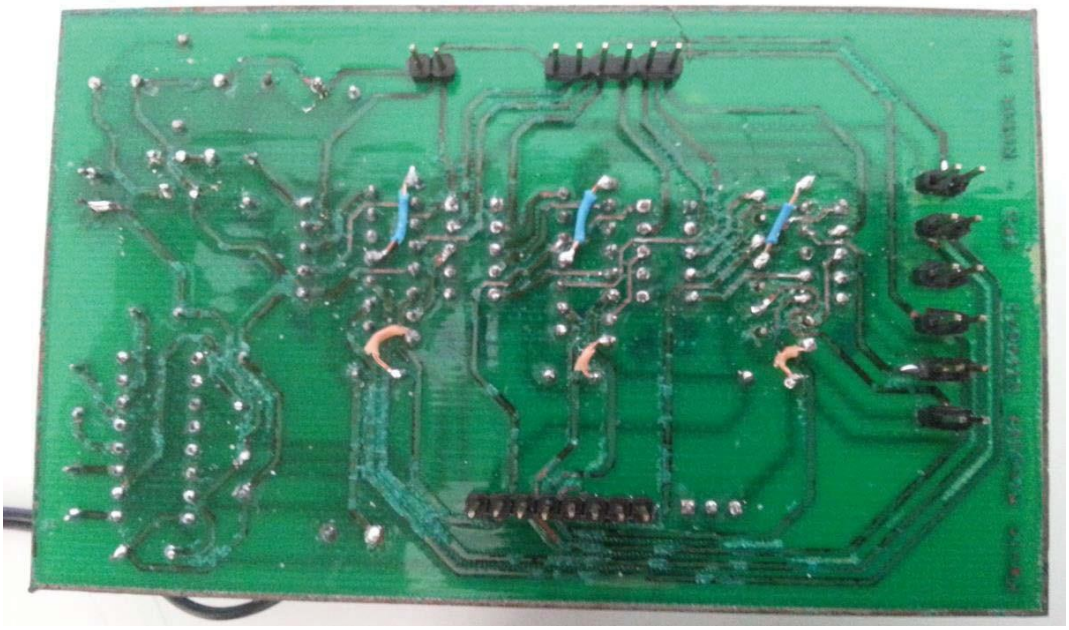


Figura 26. Placa PCB final (capa BOT)

Como podemos ver en la imagen anterior, en la placa fabricada se han empleado conexiones extra entre algunos pines. Esto fue resultado de un fallo en el diseño esquemático al no realizar estas conexiones. Tras comprobar el error, se pasó a realizar las conexiones faltantes e inmediatamente fue modificado tanto el diseño esquemático como el diseño físico de la placa para evitar posteriores reproducciones erróneas de la placa. A pesar de esto, no fue posible fabricar la placa nuevamente por razones de tiempo. Remarcar que los diseños expuestos al final del apartado 3.1 y al final del apartado 3.2 son los corregidos.

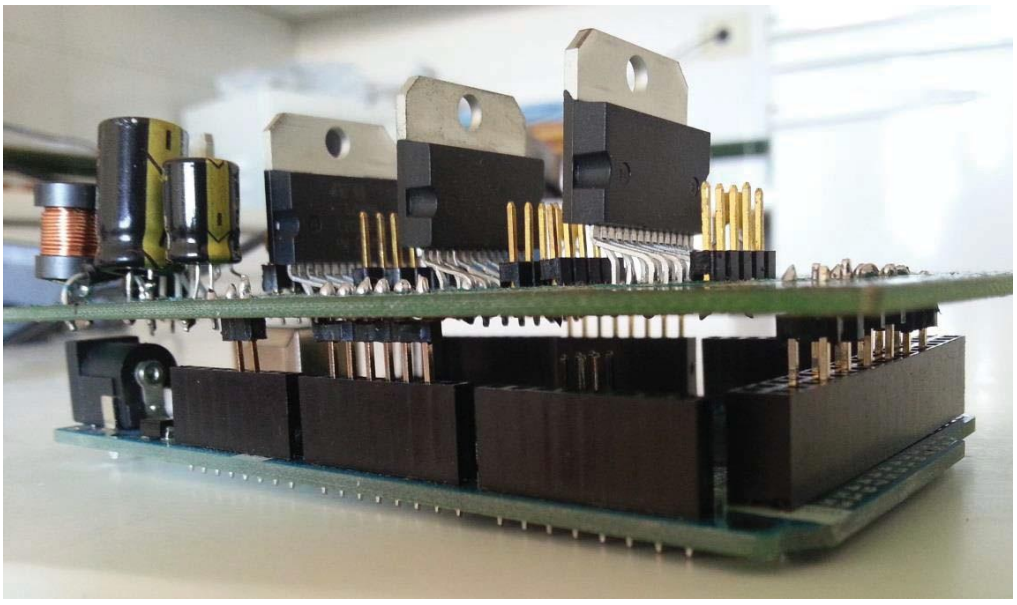


Figura 27. Comprobación de la correcta colocación de los pines de conexión con Arduino

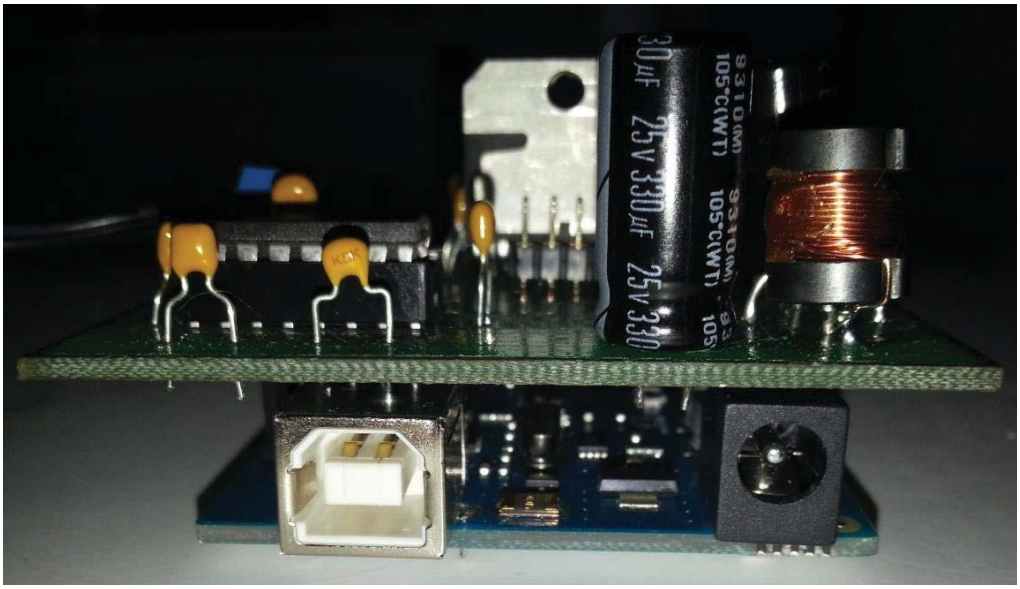


Figura 28. Sistema ensamblado de placa PCB y Arduino

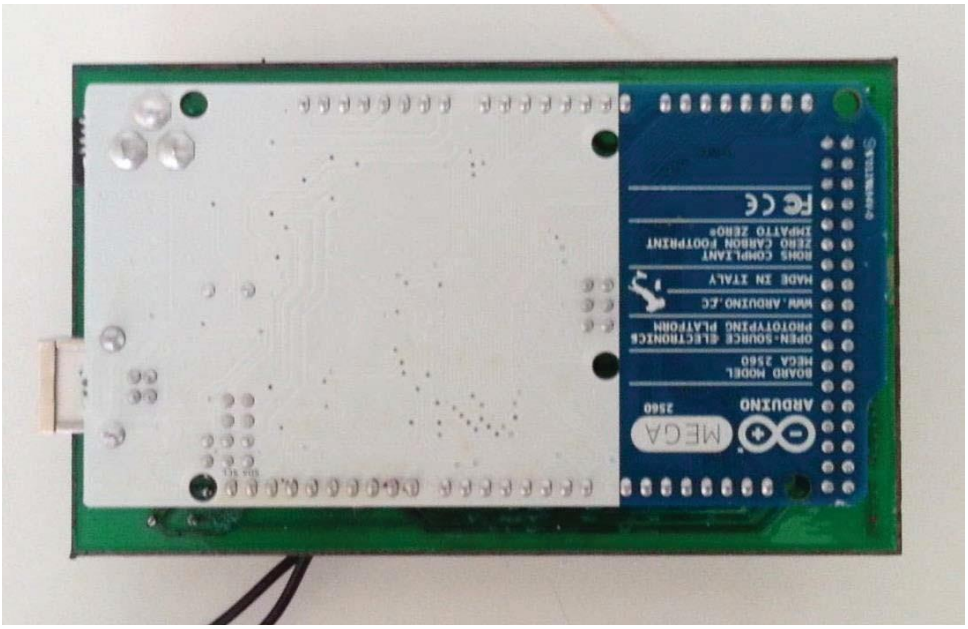


Figura 29. Comparación de tamaño entre placa PCB y Arduino

4. Espacio de Trabajo

4.1 Introducción

4.1.1 Definición

"*El espacio de trabajo de un robot está definido como el grupo de puntos que pueden ser alcanzados por su efector-final.*"

Dicho de otro modo, el espacio de trabajo de un robot es **el espacio en el cual el mecanismo puede trabajar** (simple y llanamente). A pesar de que esta definición está muy extendida, diversos autores también se refieren al espacio de trabajo como volumen de trabajo y envolvente de trabajo.

4.1.2 Principales características de un espacio de trabajo

Cuando se pretende estudiar un espacio de trabajo, lo más importante es su forma y volumen (dimensiones y estructura). Ambos aspectos tienen una importancia significativa debido al impacto que éstos ejercen en el diseño del robot y también en su manipulabilidad.
Si se pretende utilizar un robot, el exacto conocimiento sobre la forma, dimensiones y estructura de su espacio de trabajo es esencial puesto que:

- La forma es importante para la definición del entorno donde el robot trabajará.
- Las dimensiones son importantes para la determinación del alcance del efector-final.
- La estructura del espacio de trabajo es importante para asegurar las características cinemáticas del robot las cuales están relacionadas con la interacción entre el robot y el entorno.

Además, la forma, dimensiones y estructura del espacio de trabajo dependen de las propiedades del robot en cuestión:

- Las dimensiones de los eslabones del robot y las limitaciones mecánicas de las articulaciones (tanto pasivas como activas) tienen una gran influencia en las dimensiones del espacio de trabajo.
- La forma depende de la estructura geométrica del robot (interferencia entre eslabones) y también de las propiedades de los grados de libertad (cantidad, tipo y límites de las articulaciones, tanto pasivas como activas).
- La estructura del espacio de trabajo viene definida por la estructura del robot y las dimensiones de sus eslabones.

4.1.3 Ventajas e inconvenientes

La visualización del espacio de trabajo de un robot tiene ventajas y desventajas:

Ventajas	Inconvenientes
La representación tridimensional del espacio de trabajo facilita la visión de la forma del mismo.	Es necesario procesar una gran cantidad de puntos para un análisis exacto.
Proporciona una idea de cómo es el espacio de trabajo.	Deben tenerse en cuenta diferentes criterios de análisis estructural del espacio de trabajo con respecto a la manipulabilidad.
Ofrece la posibilidad de optimizar las características del robot.	Puesto que los robots con diferentes estructuras tienen diferentes espacios de trabajo, debe analizarse cada uno por separado.

Figura 30. Tabla de Ventajas e inconvenientes del espacio de trabajo

4.2 Cálculo del Espacio de Trabajo

4.2.1 Puntos Límite

El primer paso para el cálculo de nuestro espacio de trabajo es situar los puntos límite, se diseñó un código en Matlab que realiza dicha función y será explicado a continuación. Dicho código está dividido en cuatro partes.

La primera, está compuesta por la declaración de las variables A y B, que son las matrices de coordenadas de la base móvil y fija respectivamente. A continuación se pide al usuario que introduzca por teclado el valor inicial y final de cada variable para realizar el barrido, se puede observar la introducción de una variable, N, la cual delimitará el "step" de los bucles "for" en la concatenación de las variables Pitch, Roll y Yaw para la generación de posiciones. Se explicará la implementación de dicha variable en el apartado de inconvenientes.

```
%Matrices de coordenadas de la base móvil y fija
respectivamente.
A = single([5 5.8 0.7 -0.4 -5.2 -4.3;3.6 2.7 -5.6 -5.8 3.4 3.9;
            0 0 0 0 0 0]);
B = single([1.2 8.8 7.3 -7.3 -8.8 -1.2; 9.3 -4.6 -6.5 -6.5 -4.6 9.3;
            0 0 0 0 0 0]);

%Introducción de los valores por teclado.
xpi    = input('Ingrese el valor inicial de X: ');
xpf    = input('Ingrese el valor final de X: ');
ypi    = input('Ingrese el valor inicial de Y: ');
ypf    = input('Ingrese el valor final de Y: ');
zpi    = input('Ingrese el valor inicial de Z: ');
zpf    = input('Ingrese el valor final de Z: ');
ppi    = input('Ingrese el valor inicial de PITCH: ');
ppf    = input('Ingrese el valor final de PITCH: ');
rpi    = input('Ingrese el valor inicial de ROLL: ');
rpf    = input('Ingrese el valor final de ROLL: ');
wpi    = input('Ingrese el valor inicial de YAW: ');
wpf    = input('Ingrese el valor final de YAW: ');
N      = 7;

%Declaración de variables, D1, D2 y D3 son los step de los for de Pitch,
%Roll y Yaw.
D1 = ((abs(ppi)+ppf)/N);
D2 = ((abs(rpi)+rpf)/N);
D3 = ((abs(wpi)+wpf)/N);

%Inicialización de las variables para realizar la concatenación.
a=single([]);
b=single([]);
c=single([]);
d=single([]);
e=single([]);
f=single([]);
```

La otra mitad de la primera parte se encarga de la concatenación de las variables para generar la matriz P, que contendrá todas las posiciones que más adelante analizaremos para saber cuáles son aceptadas. Los bucles "for" iteran desde el valor inferior hasta el superior en "steps" de 0.5 salvo las orientaciones que iteran en "steps" de 8.

```
%Parte que se encarga de concatenar los valores introducidos por
teclado.
for i=xpi:0.5:xpf
    for j=ypi:0.5:ypf
         for k=zpi:0.5:zpf
            for l=ppi:D1:ppf
                for m=rpi:D2:rpf
                    for n=wpi:D3:wpf

                        a = [a i ];
                        b = [b j ];
                        c = [c k ];
                        d = [d l ];
                        e = [e m ];
                        f = [f n ];

                        P=single([a' b' c' d' e' f']);

                    end
                end
             end
        end
    end
end

%fip, nº de filas, cop, nº de columnas de la matriz P.
[fip,cop] = size(P);
```

La segunda parte del código es la cinemática inversa del robot, efectuado por un programa realizado para un proyecto anterior de la Universidad de la Laguna. A continuación se presentan los datos del proyecto en cuestión

- Título: Diseño e Implementación de un Sistema de Estabilizacion Activa para el Sistema de Cámaras del Prototipo Verdino.
- Titulación: Ingeniería en Automática y Electrónica Industrial
- Autores: Antonio Luis Morell González e Iván Daniel Peraza Rocha.
- Fecha: Julio, 2010.

Ésta se encarga de ejecutar los cálculos pertinentes utilizando en cada iteración la fila i-ésima de la matriz P como el vector pose, donde fip es el número de filas de la matriz P. La matriz RR es la matriz de rotación y finalmente la matriz L1 es el resultado final con las longitudes que deberán tomar los 6 motores de la plataforma.

```
L1=single([]);
%Cinemática inversa.

for i=1:fip

    pose = P(i,:)';
    A2 = A(1:3,:);
    B2 = B(1:3,:);
    R = pose(4:6,:);
    D = pose(1:3,:);
    Rrad = (R.*pi)./180;
    cphi    = cos(Rrad(1));
    ctita = cos(Rrad(2));
    cpsi    = cos(Rrad(3));
    sphi    = sin(Rrad(1));
    stita = sin(Rrad(2));
    spsi    = sin(Rrad(3));
    sphi_stita = sphi*stita;
    cphi_stita = cphi*stita;

     RR=[ctita*cpsi+sphi_stita*spsi -ctita*spsi+sphi_stita*cpsi cphi_stita ;
        cphi*spsi                    cphi*cpsi                  -sphi      ;
        -stita*cpsi+sphi*ctita*spsi  stita*spsi+sphi*ctita*cpsi cphi*ctita];

     L1 = [L1 sqrt(sum((RR * A2 + D(:,ones(6,1)) - B2).^2))'];

end

L2 = L1';
```

A continuación se convierten los valores a un rango entre 0 y 1023, para, posteriormente pasar por el bucle de seguridad. El primer paso es dividir la matriz P en celdas, una vez están creadas las celdas, éstas pasan por el segundo bucle que convierte sus valores, para, finalmente pasar por el bucle de seguridad. Éste último se encarga de ir aumentando el valor del vector temp si existe alguna variable de cada celda que excede los valores permitidos.

```
AA=single([]);
LL=single([]);

%Separación de las poses en celdas para ser evaluadas.
for i=1:fip
    AA{i} = L2(i,:);
    LL{i} = P(i,:);

end
%Conversión de la pose.
for i = 1:fip
    for j=1:6
        AA{i}(j) = AA{i}(j)-17.5; AA{i}(j) =

        1024*AA{i}(j)/5;

    end
end


temp = zeros(fip,6); w = 0;

%Filtro, sólo la pose con temp = 0 es aceptada.
```

```
for i=1:fip
    for j = 1:6
        if (AA{i}(j)<0 || AA{i}(j)>1023)
            temp(i,j)=temp(i,j)+1;
        end
    end
end
```

La última parte del código es la encargada de proporcionar los valores límites. Mediante otro bucle "for" se comprueba que la suma de cada vector temp sea igual a cero, si esto es así, la celda correspondiente se concatena en la matriz RTY. Una vez hecho esto sólo queda buscar el valor máximo y mínimo de cada columna para obtener los puntos límite.

```
RTY=single([]);

%Si la suma del vector temp = 0, la pose es aceptada.
for rty=1:fip
    if sum(temp(rty,:)) == 0
        RTY = [RTY LL{rty}' ];
    end
end

M=RTY';

%Valores máximos y mínimos de cada variable de pose.
minX = min(M(:,1));
maxX = max(M(:,1));

minY = min(M(:,2));

maxY = max(M(:,2));


minZ = min(M(:,3));

maxZ = max(M(:,3));


minP = min(M(:,4));

maxP = max(M(:,4));


minR = min(M(:,5));

maxR = max(M(:,5));


minYW = min(M(:,6));

maxYW = max(M(:,6));
```

El primer paso para hallar los puntos límite fue buscar las posiciones extremas XYZ. Se introdujeron los valores de la tabla y se obtuvieron los siguientes resultados.

	X	Y	Z
Valor Mínimo	-10	-10	15
Valor Máximo	10	10	22
Límite Inferior	-7.5	-8	16.5
Límite Superior	7	5.5	21

Figura 31. Tabla Valores Límite XYZ

Una vez encontrados los puntos límite XYZ se realizó la búsqueda de los valores límite de cada ángulo por separado y se obtuvieron los siguientes resultados.

	Pitch	Roll	Yaw
Valor Mínimo	-90	-90	-90
Valor Máximo	90	90	90
Límite Inferior	-35	-30	-83
Límite Superior	30	27	87

Figura 32. Tabla Valores Límite Pitch, Roll y Yaw

4.3 Espacio Total de Trabajo

Una vez hallados los puntos límites, el siguiente paso fue constituir el espacio de trabajo completo. Utilizamos el código mencionado anteriormente junto con cuatro partes más. Esta vez los datos introducidos por teclado fueron los valores límite de cada variable, pero limitando los valores de Z por la cantidad de puntos generados, es decir, fuimos estudiando el espacio de trabajo por planos, para un mismo valor de Z exploramos el resto de variables.

La quinta parte del código se encarga de dividir en submatrices la matriz M, que contiene todas las posiciones aceptadas por la plataforma. Debido a que para un mismo punto XYZ existen multitud de combinaciones de orientación la función de este código es seleccionar el valor máximo que puede alcanzar cada orientación en ese punto en concreto, esto no quiere decir que la posición devuelta sea una posición alcanzable por la plataforma, pero proporciona una noción de las orientaciones en el espacio de trabajo. Se ha realizado de esta manera porque si se graficara cada valor de orientación en un mismo punto existiría una superposición de puntos y el resultado final sería el último punto graficado.

De esta manera, la matriz M va iterando a través de los tres bucles “for” y se subdivide, como se puede observar, la primera submatriz devuelta será la compuesta por:

X = -7.5

Y = -8

Z = 16.5

Ahora bien, para esa submatriz, el código se encarga de encontrar y devolver el máximo valor de Pitch, Roll y Yaw.

```
SM4=single([]);

%Código que separa la matriz de poses aceptadas en submatrices con
un mismo valor para cada columna.

   for i2=minX:0.5:maxX
       for j2=minY:0.5:maxY
           for k2 =minZ:0.5:maxZ

               [fi,co]=find(M(:,1)== i2);

                fimx = max(fi);

                fimn = min(fi);

                comx = max(co);

                comn = min(co);

                SM = M(fimn:fimx,:);
```

```
                [fi2,co2]=find(SM(:,2)== j2);

                fimx2 = max(fi2);

                fimn2 = min(fi2);

                comx2 = max(co2);

                comn2 = min(co2);

                SM2 = SM(fimn2:fimx2,:);


                [fi3,co3]=find(SM2(:,3)== k2);

                fimx3 = max(fi3);

                fimn3 = min(fi3);

                comx3 = max(co3);

                comn3 = min(co3);

                SM3 = SM2(fimn3:fimx3,:);

                [tt,yy] = size(SM3);

                %Llegados a este punto podemos encontrarnos que la
                %submatriz SM3 esté formada por una sola fila, de
                %ser así se concatena esa única fila. Por el
                %contrario, el proceso se ejecuta como antes,
                %buscando el máximo valor para cada %ángulo.

                if tt == 1
                    SM4 = [SM4 SM3'];

                else
                    SM4 = [SM4 max(SM3)'];

                end
            end
        end
    end

YRT = SM4';
```

Una vez obtenido el resultado anterior se generan las coordenadas XYZ para obtener la gráfica. Dado que, como se ha explicado anteriormente, se decidió realizar el estudio por planos, las variables del código que se expone a continuación varían desde 1 hasta 10. La primera parte del bucle se encarga de crear las matrices XYZ y la segunda de asignar el color correspondiente en el espacio RGB dependiendo el valor del ángulo correspondiente, esto es, rojo para Pitch, verde para Roll y azul para Yaw.

Para determinar la variación del color de cada posición se hace uso de la fórmula abajo expuesta, la cual devuelve un valor entre 0, para el máximo valor del ángulo y 1 para el mínimo.

```
[pt1,pt2] = size(SM4');

X1=[];
Y1=[];
Z1=[];
RJ1=[];
VR1=[];
AZ1=[];

%De 1 a 10, se crean los puntos XYZ y los colores de P,R,Y para
la gráfica.
for l=1:pt1

    X1 = [X1 YRT(l,1)];
    Y1 = [Y1 YRT(l,2)];
    Z1 = [Z1 YRT(l,3)];

   RJ1 = [RJ1 ((max(abs(P(:,4))-abs(YRT(l,4))))/max(abs(P(:,4))))];
   VR1 = [VR1 ((max(abs(P(:,5))-abs(YRT(l,5))))/max(abs(P(:,5))))];
   AZ1 = [AZ1 ((max(abs(P(:,6))-abs(YRT(l,6))))/max(abs(P(:,6))))];

end
```

Una vez se han obtenido los resultados de los 10 planos se concatenan los resultados. XF, YF y ZF para las coordenadas de cada punto y RJF, VRF y AZF para los colores.

```
%Se concatenan las 10 matrices creadas anteriormente, es el paso
previo %a graficar.
XF1 = [X1 X2 X3 X4 X5 X6 X7 X8 X9 X10];
YF1 = [Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8 Y9 Y10];
ZF1 = [Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10];

RJF1 = [RJ1 RJ2 RJ3 RJ4 RJ5 RJ6 RJ7 RJ8 RJ9 RJ10];
VRF1 = [VR1 VR2 VR3 VR4 VR5 VR6 VR7 VR8 VR9 VR10];
AZF1 = [AZ1 AZ2 AZ3 AZ4 AZ5 AZ6 AZ7 AZ8 AZ9 AZ10];
```

Por último, se grafica el resultado. Utilizamos un bucle "for" que itera desde 1 hasta pt2, que es el número de columnas de la matriz XF1. A continuación se detallan los parámetros del "plot":

·mo: especificación de marca tipo circulo.

·MarkerEdgeColor 'k': color negro para el borde de la marca.

·MarkerFaceColor [ ]: color de relleno de la marca, es una matriz formada por valores entre 0 y 1 en el espacio de color RGB.

·LineWidth: ancho de la línea del borde.

·MarkerSize: tamaño de la marca.

·label: etiquetas de los ejes.

·axis [ ]: límites de los ejes.

```
[pt1,pt2] = size(XF1);

figure;

%Código que genera la gráfica.
for i=1:pt2
    plot3(XF1(i),YF1(i),ZF1(i),'mo','MarkerEdgeColor','k','MarkerFaceColor',...
         [RJF1(i) VRF1(i) AZF1(i)],'LineWidth',1,'MarkerSize',10);

    hold on;
    xlabel('Eje X');
    ylabel('Eje Y');
    zlabel('Eje Z');
    axis([-8,7,-8,6,15.5,22])
end
```

4.4 Resultados

Se ha de destacar que al no ser valores simétricos se decidió realizar dos gráficas. Una con los valores positivos de Pitch, Roll y Yaw y otra con los valores negativos.

4.3.1 Valores Positivos

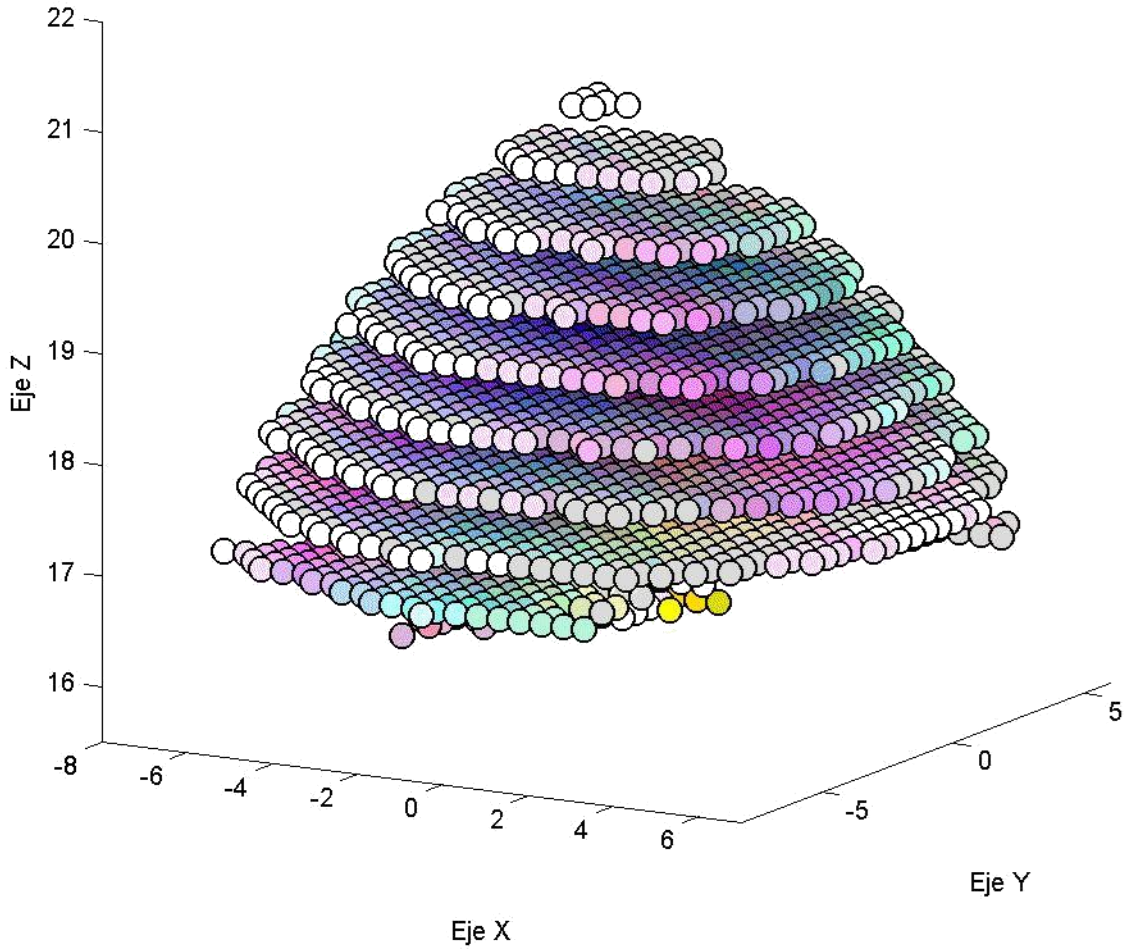


Figura 33. Gráfica Valores Positivos

Por cada plano se generan 430.080 puntos, lo que hace un total de 4.300.800 puntos, de los cuales sólo 3111 componen la gráfica.

Se observa como la gráfica tiene una forma piramidal, esto es así ya que los motores van limitado el movimiento de la base conforme aumentan su longitud. Vemos como en la base se alcanzan los puntos límite en el eje X e Y, y en lo alto de la pirámide el número de puntos se ve reducido considerablemente.

En cuanto a los colores, los más claros representan los valores mínimos y los más oscuros los máximos, así, podemos observar cómo en las posiciones más extremas existe una carencia de color, lo que significa una orientación nula del efector final y conforme nos dirigimos a las posiciones centrales los colores se tornan más oscuros, lo que nos indica que existen múltiples configuraciones posibles de la orientación del efector final.

4.3.2 Valores Negativos

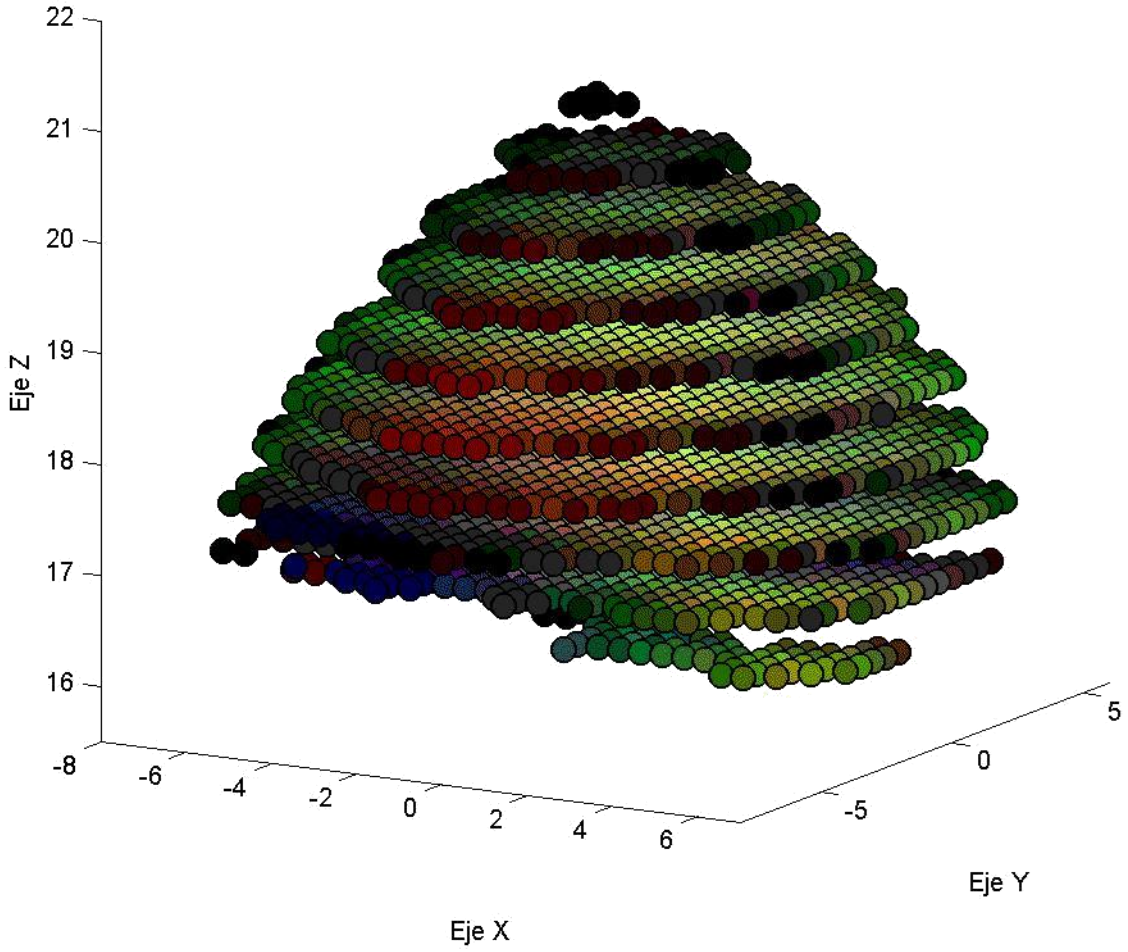


Figura 34. Gráfica Valores Negativos

Al igual que la gráfica con los valores positivos, ésta presenta forma piramidal, debido a la limitación de los motores conforme aumentan su longitud.

A diferencia del la gráfica de valores positivos, ésta se compone de 3234 puntos.

En cuanto a los colores, se ha hecho de forma inversa que en el caso anterior, los colores oscuros representan el mínimo y los más claros el máximo. Siendo los extremos las zonas más limitadas y las zonas centrales las que presentan múltiples configuraciones.

4.5 Inconvenientes

El mayor inconveniente que hemos tenido en esta parte ha sido la limitación propia del software para generar la mayor cantidad de puntos posibles.

En el apartado anterior se menciona el uso de la variable N para acotar los "steps" de los bucles "for" a la hora de generar las posiciones, esto se hizo de esa manera porque si realizábamos el barrido completo, esto es, grado a grado, se generarían 64.162.560 puntos, algo imposible de realizar.

Por otro lado, la primera vez que se realizó la gráfica, ésta se componía de unos 300 puntos, diez veces menos que los generados mediante el método de ir plano a plano.

Por último, y debido a la asimetría de los valores límite de los ángulos de orientación se decidió realizar dos gráficas, una para valores positivos y otra para valores negativos, algo que aporta un espectro mayor de colores al reducir el barrido a la mitad.

5. Simulación por ordenador

En este capítulo se explicará el código utilizado para la simulación del movimiento de la plataforma digitalmente. Se utilizó el software informático Matlab para generar dicho código.

El código se compone básicamente de tres partes. La primera es la declaración de variables e introducción de la posición deseada mediante teclado, luego se comprueba, mediante la cinemática inversa que dicha posición es alcanzable y, por último, una vez aceptada la posición, se realizan los cálculos para la simulación.

5.1 Inicialización

En primer lugar se inicializa la variable pose, que es el vector de posición final que tomará la base móvil. Está compuesto por las variables XYZ y los ángulos de rotación Pitch, Roll y Yaw. Por último se inicializa también la variable MAX, su uso será explicado más adelante.

```
clear all;
clc;

%Inicialización
pose = [0 0 0 0 0 0]';
MAX = 1;
```

Las matrices A y B son las coordenadas de los vértices de la base móvil y fija respectivamente. X0, Y0 y Z0 son las matrices de coordenadas de la base fija para poder graficarla.

A3 y B3 son las matrices de coordenadas de los anclajes de los motores entre la base móvil y fija. Ha de aclararse que las matrices A y B se utilizan para graficar la simulación y A3 y B3 para realizar los cálculos cinemáticos de la plataforma.

XX, YY, ZZ, PP, ROO e YWW son las componentes del vector de posición pose3 que será analizado posteriormente.

Por último, se pide al usuario que introduzca por teclado la posición que se desea y se realizan la declaración de la variable last_pose y la concatenación de las variables introducidas por teclado en el vector pose2.

```
%Matrices de coordenadas de los vértices de la base móvil y
%de la base fija, respectivamente.
A = [-1.5 -6.5 -5 5 6.5 1.5; -5.628 3.03 5.628 5.628 3.03 -5.628;
     0 0 0 0 0 0;1 1 1 1 1 1];
B = [-8.5 -11 -2.5 2.5 11 8.5;-11.69 -7.36 7.362 7.362 -7.36 -11.69;
    0 0 0 0 0 0;1 1 1 1 1 1];

X0 = [B(1,1) B(1,2) B(1,3) B(1,4) B(1,5) B(1,6) B(1,1)];
Y0 = [B(2,1) B(2,2) B(2,3) B(2,4) B(2,5) B(2,6) B(2,1)];
Z0 = [B(3,1) B(3,2) B(3,3) B(3,4) B(3,5) B(3,6) B(3,1)];
```

```
%Coordenadas de los anclajes entre la base móvil y la base fija.
A3 = [5 5.8 0.7 -0.4 -5.2 -4.3;3.6 2.7 -5.6 -5.8 3.4 3.9;
      0 0 0 0 0 0];
B3 = [1.2 8.8 7.3 -7.3 -8.8 -1.2; 9.3 -4.6 -6.5 -6.5 -4.6 9.3;
      0 0 0 0 0 0];

%Inicialización de las variables que conforman la matriz pose3.
XX=[];
YY=[];
ZZ=[];
PP=[];
ROO=[];
YWW=[];

%Introducción por teclado de la posición deseada.
fprintf('\n');
X = input('Introduce el Valor de X: ');
Y = input('Introduce el Valor de Y: ');
Z = input('Introduce el Valor de Z: ');
PI = input('Introduce el Valor de PITCH: ');
RO = input('Introduce el Valor de ROLL: ');
YW = input('Introduce el Valor de YAW: ');
fprintf('\n');

%Variable que propicia el movimiento desde la última posición dada.
%pose2, concatenación de las variables de posición.
last_pose = pose;
pose2 = [ X Y Z PI RO YW ]';
```

5.2 Comprobación de posición

En este fragmento del código se observa la cinemática inversa del robot, la cual se encarga mediante la matriz de rotación RR y los cálculos realizados en L1 de devolver las longitudes finales que cada motor ha de alcanzar para la posición requerida.

El bucle “for” que se encarga de transformar los valores de la variable L1 en valores entre 0 y 1023 para luego pasar por el bucle de seguridad que aumenta el valor de la variable variable temp si existe alguna componente de L1 que excede los valores permitidos.

```
%Cinemática inversa.
A3 = A3(1:3,:);
B3 = B3(1:3,:);
R = pose2(4:6,:);
D = pose2(1:3,:);
Rrad = (R.*pi)./180;
cphi = cos(Rrad(1));
```

```
ctita = cos(Rrad(2));
cpsi = cos(Rrad(3));
sphi = sin(Rrad(1));
stita = sin(Rrad(2));
spsi = sin(Rrad(3));
sphi_stita = phi*stita;
cphi_stita = phi*stita;
RR=[ctita*cpsi+sphi_stita*spsi,-ctita*spsi+sphi_stita*cpsi, cphi_stita ;
    cphi*spsi,                 cphi*cpsi,                  -sphi;
    -stita*cpsi+sphi*ctita*spsi,stita*spsi+sphi*ctita*cpsi, cphi*ctita];

L1 = sqrt(sum((RR * A3 + D(:,ones(6,1)) - B3).^2));
%Conversión de los valores entre 0 y 1023.
for i = 1:6

    L1(i) = L1(i)-17.5;
    L1(i) = 1024*L1(i)/5;

end
temp = 0;
%Bucle de seguridad.
for i = 1:6

    if (L1(i)<0 || L1(i)>1023)
    temp=temp+1;
    end

end
```

5.3 Generación de secuencias

Una vez la posición es aceptada hay que contemplar dos casos, cuando MAX = 0 y cuando MAX =! 0.

En el primer caso hay que evaluar cada variable con su valor anterior y comprobar si ha variado. Si no es así, se genera un vector fila con n columnas, donde n es el valor de last_MAX, que multiplica al valor actual de la variable. En el caso de que el valor de la variable sea diferente a la posición anterior se genera un vector fila mediante un bucle "for" que itera desde la posición anterior hasta la actual en "steps" resultantes de hallar la diferencia de posición partido por el valor de la variable last_MAX. Esto genera el mismo número de "stepts" para las seis variables y garantiza el movimiento simultáneo de la plataforma.

```
%Si temp == 0 la posición es aceptada.
if temp == 0
    pose = pose2;
    fprintf('Vector de Posición aceptado \n')
    pose(3)=pose(3)-17;

    last_MAX = MAX;
    MAX=max(abs(pose(:)));

%Si MAX == 0 se utiliza la variable last_MAX para el tamaño
%de la matriz unitaria o para los steps del bucle for.

    if MAX == 0

%Si la última posición coincide con la actual se genera una
%matriz con el mismo valor, ones*pose(i).

        if last_pose(1) == pose(1)

            XX = ones(1,last_MAX+1)*pose(1);

        else

%Si la posición es diferente a la anterior, se crea una matriz
que itera desde la posición anterior hasta la actual.

            for i=last_pose(1):((pose(1)- last_pose(1))/last_MAX):pose(1)

                XX = [XX i];

            end

        end
```

En el caso de que MAX =! 0, se procede de la misma forma que antes, salvo que ahora el tamaño del vector fila depende de la variable MAX y la diferencia entre la posición anterior y actual, para la iteración del bucle for, en el caso de que la posición actual difiera de la anterior es partido por la variable MAX.

Se ha de aclarar que tanto en el código anterior como en este se hace uso de los parámetros MAX+1 o last_MAX+1 ya que la indexación de las matrices comienzan por el valor 1.

```
%Si MAX =! 0, se utiliza la variable MAX para el tamaño que la
matriz unitaria o para los steps del bucle for.
    else

        if last_pose(1) == pose(1)

            XX = ones(1,MAX+1)*pose(1);

        else

            for i=last_pose(1):((pose(1)- last_pose(1))/MAX):pose(1)

                XX = [XX i];
            end

        end
```

Una vez generados los vectores de posición, éstos se concatenan en la matriz pose3.

A continuación, nos encontramos otra vez en la situación de contemplar el caso en el que MAX = 0 ó MAX =! 0.

Si MAX = 0, mediante un bucle “for” iteramos desde 1 hasta last_MAX+1 la fila i-ésima de la traspuesta de pose4, calculamos la cinemática inversa mediante la multiplicación de la matriz de rotación RR2 por la matriz de coordenadas de la base móvil A y el resultado es guardado en la variable A2.

De esta forma, el código irá graficando todas las posiciones intermedias entre la posición anterior y la actual.

```
%Se concatenan los resultados en pose3 para realizar los cálculos
%de la simulación.
pose3 = [XX' YY' ZZ' PP' ROO' YWW'];

%En el caso de que MAX == 0, se utiliza last_MAX para la iteración.
if MAX == 0
  for i = 1:1:(last_MAX+1)

      pose4=pose3(i,:)';

      R = pose4(4:6,:);
```

```
Rrad = (R.*pi)./180;
cphi = cos(Rrad(1));
ctita = cos(Rrad(2));
cpsi = cos(Rrad(3));
sphi = sin(Rrad(1));
stita = sin(Rrad(2));
spsi = sin(Rrad(3));
sphi_stita = sphi*stita;
cphi_stita = cphi*stita;

RR2 = [ctita*cpsi+sphi_stita*spsi, -ctita*spsi+sphi_stita*cpsi,
       cphi_stita, pose4(1);
       cphi*spsi,  cphi*cpsi,  -sphi,  pose4(2);
       -stita*cpsi+sphi*ctita*spsi,  stita*spsi+sphi*ctita*cpsi,
       cphi*ctita,  pose4(3);
       0, 0, 0, 1];

A2 = RR2*A;
```

Una vez calculada la cinemática inversa creamos las matrices de coordenadas para graficar el resultado. X2, Y2 y Z2 son las filas 1, 2 y 3 de la matriz A2 y las matrices XPi, YPi y ZPi (i = 1:6) son las uniones (motores) entre la base fija y la base móvil.

Por último, mediante la función plot3 se grafica la simulación en 3D y como parámetros podemos observar que a las uniones se les asigna el color negro, a la base fija el azul y a la base móvil el color rojo.

```
X2 = [A2(1,1) A2(1,2) A2(1,3) A2(1,4) A2(1,5) A2(1,6) A2(1,1)];
Y2 = [A2(2,1) A2(2,2) A2(2,3) A2(2,4) A2(2,5) A2(2,6) A2(2,1)];
Z2 = [A2(3,1) A2(3,2) A2(3,3) A2(3,4) A2(3,5) A2(3,6) A2(3,1)];


XP1 = [B(1,1) A2(1,1)];
YP1 = [B(2,1) A2(2,1)];
ZP1 = [B(3,1) A2(3,1)];


XP2 = [B(1,2) A2(1,2)];
YP2 = [B(2,2) A2(2,2)];
ZP2 = [B(3,2) A2(3,2)];


XP3 = [B(1,3) A2(1,3)];
YP3 = [B(2,3) A2(2,3)];
ZP3 = [B(3,3) A2(3,3)];
```

```
XP4 = [B(1,4) A2(1,4)];
YP4 = [B(2,4) A2(2,4)];
ZP4 = [B(3,4) A2(3,4)];

XP5 = [B(1,5) A2(1,5)];
YP5 = [B(2,5) A2(2,5)];
ZP5 = [B(3,5) A2(3,5)];

XP6 = [B(1,6) A2(1,6)];
YP6 = [B(2,6) A2(2,6)];
ZP6 = [B(3,6) A2(3,6)];

pause(0.1);
plot3(X2,Y2,Z2,'r',X0,Y0,Z0,'b',XP1,YP1,ZP1,'-k',XP2,YP2,ZP2,...
      '-ok',XP3,YP3,ZP3,'-ok',XP4,YP4,ZP4,'-ok',XP5,YP5,ZP5,...
      '-ok',XP6,YP6,ZP6,'-ok');

       hold off
       xlabel('eje x')
       ylabel('eje y')
       zlabel('eje z')

       axis([-25,25,-25,25,0,30]);
       axis square;
       grid off
  end
```

En el caso de que MAX =! 0, se procede de la misma forma que lo visto anteriormente, pero esta vez la iteración se realiza desde 1 hasta MAX+1.

```
%Si MAX =! 0, se utiliza la variable MAX para la iteración.
else
      for i = 1:1:(MAX+1)
```

Por último, si temp =! 0, el vector de posición no es aceptado y se indica por pantalla mediante un mensaje.

```
else

    fprintf('\n Vector de posición no aceptado \n');

end
```

5.4 Resultados

A continuación se expone el resultado de enviar la plataforma a la posición:

- [1 1 19 20 0 0]

```
Introduce el Valor de X: 1
Introduce el Valor de Y: 1
Introduce el Valor de Z: 19
Introduce el Valor de PITCH: 20
Introduce el Valor de ROLL: 0
Introduce el Valor de YAW: 0

Vector de Posición aceptado
```

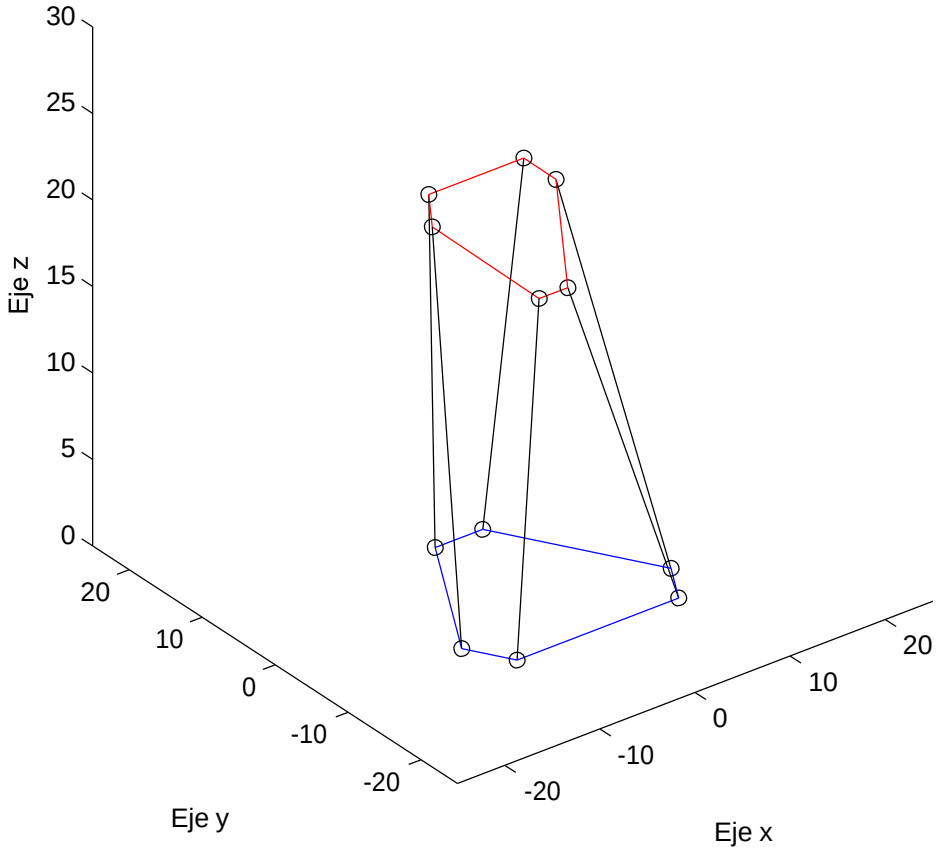


Figura 35. Gráfica Plataforma Posición final

6. Programación de la plataforma

6.1 Principio de la programación

El proceso de programación referido a la robótica consiste en enviarle las instrucciones necesarias al sistema de control (mediante el empleo de un lenguaje de programación) para que el robot realice unas tareas determinadas.

Para realizar esta programación se establecerá un algoritmo que cumpla con las especificaciones requeridas y posteriormente será traducido a un lenguaje que sea inteligible por el sistema de control que gobierna el robot. Se empleará, por tanto, los denominados lenguajes de alto nivel, los cuales permiten una mayor facilidad de comprensión para los humanos y que posteriormente el propio programa se encargará de traducirlo al lenguaje máquina empleado por el controlador. Actualmente no existen un lenguaje de programación estandarizado y general en el ámbito de la robótica, puesto que en la mayoría de los casos son los fabricantes los que establecen su propio lenguaje de programación para un producto concreto (algunos ejemplos de estos lenguajes son: VAL, RCL, AL, MAPLE, RAPT, LAMA, STRIPS, etc.). También existen los lenguajes de propósito general que pueden utilizarse para cualquier tipo de aplicación y son muy usados en robótica. En nuestro caso emplearemos este tipo de lenguajes, centrándonos en algunos específicamente como son MATLAB, Python o el propio lenguaje de Arduino.

A la hora de enfrentarse a la programación de cualquier prototipo o producto también es necesario tener en cuenta algunos aspectos que deben estar implementados en el código como puede ser sistemas de protección para la seguridad de los usuarios finales o para evitar un deterioro en la propia estructura del producto. En el caso de este proyecto en particular, al trabajar con un robot paralelo se debe poner especial atención en los siguientes aspectos:

- El movimiento de los actuadores, debido a que existen relaciones de tipo mecánicas que si no se tienen en cuenta mediante un sistema de protección pueden generar esfuerzos no deseados que deriven en la rotura de algún elemento.
- Realizar en todo momento movimientos suaves y controlados para evitar accidentes tanto en la estructura del robot como en el ambiente de trabajo en el que estará ejerciendo sus funciones.
- Un interfaz sencillo y lo más guiado posible para evitar un posible error al introducir los valores deseados.
- Métodos se seguridad que eviten atender a instrucciones de posición que no sean alcanzables físicamente por el robot.

En este proyecto partimos de un programa previamente diseñado para la plataforma, en el que se hacía una distinción entre el cálculo de las consignas de cada actuador aplicando el método de la cinemática inversa (para lo cual se empleaba el software matemático MATLAB) y el posicionamiento y control de los actuadores (código implementado directamente en Arduino). Sin embargo, se ha considerado oportuno realizar algunos cambios en el programa, para lograr cumplir con los objetivos

propuestos para la mejora del dispositivo, y los cuales se explicarán en los apartados siguientes.

6.2 Controlador PID

Un controlador PID (Proporcional, Integral y Derivativo) es una herramienta mediante la que es posible controlar un sistema atendiendo al error existente entre el valor actual que posee la variable que deseamos controlar y la consigna o valor final que se desea que tenga dicha variable. Por lo tanto, se trata de un sistema que debe estar realimentado para ir conociendo la diferencia en todo momento entre los dos parámetros mencionados e ir ajustando la salida en función de ello. Este controlador está formado por tres acciones que se aplican de manera conjunta sobre el error con el objetivo de obtener una salida apropiada y las cuales se exponen a continuación:

- Acción proporcional (P): Consiste en aplicar una señal proporcional al error actual. De esta forma, para un error grande, se envía una señal de control grande, y pequeña cuando el error es pequeño. El problema de aplicar solamente esta acción reside en que no proporciona un error nulo en el estacionario. El cálculo asociado a esta acción equivale a:

$$P(t) = k_p \cdot error(t) \quad ; \quad \text{donde } k_p : \text{constante proporcional}$$

- Acción integral (I): Consiste en aplicar una señal proporcional a la integral del error. Aplicando esta acción conseguimos anular el error en el estado estacionario pero puede provocar sobreoscilaciones e inestabilidad. El problema de esta aportación es que es acumulativa y si no se limita puede tomar valores muy grandes con el tiempo. Esta contribución integral toma la siguiente forma:

$$I(t) = q_i \int_0^t error(t)dt \quad ; \quad \text{donde } q_i : \text{constante integral}$$

- Acción derivativa (D): Consiste en aplicar una señal proporcional a la derivada del error. Esta aportación sirve para limitar la acción de control en los momentos en que el error es muy elevado, reduciendo las sobreoscilaciones. La forma de esta contribución sigue la forma siguiente:

$$D(t) = q_d \frac{d\, error(t)}{dt} \quad ; \quad \text{donde } q_d : \text{constante derivativa}$$

Por lo tanto, la ecuación de un controlador PID estará compuesta por la aportación de cada una de las acciones explicadas anteriormente y se corresponderá con:

$$u(t) = P(t) + I(t) + D(t)$$

Sin embargo, esta fórmula se corresponde con PID de tipo continuo y en este proyecto nuestra herramienta de muestreo de los datos es Arduino, el cual nos

proporciona medidas discretas del error, es decir, tomadas cada cierto intervalo de tiempo y no de manera continua. Concretamente, las medidas de los potenciómetros que nos dan información sobre la posición de los motores se realiza dentro de una interrupción que se ejecuta cada 3 milisegundos, por lo tanto es necesario el empleo de un PID de tipo discreto que es muy similar al PID continuo pero con algunas modificaciones en sus ecuaciones. Las acciones de un controlador de este tipo son las siguientes:

$$P_k = k_p \cdot error_k \quad ; \quad \text{donde } k \text{ : instante actual}$$

$$I_k = I_{k-1} + q_i \cdot error_k \quad ; \quad \text{donde } k-1 \text{ : instante anterior}$$

$$D_k = q_d \cdot (error_k - error_{k-1})$$

Como se puede observar, en este caso se tiene en cuenta los instantes de tiempo actual y anterior que reflejan que no se trata de un sistema continuo sino de tipo discreto, lo cual supone un tipo de controlador posible de implementar en este proyecto.

Las constantes proporcional, integral y derivativa necesarias para la sintonización del PID fueron escogidas mediante ensayo buscando, en primer momento, unos valores para las constantes proporcional y derivativa que permitieran una respuesta rápida y suave ante el cambio de la consigna. Después de conseguir esto, se fue incrementando progresivamente la constante integral hasta un punto en el que se consiguiera anular el error estacionario sin provocar inestabilidad.

6.3 Adquisición de los valores de posición de los motores

Para conocer la posición de los motores, estos están dotados de un potenciómetro cuyo valor de resistencia nos da una referencia sobre la longitud que posee el actuador. Trabajan con una diferencia de potencial de 5V, donde 0V corresponde con el pistón completamente retraído y 5V cuando está completamente extendido.

Estas variaciones de voltaje son leídas a través de los pines en entradas analógicas de Arduino, el cual está dotado de un conversor analógico-digital (CAD) que las convierte en señales digitales con un rango comprendido entre 0 y 1023 para poder procesarlas y trabajar con ellas. En el caso del Arduino Mega con el que contaba originalmente la plataforma, el CAD tenía una frecuencia de reloj de 16MHz que se dividía entre un factor de escala que fue modificado para finalmente obtener una frecuencia de 250KHz, o lo que es lo mismo, 1 ciclo cada 4us (ya que el período es igual a la inversa de la frecuencia). Esta modificación se realizó modificando algunos bits específicos de un determinado registro.

Para el caso de Arduino Due, la modificación de la frecuencia de reloj del CAD se establece por defecto en 21MHz, es decir, 1 ciclo cada 36.6us. Sin embargo, se

modificó uno de sus registros para disminuir este tiempo hasta lograr una frecuencia de 1 ciclo cada 3.97us añadiendo la siguiente línea de código:

```
REG_ADC_MR = (REG_ADC_MR & 0xFFF0FFFF) | 0x00020000;
```

Con esto conseguimos una mayor velocidad de conversión y por tanto una mejor respuesta del sistema en cuanto a la las señales analógicas leídas de los potenciómetros.

6.4 Frecuencia de la PWM

La PWM (Pulse Width Modulation o Modulación por Ancho de Pulso) es el tipo de señal mediante la que se alimenta a los motores para moverlos. Su ciclo de trabajo está comprendido entre 0 y 255 siendo el '0' equivalente a una PWM con una anchura de pulso nula, y el '255' una PWM con una anchura de pulso igual al 100% del periodo de la señal.

Estas señales también tienen una frecuencia determinada que viene definida por la placa de Arduino y, al igual que en el caso del CAD, puede ser modificada. En el proyecto previo se configuró esta frecuencia a 4000Hz realizando algunas modificaciones sobre el propio código. En el caso de Arduino Due, esta configuración no es tan evidente y se debe realizar sobre el archivo 'wiring_analog.c' almacenado en la dirección 'arduino/hardware/arduino/sam/cores/arduino/'.Una vez abierto este archivo se debe buscar la siguiente parte del código:

```
if ((attr & PIN_ATTR_PWM) == PIN_ATTR_PWM) {
    ulValue = mapResolution(ulValue, _writeResolution, PWM_RESOLUTION);

    if (!PWMEnabled) {
        // PWM Startup code
        pmc_enable_periph_clk(PWM_INTERFACE_ID);
        PWMC_ConfigureClocks(PWM_FREQUENCY * PWM_MAX_DUTY_CYCLE, 0, VARIANT_MCK);
        PWMEnabled = 1;
    }

    uint32_t chan = g_APinDescription[ulPin].ulPWMChannel;
    if (!pinEnabled[ulPin]) {
        // Setup PWM for this pin
        PIO_Configure(g_APinDescription[ulPin].pPort,
                g_APinDescription[ulPin].ulPinType,
                g_APinDescription[ulPin].ulPin,
                g_APinDescription[ulPin].ulPinConfiguration);
        PWMC_ConfigureChannel(PWM_INTERFACE, chan, PWM_CMR_CPRE_CLKA, 0, 0);
        PWMC_SetPeriod(PWM_INTERFACE, chan, PWM_MAX_DUTY_CYCLE);
        PWMC_SetDutyCycle(PWM_INTERFACE, chan, ulValue);
        PWMC_EnableChannel(PWM_INTERFACE, chan);
        pinEnabled[ulPin] = 1;
    }

    PWMC_SetDutyCycle(PWM_INTERFACE, chan, ulValue);
    return;
}
```

Y realizar las siguientes modificaciones:

```
if ((attr & PIN_ATTR_PWM) == PIN_ATTR_PWM) {
    //ulValue = mapResolution(ulValue, _writeResolution, PWM_RESOLUTION);


    if (!PWMEnabled) {
        // PWM Startup code
        pmc_enable_periph_clk(PWM_INTERFACE_ID);
        //PWMC_ConfigureClocks(PWM_FREQUENCY * PWM_MAX_DUTY_CYCLE, 0, VARIANT_MCK);
        // Select CLKA source as the clock with freq as specified i.e 2000KHz
        PWMC_ConfigureClocks(2000 * 1000, 0, VARIANT_MCK);
        PWMEnabled = 1;
    }

    uint32_t chan = g_APinDescription[ulPin].ulPWMChannel;
    if (!pinEnabled[ulPin]) {
        // Setup PWM for this pin
        PIO_Configure(g_APinDescription[ulPin].pPort,
                g_APinDescription[ulPin].ulPinType,
                g_APinDescription[ulPin].ulPin,
                g_APinDescription[ulPin].ulPinConfiguration);
        PWMC_ConfigureChannel(PWM_INTERFACE, chan, PWM_CMR_CPRE_CLKA, 0, 0);
        //PWMC_SetPeriod(PWM_INTERFACE, chan, PWM_MAX_DUTY_CYCLE);
        PWMC_SetPeriod(PWM_INTERFACE, chan, 2000*0.25);
        PWMC_SetDutyCycle(PWM_INTERFACE, chan, ulValue);
        PWMC_EnableChannel(PWM_INTERFACE, chan);
        pinEnabled[ulPin] = 1;
    }

    PWMC_SetDutyCycle(PWM_INTERFACE, chan, ulValue);
    return;
}
```

Con esto modificamos la frecuencia y el ciclo de trabajo máximo que viene establecidos por defecto y también modificamos el período para obtener una frecuencia de 4000Hz o 1 ciclo cada 0.25ms.

6.5 Realimentación del sistema de control

Uno de los problemas con los que se tuvo que lidiar fue el tipo de control que se le aplicaba a los motores para conseguir llegar a la posición deseada. La plataforma tenía implementado un sistema de control de tipo punto a punto en el que se le enviaban las consignas necesarias para cada motor y mediante un controlador de tipo PID se ordenaba a cada uno moverse lo más rápido posible desde su posición actual hasta la deseada. Sin embargo, debido a las propiedades de la plataforma estudiada, un control de este tipo supone varios problemas entre los cuales se encuentra el hecho de que no se consigue una trayectoria suave y controlada como sería deseable. Este problema puede llegar a ser muy grave si se tiene en cuenta que el desplazamiento de cada actuador influye en los demás, y por lo tanto, cabe la posibilidad de someterlos a grandes esfuerzos que supongan un comportamiento no deseado o incluso la rotura de los mismos. También supone una desventaja si el objetivo del robot es realizar un trabajo de manera precisa, puesto que si el movimiento es muy brusco el resultado obtenido no será bueno.

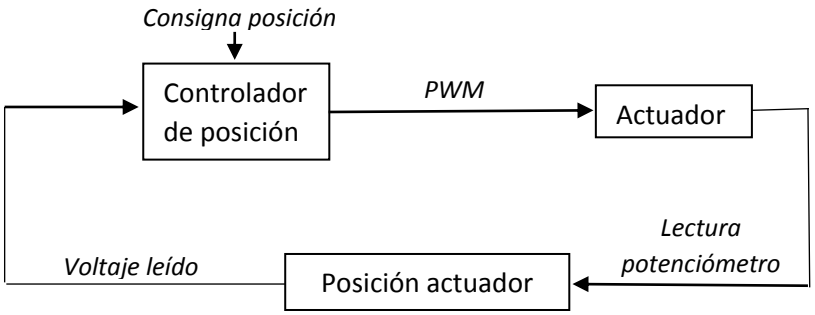


Figura 36. Diagrama de control original de cada motor

Por lo tanto, se pensaron diferentes alternativas basadas en implementar algún tipo de realimentación adicional para intentar corregir este defecto. Finalmente se decidió añadir el parámetro de la velocidad de cada motor al cálculo, además del control de posición ya implementado. La explicación de la solución se basa en el hecho de que al mover la plataforma a una posición determinada, cada motor va a tener un recorrido diferente y, evidentemente, sería deseable que todos lleguen a su posición de manera simultánea para evitar forzarlos durante el trayecto. Para conseguir este objetivo, se recurre al hecho de que las velocidades de cada motor necesarias para que todos lleguen a un punto de manera simultánea serán diferentes, de forma que los motores que deban realizar una trayectoria mayor deberán también elevar su velocidad, mientras que los de menor recorrido deberán hacer su movimiento de forma más lenta. Llevar esta idea a la práctica supuso añadir otro controlador también de tipo PID que controlase dicha variable.

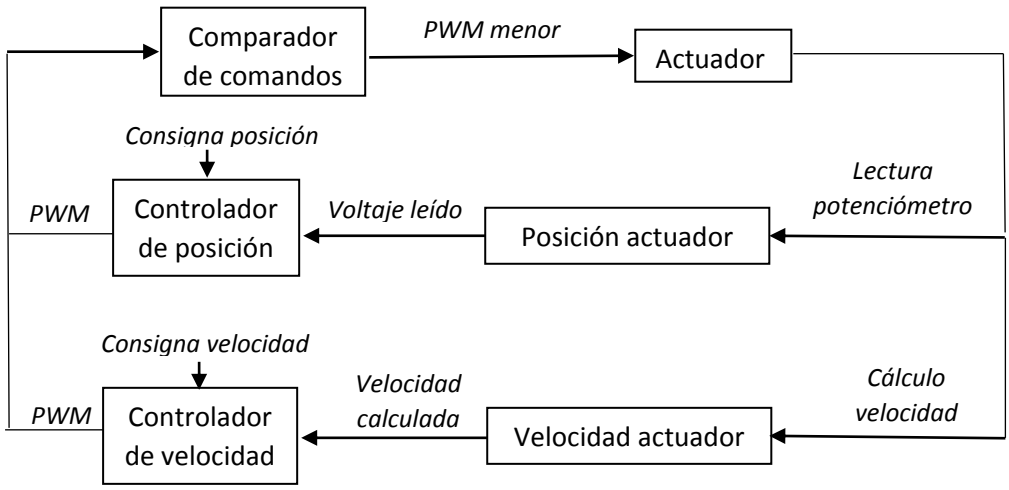


Figura 37. Diagrama de control de cada motor con realimentación de velocidad

Es decir, finalmente obtendríamos para cada motor dos controladores: uno sería el encargado de llevar el actuador desde la posición actual a la que marque la consigna y el otro debería controlar la velocidad a la que dicho motor se desplaza de un punto al otro. Para que ambos controladores funcionen conjuntamente se escoge en cada momento el que tenga como resultado un comando menor, para que de esta forma,

generalmente se desplacen de tal manera que durante el recorrido se les enviará el ancho de PWM que marque el controlador de velocidad y al acercarse a la posición indicada por la consigna actuará el comando de posición, haciendo posible realizar una trayectoria suave.

Como la carga computacional aumentaría al añadirle un controlador adicional para la velocidad, se ha considerado una buena opción cambiar el hardware que realiza las operaciones necesarias. El motivo de esta decisión se basa en que el medio sobre el que se ejecutaba el programa original era un Arduino Mega 2560 pero, sin embargo, para poder ejecutar el nuevo código sin ningún problema sería necesario emplear un dispositivo más potente y con una velocidad de procesamiento mayor para conseguir una respuesta óptima. Por esta razón se sustituyó la placa existente por un Arduino Due cuyas características suponían una ventaja para el desarrollo de las mejoras implementadas en el presente proyecto.

6.6 Programación en Arduino

Arduino es un hardware libre (es decir, que sus especificaciones y diagramas esquemáticos son de acceso público y gratuitos), que consiste en una placa de circuito impreso (PCB) dotada de un microcontrolador (cuyo tipo y características varían dependiendo del modelo) y puertos digitales y analógicos de entrada y salida. Se programa a través de un entorno de desarrollo diseñado de manera sencilla y óptima que convierten a estos dispositivos en herramientas verdaderamente útiles para la creación de proyectos y aplicaciones muy diversas.

El lenguaje de programación empleado es propio de este hardware y está basado en Wiring, siendo un lenguaje sencillo que facilita en gran medida la tarea del programador.

Las funciones de este dispositivo pueden resumirse de manera muy básica en tres: en primer lugar tenemos una interfaz de entrada, que puede estar directamente unida a los periféricos, o conectarse a ellos a través de puertos. El objetivo de esta interfaz de entrada es llevar la información al microcontrolador que será el elemento encargado de procesar esos datos. Las características del microcontrolador es la principal diferencia entre los diversos modelos de estas placas y se deberá evaluar las necesidades específicas de cada proyecto para saber qué tipo de Arduino implementar. Por último tenemos una interfaz de salida, que lleva la información procesada a los periféricos de salida a los cuales estarán conectados los dispositivos encargados de hacer uso de los datos finales.

6.6.1 Arduino Mega

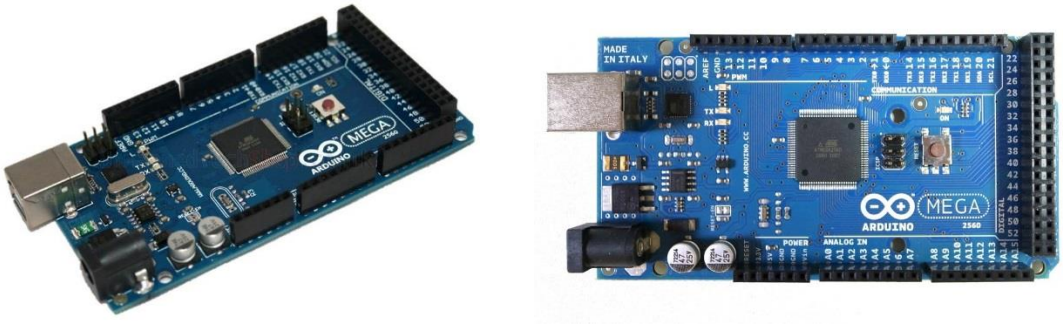


Figura 38. Placa Arduino Mega

El Arduino Mega está basado en el microcontrolador ATMega2560. Tiene 54 pines de entradas/salidas digitales (14 de las cuales pueden ser utilizadas como salidas PWM), 16 entradas analógicas, 4 UARTs, un cristal oscilador de 16 Mhz, conexión USB, jack de alimentación, conector ICSP y botón de reset.

Características:

- Microcontrolador ATmega2560.
- Voltage de entrada de – 7-12V.
- 54 pines digitales de Entrada/Salida (14 de ellos son salidas PWM).
- 16 entradas analógicas.
- 256k de memoria flash.
- Velocidad del reloj de 16Mhz

6.6.2 Arduino Due

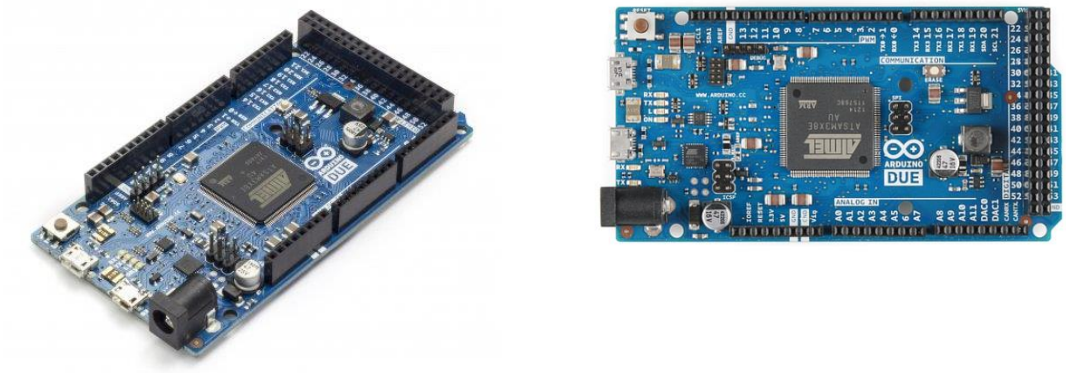


Figura 39. Placa Arduino Due

Arduino Due es una placa con un microcontrolador Atmel SAM3X8E ARM Cortex-M3 de 32 bits. Este chip trabaja a 84Mhz, con un voltaje máximo de 3,3 voltios

y aporta una potencia de cálculo bastante superior a otros microcontroladores Arduino. Por eso es idóneo para todos aquellos proyectos con alta capacidad de procesamiento. Para el almacenamiento se dispone de 512KB de flash, una cantidad muy grande de memoria para cualquier código de programación.

El sistema dispone de 54 pines de entrada y salida digitales, de los cuales 12 de ellos pueden ser usados como PWM. También tiene 12 pines analógicos, 4 UARTs , capacidades de conexión USB OTG, dos conexiones DAC (conversión digital a analógico), 2 TWI, un power jack, SPI y JTAG.

Características:

- Microcontrolador: AT91SAM3X8E
- Voltaje de operación: 3.3V
- Voltaje de entrada recomendado: 7-12V
- 54 pines digitales de Entrada/Salida (12 de ellos son salidas PWM).
- 12 entradas analógicas.
- 2 salidas analógicas.
- Memoria Flash: 512 KB disponibles para aplicaciones del usuario
- Velocidad del reloj de 84 MHz

6.6.3 Comparación entre Arduino Mega Y Arduino Due

Como se ha comentado anteriormente, en el momento de crear el proyecto original solo se implementó el código que realizaba el control de posición de los motores, y se programó el cálculo de las consignas mediante la cinemática inversa en MATLAB, con el objetivo de no sobrecargar la placa, ya que se estimó que la velocidad del procesador no sería suficiente para realizar todos los cálculos de manera óptima.

En este caso, otro de los objetivos planteados en este trabajo era conseguir unificar la programación en un solo código y que la placa fuera capaz de procesarlo sin ningún problema, para lo cual se optó por remplazar el Arduino Mega del que disponía la plataforma robótica por un Arduino Due. El principal motivo de esta elección fue que la velocidad de reloj pasaría de 16Mhz a 84Mhz, lo cual supondría una mejora notable en la velocidad de procesamiento. A modo de resumen, se expone a continuación una tabla comparativa entre las características más relevantes de ambas placas.

Característica de Arduino	UNO	Mega 2560	Leonardo	DUE
Tipo de microcontrolador	Atmega 328	Atmega 2560	Atmega 32U4	AT91SAM3X8E
Velocidad de reloj	16 MHz	16 MHz	16 MHz	84 MHz
Pines digitales de E/S	14	54	20	54
Entradas analógicas	6	16	12	12
Salidas analógicas	0	0	0	2 (DAC)
Memoria de programa (Flash)	32 Kb	256 Kb	32 Kb	512 Kb
Memoria de datos (SRAM)	2 Kb	8 Kb	2.5 Kb	96 Kb
Memoria auxiliar (EEPROM)	1 Kb	4 Kb	1 Kb	0 Kb

Figura 40. Tabla de características Arduino

6.7 Cinemática inversa

La cinemática inversa es un método muy empleado en robótica para determinar los valores de desplazamiento y rotación de un actuador a partir de las coordenadas de la posición a adoptar. En nuestro caso, se emplea la cinemática inversa para saber el desplazamiento y rotación que debe realizar cada motor con el objetivo de conseguir la pose de la base móvil deseada.

Para ello, se debe crear una matriz de transformación que haga posible relacionar el sistema de referencia solidario a la base móvil con el sistema de referencia de la base fija, para poder referenciar los puntos a un único sistema y obtener las coordenadas de manera adecuada. Esta matriz está formada por una matriz de rotación, un vector de posición, un vector de cambio de perspectiva y un elemento denominado factor de escala. En este caso en concreto, el cambio de perspectiva y el factor de escala no tendrán ninguna función, adoptando el valor 0 y 1 respectivamente. La matriz de transformación quedaría por lo tanto de la siguiente forma:

$$H_1^0 = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde: **Rojo** = matriz de rotación

Verde = matriz de traslación

Azul = vector de perspectiva

Violeta = factor de escala

Figura 41. Matriz de Transformación

El vector de traslación corresponde con las coordenadas del vector que va desde el centro de la plataforma inferior hasta el centro de la plataforma superior. Por otra parte, la matriz de rotación indica los ángulos que tomaría la base móvil con respecto a la base fija.

La matriz de transformación que vimos anteriormente, en nuestro caso, es aplicada a cada uno de los puntos de anclaje de la base superior para comprobar donde quedaría dicho punto tras el movimiento, con respecto al sistema de referencia de la base fija. Tras esto, serían conocidos todos los puntos de anclaje superiores e inferiores y solo faltaría calcular la distancia que existe entre los anclajes de cada motor, haciendo uso del cálculo de la distancia euclídea (derivada del teorema de Pitágoras).

Volviendo a la matriz de rotación, es la resultante de aplicar tres rotaciones genéricas a la plataforma superior: un ángulo de guiñada (yaw, en torno al eje y), otro de cabeceo (pitch, en torno al eje x) y otro de alabeo (roll, en torno al eje z), los denominados ángulos de Euler.

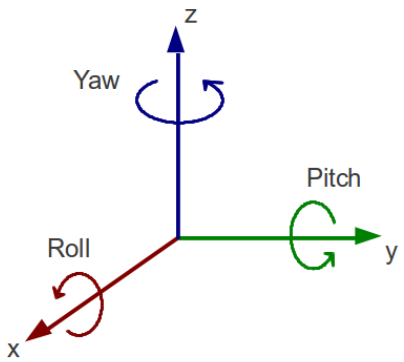
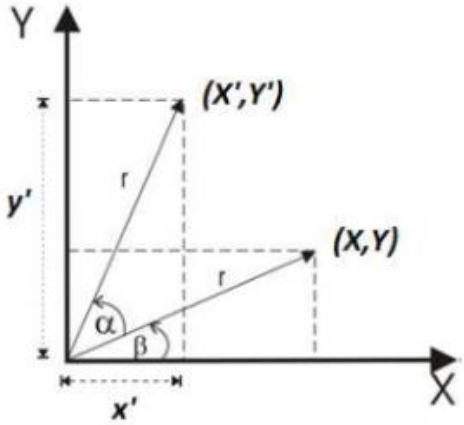


Figura 42. Ángulos de Euler

Para obtener la matriz de rotación final, se puede estudiar por separado cada uno de los giros y finalmente unir los tres movimientos en esta matriz. Esto simplifica los cálculos, ya que, cada giro por separado se puede estudiar como un problema en 2D.

Podríamos estudiar, a modo de ejemplo, el giro roll (en torno al eje z).



El vector inicial XY, forma un ángulo β con el eje X. Dicho vector se gira un ángulo α en sentido antihorario (lo consideraremos positivo) dando lugar al vector X'Y'. Partiendo del dibujo podemos observar lo siguiente:

$$x' = r \cdot \cos(\alpha + \beta) = r \cdot [\cos(\alpha) \cdot \cos(\beta) - \sin(\alpha) \cdot \sin(\beta)]$$

$$x' = r \cdot \cos(\alpha)\cos(\beta) - r \cdot \sin(\alpha)\sin(\beta)$$

$$pero\ como, \quad x = r \cdot cos(\beta) \quad ; \quad y = r \cdot sin(\beta) \quad entonces:$$

$$\boxed{x' = x \cdot \cos(\alpha) - y \cdot \sin(\alpha)}$$

De forma similar obtenemos:

$$\boxed{y' = x \cdot \sin(\alpha) + y \cdot \cos(\alpha)}$$

Por tanto, de forma matricial, podemos escribir el giro de la siguiente manera:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Esta matriz expresada de forma tridimensional quedaría:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

La tercera fila y columna de la matriz de rotación toma esta forma para dejar invariante el eje z ya que el giro es en torno a él.

De igual forma que hemos hecho este cálculo, podríamos realizar lo mismo para los giros pitch y yaw. Las matrices de rotación resultantes serían las siguientes:

- Pitch (β, eje x):

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) \\ 0 & \sin(\beta) & \cos(\beta) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

- Yaw (γ, eje y):

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos(\gamma) & 0 & \sin(\gamma) \\ 0 & 1 & 0 \\ -\sin(\gamma) & 0 & \cos(\gamma) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

- Roll (α, eje z):

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

La matriz de rotación final se formaría al multiplicar las matrices Z·Y·X, en este orden. A continuación podemos ver el resultado de esta multiplicación. Hay que tener en cuenta que el signo de los elementos depende de qué sentido se haya tomado como positivo para cada ángulo. Para el caso de este ejemplo ha resultado:

$$R_1^0 = \begin{pmatrix} \cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\gamma)\sin(\beta) & \sin(\alpha)\sin(\beta) + \cos(\alpha)\sin(\gamma)\cos(\beta) \\ \sin(\alpha)\cos(\gamma) & \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\gamma)\sin(\beta) & -\cos(\alpha)\sin(\beta) + \sin(\alpha)\sin(\gamma)\cos(\beta) \\ -\sin(\gamma) & \cos(\gamma)\sin(\beta) & \cos(\gamma)\cos(\beta) \end{pmatrix}$$

Con la matriz de rotación obtenida, lo único que faltaría sería armar la matriz de transformación, sustituyendo los ángulos α, β y γ de la matriz de rotación por los valores deseados e incluyendo el vector de posición pertinente.

En nuestro caso, la cinemática inversa se emplea dentro del código para poder calcular el desplazamiento y rotación necesarios para alcanzar la posición determinada por la pose que se desea que adopte la plataforma. Una vez hemos aplicado este método, ya se poseen los argumentos de entrada suficientes para que los controladores de posción y velocidad generen los comandos de PWM necesarios para el movimiento de los motores.

6.8 Descripción del programa

Al realizar el cambio de hardware ha sido necesario hacer algunas modificaciones en el código con el objetivo de adaptar el programa al funcionamiento del Arduino Due (debido a que se programó para que funcionase en el modelo Arduino Mega) y se ha añadido la parte de control de velocidad y funcionamiento simultáneo de ambos controladores. El programa completo se describe a continuación además de estar incluido como anexo de la memoria.

Los programas diseñados para Arduino tienen una estructura bastante sencilla en la que se pueden diferenciar tres partes básicas:

- Una parte inicial en la que se incluyen las librerías que contienen las funciones necesarias que se emplean a lo largo del código, definición de macros y declaración de las variables globales cuyo valor estará disponible desde cualquier parte del código.
- El bloque denominado como ‘setup’, donde se llevan a cabo todas aquellas configuraciones necesarias a ajustar en el procesador para la correcta ejecución del programa, como por ejemplo, la definición de los pines utilizados como entradas o salidas, la configuración de interrupciones y los ajustes de diferentes parámetros del hardware como velocidad de reloj o frecuencia de algunas señales generadas.
- El bloque ‘loop’, que como su nombre indica, posee un comportamiento cíclico (es decir, que se está ejecutando constantemente durante el funcionamiento del programa) es el encargado de almacenar el código del programa. Este código está compuesto por las instrucciones que definen el comportamiento de los pines de salida, la forma en que se emplea la información obtenida de las entradas y la realización de todos los cálculos necesarios para llevar esto a cabo de forma correcta, así como las comunicaciones mediante el puerto serial si fuera necesario.

6.8.1 Inclusión de librerías, macros y declaración de variables

En primer lugar, se incluye la librería 'DueTimer' que posibilita el uso de las funciones de temporización de la placa. Se definen los pines de salida que servirán para controlar los motores, ya que cada actuador cuenta con dos pines de salida que irán conectados al puente en H que se emplean para controlar el sentido del movimiento y un pin de enable mediante el cual se enviará la PWM calculada por el sistema de control implementado. También se definen las variables de los controladores (constantes proporcionales, derivativas e integrales de cada PID), así como los vectores y variables globales que almacenarán una información específica la cual se empleará en alguno de los bloques del programa para realizar las operaciones necesarias. A continuación se muestra este fragmento del código:

```
#include "DueTimer.h" //Se incluye la librería específica para
controlar los  timer de Arduino Due

int entrada1[6]={48,44,40,36,32,28}; //Vector que define los
pines de la entrada 1 de los puentes en H de cada motor
int entrada2[6]={49,45,41,37,33,29}; //Vector que define los
pines de la entrada 2 de los puentes en H de cada motor
int enable[6]={7,6,5,4,3,2}; //Vector que define los pines de
enable por el cual  se enviarán los comandos de PWM a cada
motor

//Variables para el controlador:
float kppos = 180; //Parte proporcional del
controlador PID de posición
float qipos = 0.1; //Parte integral del controlador
PID de posición
float qdpos = 180; //Parte derivativa del
controlador PID de posición
float kpvel = 90.0; //Parte proporcional del
controlador PID de velocidad
float qivel = 0.1; //Parte integral del controlador
PID de velocidad
float qdvel = 90.0; //Parte derivativa del
controlador PID de velocidad
float Ppos[6] = {0,0,0,0,0,0}; //Vector que almacena la acción
proporcional del  PID de posición calculada para cada motor
float Ipos[6]={0,0,0,0,0,0}; //Vector que almacena la acción
integral del PID de  posición calculada para cada motor
float Dpos[6]={0,0,0,0,0,0}; //Vector que almacena la acción
derivativa del PID  de posición calculada para cada motor
float Pvel[6] = {0,0,0,0,0,0}; //Vector que almacena la acción
proporcional del  PID de velocidad calculada para cada motor
float Ivel[6]={0,0,0,0,0,0}; //Vector que almacena la acción
```

```
integral del PID de velocidad calculada para cada motor
float Dvel[6]={0,0,0,0,0,0}; //Vector que almacena la acción
derivativa del PID de velocidad calculada para cada motor

float consignapos[6]={2.5,2.5,2.5,2.5,2.5,2.5}; //Vector que
contiene el valor de las consignas de posición de cada motor
float consignavel[6]={0,0,0,0,0,0}; //Vector que contiene el
valor de las consignas de velocidad de cada motor

float posicion[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor leído de posición de cada motor
float velocidad[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor calculado de velocidad de cada motor

float error0pos[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error calculado de posición de cada motor
float error1pos[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error anterior de posición de cada motor
float error0vel[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error calculado de velocidad de cada motor
float error1vel[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error anterior de velocidad de cada motor

float salidaMotorPosiciones[6]={0,0,0,0,0,0}; //Vector que
contiene los comandos de PWM obtenidos del control de posición
float salidaMotorVelocidades[6]={0,0,0,0,0,0}; //Vector que
contiene los comandos de PWM obtenidos del control de velocidad

float tiemposMotores[6]={0,0,0,0,0,0}; //Vector que contiene
los tiempos necesarios en los que cada motor debe realizar su
recorrido
float tiempo = 0.0; // Variable que guarda el tiempo que tarda
el motor que realiza el mayor recorrido

float calcconsigna[6] = {0,0,0,0,0,0}; //Vector que contiene
el valor de las consignas de posición escaladas entre 0 y 1024
float consigna1[6] = {0,0,0,0,0,0}; //Vector que contiene las
consignas de posición en cm
float posicionanterior[6] = {0,0,0,0,0,0}; //Vector que guarda
la posición del instante anterior
float posicioninicial[6] = {0,0,0,0,0,0}; //Vector que guarda
las posiciones leídas al comenzar el programa
unsigned long tiempoLectActual = 0.0; //Variable que almacena
el tiempo en el que se realiza cada lectura de la posición

int inicio = 0; //Variable que indica si es la primera vez que
```

```
se ejecuta el programa
int inicioInterrup = 0; //Variable que indica si es la primera
vez que se ejecuta la interrupción
```

6.8.2 Función 'setup'

En esta función se establece la comunicación serial a una velocidad de 9600 bps (baudios por segundo), se configuran los pines declarados en el apartado anterior (entradas de los puente en H y enables) como salida, además de cuatros pines analógicos de entrada que sirven para leer los potenciómetros integrados en los motores y que nos dan información sobre la posición en la que se encuentran. Por último se configura la ejecución de una interrupción cada 3 milisegundos que llamará a la función 'control' encargada del control de los actuadores, con el objetivo de realimentar la información de la posición cada cierto período de tiempo.

```
void setup() {

  REG_ADC_MR = (REG_ADC_MR & 0xFFF0FFFF) | 0x00020000;
//Disminuye tiempo de conversión desde 36.6us hasta 3.97us

Serial.begin(9600); //Se abre el puerto serie con una
velocidad de 9600 bps
  //Se configuran como pines de salida de la placa las
entradas y enable de cada motor
pinMode(entrada1[0],OUTPUT);
pinMode(entrada2[0],OUTPUT);
pinMode(enable[0],OUTPUT);

pinMode(entrada1[1],OUTPUT);
pinMode(entrada2[1],OUTPUT);
pinMode(enable[1],OUTPUT);

pinMode(entrada1[2],OUTPUT);
pinMode(entrada2[2],OUTPUT);
pinMode(enable[2],OUTPUT);

pinMode(entrada1[3],OUTPUT);
pinMode(entrada2[3],OUTPUT);
pinMode(enable[3],OUTPUT);

pinMode(entrada1[4],OUTPUT);
pinMode(entrada2[4],OUTPUT);
pinMode(enable[4],OUTPUT);

pinMode(entrada1[5],OUTPUT);
pinMode(entrada2[5],OUTPUT);
```

```
pinMode(enable[5],OUTPUT);
  //Se configuran los pines de entradas analógicas encargados de leer los potenciómetros de los motores
pinMode(A1,INPUT);
pinMode(A2,INPUT);
pinMode(A3,INPUT);
pinMode(A4,INPUT);
pinMode(A5,INPUT);
pinMode(A6,INPUT);
//Se establece la función control como una interrupción que se ejecutará cada 3 ms.
Timer3.attachInterrupt(control).start(3000);
```

6.8.3 Función 'longitudes'.

Esta función se encarga de calcular los puntos a los que debe llegar cada actuador a través de una pose dada por el usuario, para a partir de aquí obtener la longitud total que se debe desplazar cada motor y generar las consignas que se deben enviar al controlador. Para llevar a cabo dicha función, se hace uso de la cinemática inversa que es un método bastante empleado en robótica con el que es posible determinar cuánto se deben de mover los actuadores a partir de las coordenadas que se desea alcanzar. Para ello, se emplea una matriz de transformación que en nuestro caso pasará a referenciar el sistema de coordenadas de la base móvil con el sistema de referencia de la base fija.

En Arduino no existe una manera específica de trabajar con matrices, lo cual nos lleva a realizar las operaciones separando cada matriz como vectores independientes. Tras declarar las matrices que contienen los puntos de referencia de la base fija (matriz B) y la base móvil (matriz A), se declaran ciertas variables que facilitan la creación de la matriz de transformación, para posteriormente definir completamente esta última matriz.

Anclajes superiores desde base móvil (en cm)						
	Motor 1	Motor 2	Motor 3	Motor 4	Motor 5	Motor 6
Coord. X	5	5.8	0.7	-0.4	-5.2	-4.3
Coord. Y	3.6	2.7	-5.6	-5.8	3.4	3.9
Anclajes inferiores desde base fija (en cm)						
	Motor 1	Motor 2	Motor 3	Motor 4	Motor 5	Motor 6
Coord. X	5	5.8	0.7	-0.4	-5.2	-4.3
Coord. Y	3.6	2.7	-5.6	-5.8	3.4	3.9

Datos extra (en cm)	
Altura desde base fija a base móvil en posición de mínima altura	15.5
Longitud mínima de cada extremidad completa (actuador + anclajes)	17.5

Figura 43. Tabla de coordenadas de los anclajes y longitudes importantes del robot

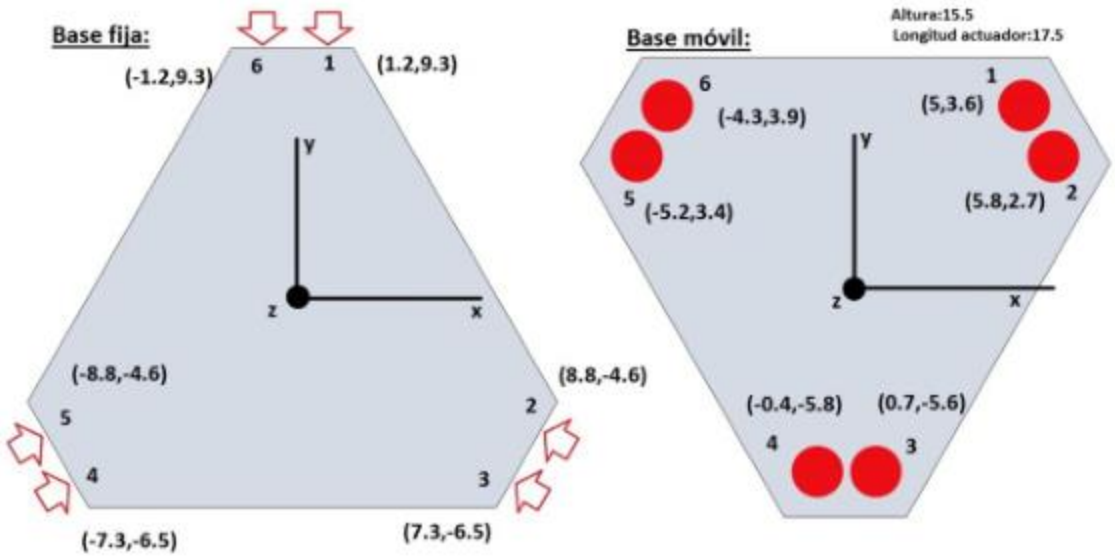


Figura 44. Coordenadas aproximadas de los anclajes de los actuadores en cm

```
int longitudes() //Función que calcula la cinemática inversa
y las consignas de los motores
{
float pose[] = {0,0,19,0,0,0}; //Vector que contiene la pose
que se desea alcanzar
//Matriz con los puntos de referencia de la base móvil
float Ax[] = {5,5.8,0.7,-0.4,-5.2,-4.3};
float Ay[] = {3.6,2.7,-5.6,-5.8,3.4,3.9};
float Az[] = {0,0,0,0,0,0};
//Matriz con los puntos de referencia de la base fija
float Bx[] = {1.2,8.8,7.3,-7.3,-8.8,-1.2};
float By[] = {9.3,-4.6,-6.5,-6.5,-4.6,9.3};
float Bz[] = {0,0,0,0,0,0};
float R[] = {pose[3],pose[4],pose[5]}; //Vector que contiene
los ángulos de orientación (p,y,r) que se desean alcanzar
float D[] = {pose[0],pose[1],pose[2]}; //Vector que contiene
las coordenadas (x,y,z) que se desean alcanzar
//Se pasan los ángulos expresados en grados a radianes
float Rrad1 = (R[0]*PI)/180;
float Rrad2 = (R[1]*PI)/180;
```

```
float Rrad3 = (R[2]*PI)/180;


//Se asignan variables con las funciones trigonométricas necesarias para realizar la matriz de transformación
float cphi = cos(Rrad1);
float ctita = cos(Rrad2);
float cpsi = cos(Rrad3);
float sphi = sin(Rrad1);
float stita = sin(Rrad2);
float spsi = sin(Rrad3);
float sphi_stita = sphi*stita;
float cphi_stita = cphi*stita;
//Matriz de transformación
float RRf1[] = {ctita*cpsi+sphi_stita*spsi, ctita*spsi+sphi_stita*cpsi, cphi_stita};
float RRf2[] = {cphi*spsi,cphi*cpsi,-sphi};
float RRf3[] = {-stita*cpsi+sphi*ctita*spsi,stita*spsi+sphi*ctita*cpsi, cphi*ctita};
//Multiplicación de la matriz de transformación por la matriz A y el resultado se almacena en una nueva matriz denominada C
float C11 = RRf1[0]*Ax[0]+RRf1[1]*Ay[0]+RRf1[2]*Az[0];
float C12 = RRf1[0]*Ax[1]+RRf1[1]*Ay[1]+RRf1[2]*Az[1];
float C13 = RRf1[0]*Ax[2]+RRf1[1]*Ay[2]+RRf1[2]*Az[2];
float C14 = RRf1[0]*Ax[3]+RRf1[1]*Ay[3]+RRf1[2]*Az[3];
float C15 = RRf1[0]*Ax[4]+RRf1[1]*Ay[4]+RRf1[2]*Az[4];
float C16 = RRf1[0]*Ax[5]+RRf1[1]*Ay[5]+RRf1[2]*Az[5];
float Cx[] = {C11,C12,C13,C14,C15,C16};

float C21 = RRf2[0]*Ax[0]+RRf2[1]*Ay[0]+RRf2[2]*Az[0];
float C22 = RRf2[0]*Ax[1]+RRf2[1]*Ay[1]+RRf2[2]*Az[1];
float C23 = RRf2[0]*Ax[2]+RRf2[1]*Ay[2]+RRf2[2]*Az[2];
float C24 = RRf2[0]*Ax[3]+RRf2[1]*Ay[3]+RRf2[2]*Az[3];
float C25 = RRf2[0]*Ax[4]+RRf2[1]*Ay[4]+RRf2[2]*Az[4];
float C26 = RRf2[0]*Ax[5]+RRf2[1]*Ay[5]+RRf2[2]*Az[5];
float Cy[] = {C21,C22,C23,C24,C25,C26};

float C31 = RRf3[0]*Ax[0]+RRf3[1]*Ay[0]+RRf3[2]*Az[0];
float C32 = RRf3[0]*Ax[1]+RRf3[1]*Ay[1]+RRf3[2]*Az[1];
float C33 = RRf3[0]*Ax[2]+RRf3[1]*Ay[2]+RRf3[2]*Az[2];
float C34 = RRf3[0]*Ax[3]+RRf3[1]*Ay[3]+RRf3[2]*Az[3];
float C35 = RRf3[0]*Ax[4]+RRf3[1]*Ay[4]+RRf3[2]*Az[4];
float C36 = RRf3[0]*Ax[5]+RRf3[1]*Ay[5]+RRf3[2]*Az[5];
float Cz[] = {C31,C32,C33,C34,C35,C36};
```

```
//Matriz D de traslación
float                          Dx[]                          =
{pose[0],pose[0],pose[0],pose[0],pose[0],pose[0]};
float                          Dy[]                          =
{pose[1],pose[1],pose[1],pose[1],pose[1],pose[1]};
float                          Dz[]                          =
{pose[2],pose[2],pose[2],pose[2],pose[2],pose[2]};
```

El siguiente paso consiste en aplicar la ecuación de la distancia euclídea para obtener la longitud que existe entre los puntos de la posición actual y la deseada (calculados anteriormente por la cinemática inversa), la cual coincidirá con la longitud final que deberán tener los actuadores.

$$d_E(P,Q) = \sqrt{(p_1-q_1)^2 + (p_2-q_2)^2 + \cdots + (p_n-q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i-q_i)^2}.$$

Al resultado obtenido le restamos el tamaño que tienen los motores cuando están completamente retraídos (17.5 cm) para obtener la longitud total que se debe extender cada uno. Con esto obtendríamos las consignas que debemos enviar al controlador pero previamente se escala el resultado en un rango de 0 a 1023 (rango en el que trabaja el Arduino) y se asegura mediante la variable 'temp' la condición de no superar estos límites para evitar enviar una señal inadecuada que pueda suponer daños en los actuadores. También se ha añadido el almacenamiento de la posición inicial obtenida a través de la lectura de los potenciómetros.

```
//Se calcula la distancia euclídea entre la matriz obtenida y
la matriz B
float Fx[] ={pow(Cx[0]+Dx[0]-Bx[0],2),pow(Cx[1]+Dx[1]-Bx[1],2),
pow(Cx[2]+Dx[2]-Bx[2],2),pow(Cx[3]+Dx[3]-Bx[3],2),pow(Cx[4]+Dx[4]-
Bx[4],2),pow(Cx[5]+Dx[5]-Bx[5],2)};
float Fy[] ={pow(Cy[0]+Dy[0]-By[0],2),pow(Cy[1]+Dy[1]-
By[1],2),pow(Cy[2]+Dy[2]-By[2],2),pow(Cy[3]+Dy[3]-By[3],2),
pow(Cy[4]+Dy[4]-By[4],2),pow(Cy[5]+Dy[5]-By[5],2)};
float Fz[] ={pow(Cz[0]+Dz[0]-Bz[0],2),pow(Cz[1]+Dz[1]-
Bz[1],2),pow(Cz[2]+Dz[2]-Bz[2],2),pow(Cz[3]+Dz[3]-Bz[3],2),
pow(Cz[4]+Dz[4]-Bz[4],2),pow(Cz[5]+Dz[5]-Bz[5],2)};
float summat[] = {Fx[0]+Fy[0]+Fz[0],Fx[1]+Fy[1]+Fz[1],
Fx[2]+Fy[2]+Fz[2],Fx[3]+Fy[3]+Fz[3],Fx[4]+Fy[4]+Fz[4],Fx[5]
+ Fy[5]+Fz[5]};
//Se guardan en el vector L las longitudes calculadas de cada
motor float L[] =
{sqrt(summat[0]),sqrt(summat[1]),sqrt(summat[2]),
sqrt(summat[3]),sqrt(summat[4]), sqrt(summat[5])};

for (int k=0; k<6; k++){
      consigna1[k] = L[k]-17.5; //Resta entre las longitudes
calculadas y las longitudes físicas de los motores (17.5cm)
      tiemposMotores[k] = (consigna1[k]/3.2)*1.7; //Cálculo
```

```
de los tiempos del recorrido de cada motor (empleando velocidad máxima = 3.2cm/s)
      calcconsigna[k] = 1023*consigna1[k]/5.0; //Se escala la consigna obtenida entre 0 y 1023 que es el rango en el que trabaja Arduino
}

//Obtencion del tiempo del motor con mayor recorrido
float maximo = 0.0; //Variable qe almacenará el mayor tiempo de los 6 motores
for (int u=0; u<6; u++) { //Bucle que recorre el vector de tiempos
      if (maximo < tiemposMotores[u]) { //compara cada elemento con el valor de la variable maximo
      maximo = tiemposMotores[u];
      }
}
tiempo = maximo; //Se guarda el tiempo máximo

float temp = 0; //Variable que contemplará si existe alguna consigna no válida
for (int w=0; w<6; w++){
    if (calcconsigna[w]<0 || calcconsigna[w]>1023){ //Se comprueba que la consigna no supere los límites
      temp ++; //Si se cumple la condición anterior se registra incrementando el número de la variable temp
      Serial.println("Posición no válida");
      }
}

int temp0 = 0;

for(int e=0; e<6; e++){
    temp0 = analogRead(e+1); //Se lee las posiciones obtenidas de los potenciómetros de los motores
posicioninicial[e] = temp0*5.0/1023.0; //Se guarda la posición inicial de cada motor y se escalan entre 0 y 5 para convertirlo a cm reales
}

return(temp); //la función devuelve el valor temp
```

6.8.4 Función 'control'

Como se explicó anteriormente, esta función se está configurada como una interrupción que se ejecuta cada 3 milisegundos y es la parte del código encargada de realizar el control de cada motor, siendo éste el motivo por el cual se inicia la función con un bucle for con 6 iteraciones para evitar tener que repetir el mismo fragmento varias veces. Es importante aclarar que una interrupción tiene más prioridad que el resto del programa, por lo que nos permite ejecutar un determinado código, independientemente de lo que se esté ejecutando antes de dicha interrupción, para continuar por donde había quedado la ejecución tras terminar. Como se puede intuir el uso de esta interrupción periódica es ideal para realizar las medidas de posición de los motores y su control de manera estable.

En primer lugar se lee el potenciómetro para calcular la posición actual del motor, la cual será una señal de entrada analógica que mediante un conversor analógico-digital se convertirá en una señal digital comprendida en un rango entre 0 y 1023. Se realiza una conversión para obtener dicho valor escalado entre 0 y 5 con el objetivo de facilitar los cálculos posteriores. En este punto se pasa a calcular los comandos de salida diferenciando entre el controlador de posición y el de velocidad.

```
void control() {

int temp2;

for(int i=0; i<6; i++){ //Se emplea un bucle de 6 iteraciones
para evitar repetir el código para cada motor

temp2 = analogRead(i+1); //Se lee el valor de la entrada
analógica que nos proporcionará la posición del motor en un
rango entre 0 y 1023
posicion[i] = temp2*5.0/1023.0; //Se escala el valor
obtenido para trabajar en un rango entre 0 y 5 voltios
```

Para configurar el controlador de posición se prosigue de la siguiente manera: con el valor de consigna calculada mediante la función 'longitudes' y la posición leída ya es posible calcular el error existente que servirá como argumento para obtener el valor de la acción proporcional, integral y derivativa (aplicando las ecuaciones de cada una, comentadas en apartados anteriores) del PID. La acción integral se limita a valores comprendidos entre 25 y -25 con el objetivo de evitar su incremento indefinido a lo largo del tiempo. Finalmente se calcula el comando de PWM a aplicar uniendo la respuesta de las tres acciones anteriores y se almacena el valor del error actual en una variable que será empleado en la siguiente ejecución de la interrupción para el cálculo de la acción derivativa.

```
//Lazo de control para POSICIÓN (controlador PID)
error0pos[i] = consignapos[i] - posicion[i]; //Cálculo del
error entre la consigna deseada y la posición real que posee
```

```
el actuador
Ppos[i] = kppos*error0pos[i]; //Parte proporcional del
controlador PID de posición

if (fabs(salidaMotorPosiciones[i])<255){ //Si el comando
enviado a los motores es menor que 255 se aplica la parte
derivativa e integral
Dpos[i] = qdpos*(error0pos[i]-error1pos[i]); //Parte
derivativa del controlador PID de posición
Ipos[i] = Ipos[i]+qipos*error0pos[i]; //Parte integral del
controlador PID de posición
if (Ipos[i] >= 25){ //Si la parte integral supera el valor
de 25, esta se establece en dicho valor
Ipos[i] = 25;}
else if (Ipos[i] <= -25){ //Si la parte integral posee un
valor inferior a -25, esta se establece en dicho valor
Ipos[i] = -25;}
}

salidaMotorPosiciones[i] = Ppos[i]+Ipos[i]-Dpos[i]; //El
comando de PWM enviado a los motores será una combinación de
las partes anterior

error1pos[i] = error0pos[i]; //Se almacena el valor del
error calculado en la variable error1pos
```

Para configurar el controlador de velocidad es necesario obtener un parámetro adicional que será el tiempo empleado en el movimiento de los actuadores. Para ello se hace uso de la función 'millis' que devuelve como resultado el tiempo durante el cual se ha estado ejecutando el programa en milisegundos. La primera vez que se ejecuta la interrupción la plataforma aún no habrá comenzado a moverse y, por lo tanto, su velocidad valdrá 0. La segunda vez que se ejecute se calculará la velocidad que lleva el actuador empleando la fórmula que establece que la velocidad del mismo será el recorrido realizado por el motor entre el tiempo que ha tardado en recorrerlo. Como vendrá expresada en mm/ms se multiplica por 100 para trabajar en cm/s. Una vez que la velocidad es conocida, ya es posible determinar el error existente entre la velocidad que posee el actuador y la consigna, para emplearlo posteriormente en el cálculo de la acciones de control del PID y los comandos de PWM que serán enviados a cada motor.

```
//Lazo de control para VELOCIDAD(controlador PID)
tiempoLectActual = millis(); //Obtención del tiempo actual
de ejecución del programa

if (inicioInterrup == 0){ //En la primera interrupción que
se realice la velocidad de la plataforma valdrá 0 porque no
ha comenzado a moverse aún
```

```
velocidad[i] = 0;} else{
//La velocidad se calcula como el espacio recorrido por el
actuador entre el tiempo de empleado en recorrerlo
velocidad[i]    =    (((fabs(posicion[i]-posicioninicial[i]))
*10)/(tiempoLectActual));}  //Se obtiene la velocidad en
mm/ms
velocidad[i] = velocidad[i] * 100; //La velocidad obtenida
anteriormente se pasa a cm/s

error0vel[i] = consignavel[i] - velocidad[i]; //Cálculo del
error entre la consigna deseada y la velocidad actual del
motor
Pvel[i]  =  kpvel*error0vel[i];  //Parte proporcional del
controlador PID de velocidad

if (fabs(salidaMotorVelocidades[i])<255){
Dvel[i]    =    qdvel*(error0vel[i]-error1vel[i]);    //Parte
derivativa del controlador PID de velocidad
Ivel[i] = Ivel[i]+qivel*error0vel[i]; //Parte integral del
controlador PID de velocidad
if (Ivel[i] >= 25){ //Si la parte integral supera el valor
de 25, esta se establece en dicho valor
Ivel[i] = 15;}
else if (Ivel[i] <= -25){ //Si la parte integral posee un
valor inferior a -25, esta se establece en dicho valor
Ivel[i] = -15;}
}

salidaMotorVelocidades[i]  =  Pvel[i]+Ivel[i]-Dvel[i];  //El
comando de PWM enviado a los motores será una combinación de
las partes anterior

error1vel[i]  =  error0vel[i]; //Se almacena el valor del
error calculado en la variable error1vel
```

A la hora de enviar la PWM al actuador, se escogerá la más pequeña de los dos controladores para conseguir obtener una trayectoria suave y controlada. En primer lugar se contempla la condición de que la salida del controlador de posición sea menor que la salida del controlador de velocidad, en cuyo caso se comprueba si el comando es positivo o negativo para poner a nivel alto el pin conectado al puente en H que muevan el actuador en el sentido correcto. También se limita el nivel de salida entre 255 y -255 que son los valores máximos que admite una PWM, para finalmente enviar el nivel de consigna calculado a través del pin de enable. Se ha incluido la condición de que la salida a enviar sea 0 si el error se encuentra dentro de un determinado rango muy cerca del valor final, debido a que como está sujeto a una medida del potenciómetro siempre existirá un pequeño error que varía y por lo tanto nunca habrá un error completamente nulo.

```
//Se comparan las salidas de los dos controladores y se escoge la menor
if (fabs(salidaMotorPosiciones[i]) < salidaMotorVelocidades[i]){ //Si la salida del controlador de posición es menor que la de velocidad

if (error0pos[i] > -0.15 && error0pos[i] < 0.15){ //Si el error se encuentra dentro de un umbral cercano a la posición deseada no se envía señal
analogWrite (enable[i], 0);

else{
  if (salidaMotorPosiciones[i] > 0.0){ //Se establece el límite superior del comando de PWM a 255 (valor máximo admisible de PWM, ya que su rango es de 0 a 255)
  if (salidaMotorPosiciones[i] >= 255.0){salidaMotorPosiciones[i] = 255.0;
}
  digitalWrite (entrada2[i], LOW); //Se establecen las entradas del puente en H para que el motor se extienda
  digitalWrite (entrada1[i], HIGH);

  analogWrite (enable[i], round(fabs(salidaMotorPosiciones[i]))); //Se envía la salida calculada en valor absoluto a los motores
}
else {
  if (salidaMotorPosiciones[i] <= -255.0){ //Se establece el límite inferior del comando de PWM a -255
  salidaMotorPosiciones[i] = -255.0;
}
 digitalWrite (entrada2[i], HIGH); //Se establecen las entradas del puente en H para que el motor se contraiga
  digitalWrite (entrada1[i], LOW);
  analogWrite (enable[i], round(fabs(salidaMotorPosiciones[i]))); //Se envía la salida calculada en valor absoluto a los motores
      }
    }
}
```

La otra posibilidad que se contempla es que la salida del controlador de velocidad sea menor que la salida del controlador de posición. En este caso, se comprobará que dicha salida sea mayor que 0 (no se comprueba que sea menor debido a

que la velocidad siempre se considerará positiva) y se limita, al igual que en el caso anterior, a un máximo de PWM de 255. Como no tenemos ninguna manera de saber el sentido del motor atendiendo solamente a la velocidad, para determinarlo se hace uso del error calculado para el controlador de posición, dado que si este es positivo significará que el motor debe extenderse y si es negativo que debe contraerse. Consultando este valor se pone a nivel alto el pin del puente en H correspondiente y finalmente se envían los comandos calculados a través del pin de enable.

```
if (fabs(salidaMotorPosiciones[i]) > salidaMotorVelocidades[i]){ //Si la salida del controlador de posición es menor que la de velocidad

if (error0pos[i] > -0.15 && error0pos[i] < 0.15){ //Si el error se encuentra dentro de un umbral cercano a la posición deseada no se envía señal
analogWrite (enable[i], 0);}
else{
   if (salidaMotorVelocidades[i] > 0.0){ //Se establece el límite superior del comando de PWM a 255
      if (salidaMotorVelocidades[i] >= 255.0){
salidaMotorVelocidades[i] = 255.0;}
if (error0pos[i] > 0){ //Si el error de posición es mayor que 0 se establecen las entradas del puente en H para que el motor se extienda
digitalWrite (entrada2[i], LOW);
digitalWrite (entrada1[i], HIGH);}
if (error0pos[i] < 0){ //Si el error de posición es menor que 0 se establecen las entradas del puente en H para que el motor se contraiga
  digitalWrite (entrada2[i], HIGH);
   digitalWrite (entrada1[i], LOW);}
   analogWrite
(enable[i],round(fabs(salidaMotorVelocidades[i]))); //Se envía la salida calculada en valor absoluto a los motores
      }
    }
  }
}
inicioInterrup = inicioInterrup + 1; //Se aumenta en 1 el valor de la variable para tener constancia que ya se ha realizado la primera interrupción
}
```

6.8.5 Función 'loop'

Esta función tiene un comportamiento cíclico y se encarga de enviar las consignas calculadas mediante la función 'longitudes' si el temp devuelto es 0 (lo cual indica que no existen errores) a la función 'control' para el cálculo de las acciones del controlador PID. Para el caso del controlador de posición las consignas son enviadas directamente, pero para el controlador de velocidad primero se calculan dichas consignas de velocidad realizando el cociente entre el recorrido que debe hacer cada motor y el tiempo que estimemos oportuno para completar el recorrido (en nuestro caso hemos elegido un tiempo que coincide con el doble de lo que tardaría en hacerse el recorrido a la velocidad máxima del motor). Una vez calculadas se comprueba que están en el rango comprendido entre 0 y 3.5cm/s (que es la velocidad máxima que puede alcanzar el actuador) y si finalmente todo es correcto y el temp3 tiene un valor de 0, entonces se envían las consignas calculadas al controlador de velocidad. Este ciclo de envío de consignas sólo se ejecuta una vez en la ejecución del programa y para conseguirlo se hace uso de la variable "inicio" que es incrementada cuando acaba este paso.

```
void loop(){
if (inicio == 0){ //El cálculo de las consignas se ejecuta
sólo una vez  para aligerar el procesado
  int temp = longitudes();
  if (temp==0){ //Se comprueba que la consigna no supera los
límites establecidos con la variable temp
//Se envían las consignas calculadas al controlador de
posición
   for (int i=0; i<6; i++){
        consigna[i] = calcconsigna[i]*5.0/1023;
      }
    }
  int temp3 = 0;
  float consigna2[6] = {0,0,0,0,0,0};

  for (int g=0; g<6; g++){
    consigna2[g] = consigna1[g]/tiempo; //La consigna de
velocidad será el recorrido que tiene que hacer cada motor
entre el tiempo empleado en recorrerlo
    if (consigna2[g]<0 || consigna2[g]>3.3){ //Si la
consigna no se cuentra entre 0 y 3.3cm/s (velocidad máxima de
los motores) se aumenta temp3
     temp3 ++;
     }
   }
  if (temp3==0){ //Se comprueba que la consigna no supera
los límites establecidos con la variable temp3
   for (int t=0; t<6; t++){
     consignavel[t] = consigna2[t]; //Se envían las
consignas calculadas  al controlador de velocidad
```

```
    }
  }
  inicio = inicio + 1; //Se aumenta en 1 la variable para
reflejar que ya  se ha efectuado el cálculo de las consignas
   }
 }
```

7. Conclusiones del Trabajo de Fin de Grado

La realización del presente proyecto ha supuesto una oportunidad para dotar de un enfoque práctico a numerosos conceptos adquiridos a lo largo del grado. Su ejecución ha servido como un método para comenzar a adquirir las destrezas necesarias que exige la ingeniería cuando nos enfrentamos a un proyecto como este, que podría llegar a ser perfectamente un caso real en el mundo laboral. También ha sido una manera de involucrar de manera práctica la gestión y organización del trabajo en grupo, ya que en nuestro caso hemos realizado el trabajo los dos autores de forma conjunta, contando con la ayuda de los dos tutores, los cuales poseen grandes conocimientos en la materia y su colaboración nos ha aportado una experiencia extra a añadir dentro del proceso.

Algunos de los conceptos y habilidades empleados durante la realización del trabajo fueron:

• Conocimiento de los aspectos básicos del ámbito de la robótica y los métodos empleados para el posicionamiento de robots.

• Capacidad de comprensión y estudio de un proyecto base del que parte el actual trabajo, con el objetivo de poder trabajar sobre él para llevar a cabo las mejoras necesarias planteadas.

• Conocimientos básicos sobre algunos lenguajes de programación, así como el estudio y comprensión de otros no empleados a lo largo del grado, además de la capacidad para traducir el comportamiento del robot deseado a dichos lenguajes.

• Estudio de aspectos de robótica de mayor complejidad como pueden ser el espacio de trabajo, las cadenas cinemáticas en robots paralelos, las trayectorias de los actuadores, etc.

• Uso y programación de placas electrónicas (en nuestro caso Arduino) que controlan el comportamiento de la plataforma robótica.

• Búsqueda de información y documentación necesaria para lograr los objetivos planteados en el proyecto.

• Capacidad para resolver de manera óptima y eficaz diversos conflictos generados durante el desarrollo.

8. Referencias

- **Robótica:**

http://www.unicauca.edu.co/ai/publicaciones/ISAShow_Vivas.pdf

http://es.slideshare.net/htrmoreno/robots-paralelos

https://en.wikipedia.org/wiki/Stewart_platform

http://dspace.ups.edu.ec/bitstream/123456789/1921/14/UPS-CT002354.pdf

http://upcommons.upc.edu/bitstream/handle/2099.1/12947/PFC.pdf

http://itzamna.bnct.ipn.mx/dspace/bitstream/123456789/12716/1/1948%202013.pdf

http://sedici.unlp.edu.ar/bitstream/handle/10915/20558/Documento_completo.pdf?sequence=1

- **Arduino:**

http://sabetecnologia.blogspot.com.es/2013/03/tabla-comparativa-de-arduinos.html

https://es.wikipedia.org/wiki/Arduino

http://booleanbite.com/web/adquisicion-de-datos-con-arduino-i-tiempo-de-muestreo-y-resolucion/

http://www.prometec.net/multiplexor-74hc4067/

http://frenki.net/2013/10/fast-analogread-with-arduino-due/

http://www.nagavenkat.adurthi.com/?p=631

https://github.com/ivanseidel/DueTimer

https://www.tr3sdland.com/2012/11/llega-arduino-due-con-arquitectura-de-32-bit/

http://www.microsmart.co.za/technical/2014/03/01/advanced-arduino-adc/

https://www.youtube.com/watch?v=g0pSfyXOXj8

http://forum.arduino.cc/index.php?topic=68771.0

http://matthewsfolbigg.xdesigns.com.au/matthews-folbigg-people/stewart-gough

- **Datasheet:**

http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf

http://www.atmel.com/Images/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf

- **Matlab:**

  http://es.mathworks.com/index.html?s_tid=gn_logo

- **Otras:**

  http://angeljohnsy.blogspot.com/2013/05/converting-rgb-image-to-hsi.html

  http://colorizer.org/

- Antonio Luis Morell González, Iván Daniel Peraza Rocha. "Diseño e Implementación de un Sistema de Estabilización Activa para el Sistema de Cámaras del Prototipo Verdino". Julio, 2010. Proyecto fin de carrera para la titulación de Ingeniería en Automática y Electrónica Industrial. Universidad de La Laguna.

- Kevin Mínguez Olivero. "Sistema de Posicionamiento XYZ". Septiembre, 2014. Proyecto fin de carrera para la titulación de Grado en Ingeniería Industrial Electrónica y Automática. Universidad de La Laguna.

    - http://riull.ull.es/xmlui/handle/915/349

9. Anexos

Anexo 1

9.1 Código Matlab: Espacio de Trabajo

```
%Matrices de coordenadas de la base móvil y fija respectivamente.
A = single([5 5.8 0.7 -0.4 -5.2 -4.3;3.6 2.7 -5.6 -5.8 3.4 3.9;
            0 0 0 0 0 0]);
B = single([1.2 8.8 7.3 -7.3 -8.8 -1.2; 9.3 -4.6 -6.5 -6.5 -4.6 9.3;
            0 0 0 0 0 0]);

%Introducción de los valores por teclado.
xpi     = input('Ingrese el valor inicial de X: ');
xpf     = input('Ingrese el valor final de X: ');
ypi     = input('Ingrese el valor inicial de Y: ');
ypf     = input('Ingrese el valor final de Y: ');
zpi     = input('Ingrese el valor inicial de Z: ');
zpf     = input('Ingrese el valor final de Z: ');
ppi     = input('Ingrese el valor inicial de PITCH: ');
ppf     = input('Ingrese el valor final de PITCH: ');
rpi     = input('Ingrese el valor inicial de ROLL: ');
rpf     = input('Ingrese el valor final de ROLL: ');
wpi     = input('Ingrese el valor inicial de YAW: ');
wpf     = input('Ingrese el valor final de YAW: ');
N       = 7;

%Declaración de variables, D1, D2 y D3 son los step de los for de
Pitch,
%Roll y Yaw.
D1 = ((abs(ppi)+ppf)/N);
D2 = ((abs(rpi)+rpf)/N);
D3 = ((abs(wpi)+wpf)/N);

%Inicialización de las variables para realizar la concatenación.
a=single([]);
b=single([]);
c=single([]);
d=single([]);
e=single([]);
f=single([]);

%Parte que se encarga de concatenar los valores introducidos por
teclado.
for i=xpi:0.5:xpf
    for j=ypi:0.5:ypf
         for k=zpi:0.5:zpf
            for l=ppi:D1:ppf
                for m=rpi:D2:rpf
                    for n=wpi:D3:wpf

                        a = [a i ];
                        b = [b j ];
                        c = [c k ];
                        d = [d l ];
                        e = [e m ];
                        f = [f n ];

                        P=single([a' b' c' d' e' f']);

                    end
                end
             end
        end
```

```
    end
end

%fip, nº de filas, cop, nº de columnas de la matriz P.
[fip,cop] = size(P);


L1=single([]);

%Cinemática inversa.
for i=1:fip

    pose = P(i,:)';
    A2 = A(1:3,:);
    B2 = B(1:3,:);
    R = pose(4:6,:);
    D = pose(1:3,:);
    Rrad  = (R.*pi)./180;
    cphi  = cos(Rrad(1));
    ctita = cos(Rrad(2));
    cpsi  = cos(Rrad(3));
    sphi  = sin(Rrad(1));
    stita = sin(Rrad(2));
    spsi  = sin(Rrad(3));
    sphi_stita = sphi*stita;
    cphi_stita = cphi*stita;

    RR=[ctita*cpsi+sphi_stita*spsi -ctita*spsi+sphi_stita*cpsi
cphi_stita ;
        cphi*spsi                   cphi*cpsi                   -sphi
;
        -stita*cpsi+sphi*ctita*spsi stita*spsi+sphi*ctita*cpsi
cphi*ctita];

        L1 = [L1 sqrt(sum((RR * A2 + D(:,ones(6,1)) - B2).^2))'];


end

L2 = L1';


AA=single([]);
LL=single([]);

%Separación de las poses en celdas para ser evaluadas.
for i=1:fip
    AA{i} = L2(i,:);
    LL{i} = P(i,:);

end

%Conversión de la pose.
for i = 1:fip
    for j=1:6
        AA{i}(j) = AA{i}(j)-17.5;
        AA{i}(j) = 1024*AA{i}(j)/5;

    end
end
```

```
temp = zeros(fip,6);
w = 0;

%Filtro, sólo la pose con temp = 0 es aceptada.
for i=1:fip
    for j = 1:6
        if (AA{i}(j)<0 || AA{i}(j)>1023)
            temp(i,j)=temp(i,j)+1;

        end
    end
end


RTY=single([]);

%Si la suma del vector temp = 0, la pose es aceptada.
for rty=1:fip
    if sum(temp(rty,:)) == 0
        RTY = [RTY LL{rty}' ];
    end
end

M=RTY';

%Valores máximos y mínimos de cada variable de pose.
minX = min(M(:,1));
maxX = max(M(:,1));

minY = min(M(:,2));
maxY = max(M(:,2));

minZ = min(M(:,3));
maxZ = max(M(:,3));

minP = min(M(:,4));
maxP = max(M(:,4));

minR = min(M(:,5));
maxR = max(M(:,5));

minYW = min(M(:,6));
maxYW = max(M(:,6));


SM4=single([]);

%Código que separa la matriz de poses aceptadas en submatrices con un
mismo
%valor para cada columna.

   for i2=minX:0.5:maxX
       for j2=minY:0.5:maxY
           for k2 =minZ:0.5:maxZ

                [fi,co]=find(M(:,1)== i2);

                fimx = max(fi);
                fimn = min(fi);
```

```
                comx = max(co);
                comn = min(co);

                SM = M(fimn:fimx,:);


                [fi2,co2]=find(SM(:,2)== j2);

                fimx2 = max(fi2);
                fimn2 = min(fi2);

                comx2 = max(co2);
                comn2 = min(co2);

                SM2 = SM(fimn2:fimx2,:);


                [fi3,co3]=find(SM2(:,3)== k2);

                fimx3 = max(fi3);
                fimn3 = min(fi3);

                comx3 = max(co3);
                comn3 = min(co3);

                SM3 = SM2(fimn3:fimx3,:);

                [tt,yy] = size(SM3);

                %Llegados a este punto podemos encontrarnos que la
                %submatriz SM3 esté formada por una sola fila, de ser
así
                %se concatena esa única fila. Por el contrario, el
proceso
                %se ejecuta como antes, buscando el máximo valor para
cada
                %ángulo.

                if tt == 1
                    SM4 = [SM4 SM3'];

                else
                    SM4 = [SM4 max(SM3)'];
                end

            end
        end
    end

    YRT = SM4';


[pt1,pt2] = size(SM4');

X1=[];
Y1=[];
Z1=[];
RJ1=[];
VR1=[];
AZ1=[];
```

```
%De 1 a 10, se crean los puntos XYZ y los colores de P,R,Y para la
gráfica.
for l=1:pt1

    X1 = [X1 YRT(l,1)];
    Y1 = [Y1 YRT(l,2)];
    Z1 = [Z1 YRT(l,3)];

    RJ1 = [RJ1 ((max(abs(P(:,4))-abs(YRT(l,4))))/max(abs(P(:,4))))];
    VR1 = [VR1 ((max(abs(P(:,5))-abs(YRT(l,5))))/max(abs(P(:,5))))];
    AZ1 = [AZ1 ((max(abs(P(:,6))-abs(YRT(l,6))))/max(abs(P(:,6))))];

end


%Se concatenan las 10 matrices creadas anteriormente, es el paso
previo
%a graficar.
XF1 = [X1 X2 X3 X4 X5 X6 X7 X8 X9 X10];
YF1 = [Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8 Y9 Y10];
ZF1 = [Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10];

RJF1 = [RJ1 RJ2 RJ3 RJ4 RJ5 RJ6 RJ7 RJ8 RJ9 RJ10];
VRF1 = [VR1 VR2 VR3 VR4 VR5 VR6 VR7 VR8 VR9 VR10];
AZF1 = [AZ1 AZ2 AZ3 AZ4 AZ5 AZ6 AZ7 AZ8 AZ9 AZ10];


[pt1,pt2] = size(XF1);

figure;

%Código que genera la gráfica.
for i=1:pt2


plot3(XF1(i),YF1(i),ZF1(i),'mo','MarkerEdgeColor','k','MarkerFaceColor
',...
          [RJF1(i) VRF1(i) AZF1(i)],'LineWidth',1,'MarkerSize',10);
    hold on;
    xlabel('Eje X');
    ylabel('Eje Y');
    zlabel('Eje Z');
    axis([-8,7,-8,6,15.5,22])
end

view(30,10)
```

Anexo 2

9.2 Código Matlab: Simulación

```
%Matrices de coordenadas de los vértices de la base móvil y de la base
%fija, respectivamente.
A = [-1.5 -6.5 -5 5 6.5 1.5; -5.628 3.03 5.628 5.628 3.03 -5.628;
    0 0 0 0 0 0;1 1 1 1 1 1];

B = [-8.5 -11 -2.5 2.5 11 8.5;-11.69 -7.36 7.362 7.362 -7.36 -11.69;
    0 0 0 0 0 0;1 1 1 1 1 1];

X0 = [B(1,1) B(1,2) B(1,3) B(1,4) B(1,5) B(1,6) B(1,1)];
Y0 = [B(2,1) B(2,2) B(2,3) B(2,4) B(2,5) B(2,6) B(2,1)];
Z0 = [B(3,1) B(3,2) B(3,3) B(3,4) B(3,5) B(3,6) B(3,1)];

%Coordenadas de los anclajes entre la base móvil y la base fija.
A3 = [5 5.8 0.7 -0.4 -5.2 -4.3;3.6 2.7 -5.6 -5.8 3.4 3.9; 0 0 0 0 0
0];
B3 = [1.2 8.8 7.3 -7.3 -8.8 -1.2; 9.3 -4.6 -6.5 -6.5 -4.6 9.3; 0 0 0 0
0 0];

%Inicialización de las variables que conforman la matriz pose3.
XX=[];
YY=[];
ZZ=[];
PP=[];
ROO=[];
YWW=[];

%Introducción por teclado de la posición deseada.
fprintf('\n');
X = input('Introduce el Valor de X: ');
Y = input('Introduce el Valor de Y: ');
Z = input('Introduce el Valor de Z: ');
PI = input('Introduce el Valor de PITCH: ');
RO = input('Introduce el Valor de ROLL: ');
YW = input('Introduce el Valor de YAW: ');
fprintf('\n');

%Variable que propicia el movimiento desde la última posición dada.
%pose2, concatenación de las variables de posición.
last_pose = pose;
pose2 = [ X Y Z PI RO YW ]';

%Cinemática inversa.
A3 = A3(1:3,:);
B3 = B3(1:3,:);
R = pose2(4:6,:);
D = pose2(1:3,:);
Rrad  = (R.*pi)./180;
cphi  = cos(Rrad(1));
ctita = cos(Rrad(2));
cpsi  = cos(Rrad(3));
sphi  = sin(Rrad(1));
stita = sin(Rrad(2));
spsi  = sin(Rrad(3));
sphi_stita = sphi*stita;
cphi_stita = cphi*stita;

RR = [ ctita*cpsi+sphi_stita*spsi -ctita*spsi+sphi_stita*cpsi
cphi_stita ;
```

```
       cphi*spsi                    cphi*cpsi                    -sphi
;
       -stita*cpsi+sphi*ctita*spsi stita*spsi+sphi*ctita*cpsi
cphi*ctita];

L1 = sqrt(sum((RR * A3 + D(:,ones(6,1)) - B3).^2));

%Conversión de los valores entre 0 y 1023.
for i = 1:6
    L1(i) = L1(i)-17.5;
    L1(i) = 1024*L1(i)/5
end

temp = 0;

%Bucle de seguridad.
for i = 1:6
    if (L1(i)<0 || L1(i)>1023)
        temp=temp+1;
    end
end

%Si MAX == 0 se utiliza la variable last_MAX para el tamaño que la
matriz
%unitaria o para los steps del bucle for.
if temp == 0
    pose = pose2;
    fprintf('Vector de Posición aceptado \n')
    pose(3)=pose(3)-17;

    last_MAX = MAX;
    MAX=max(abs(pose(:)));

    %Si la última posición coincide con la actual se genera una matriz
    %con el mismo valor, ones*pose(i).
    if MAX == 0

        if last_pose(1) == pose(1)

            XX = ones(1,last_MAX+1)*pose(1);

        else

    %Si la posición es diferente a la anterior, se crea una matriz que
    %itera desde la posición anterior hasta la actual.

            for i=last_pose(1):((pose(1)-
last_pose(1))/last_MAX):pose(1)

                XX = [XX i];

            end

        end

        if last_pose(2) == pose(2)

            YY = ones(1,last_MAX+1)*pose(2);

        else
```

```
            for i=last_pose(2):((pose(2)-last_pose(2))/last_MAX):pose(2)

                YY = [YY i];

            end

        end

        if last_pose(3) == pose(3)

            ZZ = ones(1,last_MAX+1)*pose(3)+17.1;

        else

            for i=last_pose(3):((pose(3)-
last_pose(3))/last_MAX):pose(3)

                ZZ = [ZZ i+17.1];

            end

        end

        if last_pose(4) == pose(4)

            PP = ones(1,last_MAX+1)*pose(4);

        else


            for i=last_pose(4):((pose(4)-
last_pose(4))/last_MAX):pose(4)

                PP = [PP i];

            end

        end

        if last_pose(5) == pose(5)

            ROO = ones(1,last_MAX+1)*pose(5);

        else

            for i=last_pose(5):((pose(5)-
last_pose(5))/last_MAX):pose(5)

                ROO = [ROO i];

            end

        end

        if last_pose(6) == pose(6)

            YWW = ones(1,last_MAX+1)*pose(6);
```

```
        else

            for i=last_pose(6):((pose(6)-
last_pose(6))/last_MAX):pose(6)

                YWW = [YWW i];

            end

        end

 %Si MAX =! 0, se utiliza la variable MAX para el tamaño que la matriz
 %unitaria o para los steps del bucle for.
    else

        if last_pose(1) == pose(1)

            XX = ones(1,MAX+1)*pose(1);

        else

            for i=last_pose(1):((pose(1)-last_pose(1))/MAX):pose(1)

                XX = [XX i];

            end

        end

        if last_pose(2) == pose(2)

            YY = ones(1,MAX+1)*pose(2);

        else

            for i=last_pose(2):((pose(2)-last_pose(2))/MAX):pose(2)

                YY = [YY i];

            end

        end

        if last_pose(3) == pose(3)

            ZZ = ones(1,MAX+1)*pose(3)+17.1;

        else

            for i=last_pose(3):((pose(3)-last_pose(3))/MAX):pose(3)

                ZZ = [ZZ i+17.1];

            end

        end
```

```
    if last_pose(4) == pose(4)

        PP = ones(1,MAX+1)*pose(4);

    else

        for i=last_pose(4):((pose(4)-last_pose(4))/MAX):pose(4)

            PP = [PP i];

        end

    end

    if last_pose(5) == pose(5)

        ROO = ones(1,MAX+1)*pose(5);

    else

        for i=last_pose(5):((pose(5)-last_pose(5))/MAX):pose(5)

            ROO = [ROO i];

        end

    end

    if last_pose(6) == pose(6)

        YWW = ones(1,MAX+1)*pose(6);

    else

        for i=last_pose(6):((pose(6)-last_pose(6))/MAX):pose(6)

            YWW = [YWW i];

        end

    end
end

 %Se concatenan los resultados en pose3 para realizar los cálculos
 %de la simulación.
 pose3 = [XX' YY' ZZ' PP' ROO' YWW'];

%En el caso de que MAX == 0, se utiliza last_MAX para la iteración.
if MAX == 0

    for i = 1:1:(last_MAX+1)

        pose4=pose3(i,:)';
        R = pose4(4:6,:);

        Rrad  = (R.*pi)./180;
        cphi  = cos(Rrad(1));
        ctita = cos(Rrad(2));
        cpsi  = cos(Rrad(3));
```

```
            sphi  = sin(Rrad(1));
            stita = sin(Rrad(2));
            spsi  = sin(Rrad(3));
            sphi_stita = sphi*stita;
            cphi_stita = cphi*stita;

            RR2 = [ ctita*cpsi+sphi_stita*spsi -
ctita*spsi+sphi_stita*cpsi cphi_stita pose4(1);
                    cphi*spsi                  cphi*cpsi
-sphi      pose4(2);
                    -stita*cpsi+sphi*ctita*spsi
stita*spsi+sphi*ctita*cpsi cphi*ctita pose4(3);
                    0 0 0 1];

             A2 = RR2*A;

             X2 = [A2(1,1) A2(1,2) A2(1,3) A2(1,4) A2(1,5) A2(1,6)
A2(1,1)];
             Y2 = [A2(2,1) A2(2,2) A2(2,3) A2(2,4) A2(2,5) A2(2,6)
A2(2,1)];
             Z2 = [A2(3,1) A2(3,2) A2(3,3) A2(3,4) A2(3,5) A2(3,6)
A2(3,1)];


             XP1 = [B(1,1) A2(1,1)];
             YP1 = [B(2,1) A2(2,1)];
             ZP1 = [B(3,1) A2(3,1)];

             XP2 = [B(1,2) A2(1,2)];
             YP2 = [B(2,2) A2(2,2)];
             ZP2 = [B(3,2) A2(3,2)];

             XP3 = [B(1,3) A2(1,3)];
             YP3 = [B(2,3) A2(2,3)];
             ZP3 = [B(3,3) A2(3,3)];

             XP4 = [B(1,4) A2(1,4)];
             YP4 = [B(2,4) A2(2,4)];
             ZP4 = [B(3,4) A2(3,4)];

             XP5 = [B(1,5) A2(1,5)];
             YP5 = [B(2,5) A2(2,5)];
             ZP5 = [B(3,5) A2(3,5)];

             XP6 = [B(1,6) A2(1,6)];
             YP6 = [B(2,6) A2(2,6)];
             ZP6 = [B(3,6) A2(3,6)];


             pause(0.1);
             plot3(X2,Y2,Z2,'r',X0,Y0,Z0,'b',XP1,YP1,ZP1,'-
ok',XP2,YP2,ZP2,...
                   '-ok',XP3,YP3,ZP3,'-ok',XP4,YP4,ZP4,'-
ok',XP5,YP5,ZP5,...
                   '-ok',XP6,YP6,ZP6,'-ok');

             hold off
             ylabel('eje y')
             zlabel('eje z')
             xlabel('eje x')
             axis([-25,25,-25,25,0,30]);
```

```
            axis square;
            grid off

        end

    %Si MAX =! 0, se utiliza la variable MAX para la iteración.
    else

        for i = 1:1:(MAX+1)

            pose4 = pose3(i,:)';
            R = pose4(4:6,:);

            Rrad  = (R.*pi)./180;
            cphi  = cos(Rrad(1));
            ctita = cos(Rrad(2));
            cpsi  = cos(Rrad(3));
            sphi  = sin(Rrad(1));
            stita = sin(Rrad(2));
            spsi  = sin(Rrad(3));
            sphi_stita = sphi*stita;
            cphi_stita = cphi*stita;

            RR2 = [ ctita*cpsi+sphi_stita*spsi -
ctita*spsi+sphi_stita*cpsi cphi_stita pose4(1) ;
                    cphi*spsi                  cphi*cpsi
-sphi      pose4(2);
                    -stita*cpsi+sphi*ctita*spsi
stita*spsi+sphi*ctita*cpsi cphi*ctita pose4(3);
                    0 0 0 1];

             A2 = RR2*A;

             X2 = [A2(1,1) A2(1,2) A2(1,3) A2(1,4) A2(1,5) A2(1,6)
A2(1,1)];
             Y2 = [A2(2,1) A2(2,2) A2(2,3) A2(2,4) A2(2,5) A2(2,6)
A2(2,1)];
             Z2 = [A2(3,1) A2(3,2) A2(3,3) A2(3,4) A2(3,5) A2(3,6)
A2(3,1)];


             XP1 = [B(1,1) A2(1,1)];
             YP1 = [B(2,1) A2(2,1)];
             ZP1 = [B(3,1) A2(3,1)];

             XP2 = [B(1,2) A2(1,2)];
             YP2 = [B(2,2) A2(2,2)];
             ZP2 = [B(3,2) A2(3,2)];

             XP3 = [B(1,3) A2(1,3)];
             YP3 = [B(2,3) A2(2,3)];
             ZP3 = [B(3,3) A2(3,3)];

             XP4 = [B(1,4) A2(1,4)];
             YP4 = [B(2,4) A2(2,4)];
             ZP4 = [B(3,4) A2(3,4)];

             XP5 = [B(1,5) A2(1,5)];
             YP5 = [B(2,5) A2(2,5)];
             ZP5 = [B(3,5) A2(3,5)];
```

```
            XP6 = [B(1,6) A2(1,6)];
            YP6 = [B(2,6) A2(2,6)];
            ZP6 = [B(3,6) A2(3,6)];


            pause(0.1);
            plot3(X2,Y2,Z2,'r',X0,Y0,Z0,'b',XP1,YP1,ZP1,'-
ok',XP2,YP2,ZP2,...
                  '-ok',XP3,YP3,ZP3,'-ok',XP4,YP4,ZP4,'-
ok',XP5,YP5,ZP5,...
                  '-ok',XP6,YP6,ZP6,'-ok');


            hold off
            ylabel('eje y')
            zlabel('eje z')
            xlabel('eje x')
            axis([-25,25,-25,25,0,30]);
            axis square;
            grid off

       end

   end

else

    fprintf('\n Vector de posición no aceptado \n');

end
```

Anexo 3

9.3 Código Arduino

```
#include "DueTimer.h" //Se incluye la librería específica para
controlar los  timer de Arduino Due

int entrada1[6]={48,44,40,36,32,28}; //Vector que define los pines
de la entrada 1 de los puentes en H de cada motor
int entrada2[6]={49,45,41,37,33,29}; //Vector que define los pines
de la entrada 2 de los puentes en H de cada motor
int enable[6]={7,6,5,4,3,2}; //Vector que define los pines de
enable por el cual  se enviarán los comandos de PWM a cada motor

//Variables para el controlador:
float kppos = 180; //Parte proporcional del controlador
PID de posición
float qipos = 0.1; //Parte integral del controlador PID
de posición
float qdpos = 180; //Parte derivativa del controlador
PID de posición
float kpvel = 90.0; //Parte proporcional del controlador
PID de velocidad
float qivel = 0.1; //Parte integral del controlador PID
de velocidad
float qdvel = 90.0; //Parte derivativa del controlador
PID de velocidad
float Ppos[6] = {0,0,0,0,0,0}; //Vector que almacena la acción
proporcional del  PID de posición calculada para cada motor
float Ipos[6]={0,0,0,0,0,0}; //Vector que almacena la acción
integral del PID de  posición calculada para cada motor
float Dpos[6]={0,0,0,0,0,0}; //Vector que almacena la acción
derivativa del PID  de posición calculada para cada motor
float Pvel[6] = {0,0,0,0,0,0}; //Vector que almacena la acción
proporcional del  PID de velocidad calculada para cada motor
float Ivel[6]={0,0,0,0,0,0}; //Vector que almacena la acción
integral del PID de  velocidad calculada para cada motor
float Dvel[6]={0,0,0,0,0,0}; //Vector que almacena la acción
derivativa del PID  de velocidad calculada para cada motor

float consignapos[6]={2.5,2.5,2.5,2.5,2.5,2.5}; //Vector que
contiene el valor de  las consignas de posición de cada motor
float consignavel[6]={0,0,0,0,0,0}; //Vector que contiene el
valor de las  consignas de velocidad de cada motor
```

```
float posicion[6] = {0,0,0,0,0,0}; //Vector que contiene el valor
leído de  posición de cada motor
float velocidad[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor calculado de  velocidad de cada motor

float error0pos[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error  calculado de posición de cada motor
float error1pos[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error  anterior de posición de cada motor
float error0vel[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error  calculado de velocidad de cada motor
float error1vel[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor del error  anterior de velocidad de cada motor

float salidaMotorPosiciones[6]={0,0,0,0,0,0}; //Vector que
contiene los comandos  de PWM obtenidos del control de posición
float salidaMotorVelocidades[6]={0,0,0,0,0,0}; //Vector que
contiene los comandos  de PWM obtenidos del control de velocidad

float tiemposMotores[6]={0,0,0,0,0,0}; //Vector que contiene los
tiempos  necesarios en los que cada motor debe realizar su
recorrido
float tiempo = 0.0; // Variable que guarda el tiempo que tarda el
motor que  realiza el mayor recorrido

float calcconsigna[6] = {0,0,0,0,0,0}; //Vector que contiene el
valor de las  consignas de posición escaladas entre 0 y 1024
float consigna1[6] = {0,0,0,0,0,0}; //Vector que contiene las
consignas de  posición  en  cm
float posicionanterior[6] = {0,0,0,0,0,0}; //Vector que guarda la
posición del  instante anterior
float posicioninicial[6] = {0,0,0,0,0,0}; //Vector que guarda las
posiciones  leídas al comenzar el programa

unsigned long tiempoLectActual = 0.0; //Variable que almacena el
tiempo en el que  se realiza cada lectura de la posición

int inicio = 0; //Variable que indica si es la primera vez que se
ejecuta el  programa
int inicioInterrup = 0; //Variable que indica si es la primera vez
que se ejecuta  la interrupción

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,//

void setup() {
```

```
  REG_ADC_MR = (REG_ADC_MR & 0xFFF0FFFF) | 0x00020000; //Disminuye
tiempo de  conversión desde 36.6us hasta 3.97us

Serial.begin(9600); //Se abre el puerto serie con una velocidad de
9600 bps
  //Se configuran como pines de salida de la placa las entradas y
enable de cada motor
pinMode(entrada1[0],OUTPUT);
pinMode(entrada2[0],OUTPUT);
pinMode(enable[0],OUTPUT);

pinMode(entrada1[1],OUTPUT);
pinMode(entrada2[1],OUTPUT);
pinMode(enable[1],OUTPUT);

pinMode(entrada1[2],OUTPUT);
pinMode(entrada2[2],OUTPUT);
pinMode(enable[2],OUTPUT);

pinMode(entrada1[3],OUTPUT);
pinMode(entrada2[3],OUTPUT);
pinMode(enable[3],OUTPUT);

pinMode(entrada1[4],OUTPUT);
pinMode(entrada2[4],OUTPUT);
pinMode(enable[4],OUTPUT);

pinMode(entrada1[5],OUTPUT);
pinMode(entrada2[5],OUTPUT);
pinMode(enable[5],OUTPUT);
  //Se configuran los pines de entradas analógicas encargados de
leer los  potenciómetros de los motores
pinMode(A1,INPUT);
pinMode(A2,INPUT);
pinMode(A3,INPUT);
pinMode(A4,INPUT);
pinMode(A5,INPUT);
pinMode(A6,INPUT);
//Se establece la función control como una interrupción que se
ejecutará cada 3 ms.
Timer3.attachInterrupt(control).start(3000);

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,//

int longitudes() //Función que calcula la cinemática inversa y las
consignas de  los motores
```

```
{
float pose[] = {0,0,19,0,0,0}; //Vector que contiene la pose que
se desea alcanzar
//Matriz con los puntos de referencia de la base móvil
 float Ax[] = {5,5.8,0.7,-0.4,-5.2,-4.3};
float Ay[] = {3.6,2.7,-5.6,-5.8,3.4,3.9};
float Az[] = {0,0,0,0,0,0};
//Matriz con los puntos de referencia de la base fija
float Bx[] = {1.2,8.8,7.3,-7.3,-8.8,-1.2};
float By[] = {9.3,-4.6,-6.5,-6.5,-4.6,9.3};
float Bz[] = {0,0,0,0,0,0};
float R[] = {pose[3],pose[4],pose[5]}; //Vector que contiene los
ángulos de  orientación (p,y,r) que se desean alcanzar
float D[] = {pose[0],pose[1],pose[2]}; //Vector que contiene las
coordenadas  (x,y,z) que se desean alcanzar
//Se pasan los ángulos expresados en grados a radianes
float Rrad1 = (R[0]*PI)/180;
float Rrad2 = (R[1]*PI)/180;
float Rrad3 = (R[2]*PI)/180;


//Se asignan variables con las funciones trigonométricas
necesarias para  realizar la matriz de transformación
float cphi = cos(Rrad1);
float ctita = cos(Rrad2);
float cpsi = cos(Rrad3);
float sphi = sin(Rrad1);
float stita = sin(Rrad2);
float spsi = sin(Rrad3);
float sphi_stita = sphi*stita;
float cphi_stita = cphi*stita;
//Matriz de transformación
float RRf1[] = {ctita*cpsi+sphi_stita*spsi, ctita*spsi+sphi_stita*cpsi,
cphi_stita};
float RRf2[] = {cphi*spsi,cphi*cpsi,-sphi};
float RRf3[] = {-stita*cpsi+sphi*ctita*spsi,stita*spsi+sphi*ctita*cpsi,
cphi*ctita};
//Multiplicación de la matriz de transformación por la matriz A y
el resultado  se almacena en una nueva matriz denominada C
float C11 = RRf1[0]*Ax[0]+RRf1[1]*Ay[0]+RRf1[2]*Az[0];
float C12 = RRf1[0]*Ax[1]+RRf1[1]*Ay[1]+RRf1[2]*Az[1];
float C13 = RRf1[0]*Ax[2]+RRf1[1]*Ay[2]+RRf1[2]*Az[2];
float C14 = RRf1[0]*Ax[3]+RRf1[1]*Ay[3]+RRf1[2]*Az[3];
float C15 = RRf1[0]*Ax[4]+RRf1[1]*Ay[4]+RRf1[2]*Az[4];
float C16 = RRf1[0]*Ax[5]+RRf1[1]*Ay[5]+RRf1[2]*Az[5];
float Cx[] = {C11,C12,C13,C14,C15,C16};
```

```
float C21 = RRf2[0]*Ax[0]+RRf2[1]*Ay[0]+RRf2[2]*Az[0];
float C22 = RRf2[0]*Ax[1]+RRf2[1]*Ay[1]+RRf2[2]*Az[1];
float C23 = RRf2[0]*Ax[2]+RRf2[1]*Ay[2]+RRf2[2]*Az[2];
float C24 = RRf2[0]*Ax[3]+RRf2[1]*Ay[3]+RRf2[2]*Az[3];
float C25 = RRf2[0]*Ax[4]+RRf2[1]*Ay[4]+RRf2[2]*Az[4];
float C26 = RRf2[0]*Ax[5]+RRf2[1]*Ay[5]+RRf2[2]*Az[5];
float Cy[] = {C21,C22,C23,C24,C25,C26};

float C31 = RRf3[0]*Ax[0]+RRf3[1]*Ay[0]+RRf3[2]*Az[0];
float C32 = RRf3[0]*Ax[1]+RRf3[1]*Ay[1]+RRf3[2]*Az[1];
float C33 = RRf3[0]*Ax[2]+RRf3[1]*Ay[2]+RRf3[2]*Az[2];
float C34 = RRf3[0]*Ax[3]+RRf3[1]*Ay[3]+RRf3[2]*Az[3];
float C35 = RRf3[0]*Ax[4]+RRf3[1]*Ay[4]+RRf3[2]*Az[4];
float C36 = RRf3[0]*Ax[5]+RRf3[1]*Ay[5]+RRf3[2]*Az[5];
float Cz[] = {C31,C32,C33,C34,C35,C36};
//Matriz D de traslación
float Dx[] = {pose[0],pose[0],pose[0],pose[0],pose[0],pose[0]};
float Dy[] = {pose[1],pose[1],pose[1],pose[1],pose[1],pose[1]};
float Dz[] = {pose[2],pose[2],pose[2],pose[2],pose[2],pose[2]};

//Se calcula la distancia euclídea entre la matriz obtenida y la
matriz B
float Fx[] ={pow(Cx[0]+Dx[0]-Bx[0],2),pow(Cx[1]+Dx[1]-Bx[1],2),
pow(Cx[2]+Dx[2]-Bx[2],2),pow(Cx[3]+Dx[3]-Bx[3],2),pow(Cx[4]+Dx[4]-
Bx[4],2),pow(Cx[5]+Dx[5]-Bx[5],2)};
float Fy[] ={pow(Cy[0]+Dy[0]-By[0],2),pow(Cy[1]+Dy[1]-
By[1],2),pow(Cy[2]+Dy[2]-By[2],2),pow(Cy[3]+Dy[3]-By[3],2),
pow(Cy[4]+Dy[4]-By[4],2),pow(Cy[5]+Dy[5]-By[5],2)};
float Fz[] ={pow(Cz[0]+Dz[0]-Bz[0],2),pow(Cz[1]+Dz[1]-
Bz[1],2),pow(Cz[2]+Dz[2]-Bz[2],2),pow(Cz[3]+Dz[3]-Bz[3],2),
pow(Cz[4]+Dz[4]-Bz[4],2),pow(Cz[5]+Dz[5]-Bz[5],2)};
float summat[] = {Fx[0]+Fy[0]+Fz[0],Fx[1]+Fy[1]+Fz[1],
Fx[2]+Fy[2]+Fz[2],Fx[3]+Fy[3]+Fz[3],Fx[4]+Fy[4]+Fz[4],Fx[5]+
Fy[5]+Fz[5]};
//Se guardan en el vector L las longitudes calculadas de cada motor
float L[] = {sqrt(summat[0]),sqrt(summat[1]),sqrt(summat[2]),
sqrt(summat[3]),sqrt(summat[4]), sqrt(summat[5])};

for (int k=0; k<6; k++){
      consigna1[k] = L[k]-17.5; //Resta entre las longitudes
calculadas y las  longitudes físicas de los motores (17.5cm)
      tiemposMotores[k] = (consigna1[k]/3.2)*1.7; //Cálculo de los
tiempos del  recorrido de cada motor (empleando velocidad máxima =
3.2cm/s)
      calcconsigna[k] = 1023*consigna1[k]/5.0; //Se escala la
consigna obtenida  entre 0 y 1023 que es el rango en el que trabaja
```

```
Arduino
}

//Obtencion del tiempo del motor con mayor recorrido
float maximo = 0.0; //Variable qe almacenará el mayor tiempo de
los 6 motores
for (int u=0; u<6; u++) { //Bucle que recorre el vector de tiempos
      if (maximo < tiemposMotores[u]) { //compara cada elemento
con el valor de  la variable maximo
      maximo = tiemposMotores[u];
      }
}
tiempo = maximo; //Se guarda el tiempo máximo

float temp = 0; //Variable que contemplará si existe alguna
consigna no válida
for (int w=0; w<6; w++){
    if (calcconsigna[w]<0 || calcconsigna[w]>1023){ //Se comprueba
que la  consigna no supere los límites
      temp ++; //Si se cumple la condición anterior se registra
incrementando el  número de la variable temp
      Serial.println("Posición no válida");
      }
}

int temp0 = 0;

for(int e=0; e<6; e++){
    temp0 = analogRead(e+1); //Se lee las posiciones obtenidas de
los  potenciómetros de los motores
posicioninicial[e] = temp0*5.0/1023.0; //Se guarda la posición
inicial de cada motor y se escalan entre 0 y 5 para convertirlo a
cm reales
}

return(temp); //la función devuelve el valor temp

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,//

void control() {

int temp2;

for(int i=0; i<6; i++){ //Se emplea un bucle de 6 iteraciones para
evitar  repetir el código para cada motor
```

```
temp2 = analogRead(i+1); //Se lee el valor de la entrada analógica que nos proporcionará la posición del motor en un rango entre 0 y 1023
posicion[i] = temp2*5.0/1023.0; //Se escala el valor obtenido para trabajar en un rango entre 0 y 5 voltios

//----------------------// //---------------------------//

//Lazo de control para POSICIÓN (controlador PID)
error0pos[i] = consignapos[i] - posicion[i]; //Cálculo del error entre la consigna deseada y la posición real que posee el actuador
Ppos[i] = kppos*error0pos[i]; //Parte proporcional del controlador PID de posición

if (fabs(salidaMotorPosiciones[i])<255){ //Si el comando enviado a los motores es menor que 255 se aplica la parte derivativa e integral
Dpos[i] = qdpos*(error0pos[i]-error1pos[i]); //Parte derivativa del controlador PID de posición
Ipos[i] = Ipos[i]+qipos*error0pos[i]; //Parte integral del controlador PID de posición
if (Ipos[i] >= 25){ //Si la parte integral supera el valor de 25, esta se establece en dicho valor
Ipos[i] = 25;}
else if (Ipos[i] <= -25){ //Si la parte integral posee un valor inferior a -25, esta se establece en dicho valor
Ipos[i] = -25;}
}

salidaMotorPosiciones[i] = Ppos[i]+Ipos[i]-Dpos[i]; //El comando de PWM enviado a los motores será una combinación de las partes anterior

error1pos[i] = error0pos[i]; //Se almacena el valor del error calculado en la variable error1pos

//----------------------// //---------------------------//

//Lazo de control para VELOCIDAD(controlador PID)
tiempoLectActual = millis(); //Obtención del tiempo actual de ejecución del programa

if (inicioInterrup == 0){ //En la primera interrupción que se realice la velocidad de la plataforma valdrá 0 porque no ha comenzado a moverse aún
velocidad[i] = 0;} else{
```

```
//La velocidad se calcula como el espacio recorrido por el
actuador entre el tiempo de empleado en recorrerlo
velocidad[i] = (((fabs(posicion[i]-posicioninicial[i]))
*10)/(tiempoLectActual));} //Se obtiene la velocidad en mm/ms
velocidad[i] = velocidad[i] * 100; //La velocidad obtenida
anteriormente se pasa a cm/s

error0vel[i] = consignavel[i] - velocidad[i]; //Cálculo del error
entre la consigna deseada y la velocidad actual del motor
Pvel[i] = kpvel*error0vel[i]; //Parte proporcional del
controlador PID de velocidad

if (fabs(salidaMotorVelocidades[i])<255){
Dvel[i] = qdvel*(error0vel[i]-error1vel[i]); //Parte derivativa
del controlador PID de velocidad
Ivel[i] = Ivel[i]+qivel*error0vel[i]; //Parte integral del
controlador PID de velocidad
if (Ivel[i] >= 25){ //Si la parte integral supera el valor de 25,
esta se establece en dicho valor
Ivel[i] = 15;}
else if (Ivel[i] <= -25){ //Si la parte integral posee un valor
inferior a -25, esta se establece en dicho valor
Ivel[i] = -15;}
}

salidaMotorVelocidades[i] = Pvel[i]+Ivel[i]-Dvel[i]; //El comando
de PWM enviado a los motores será una combinación de las partes
anterior

error1vel[i] = error0vel[i]; //Se almacena el valor del error
calculado en la variable error1vel

//-----------------------// //---------------------------//

//Se comparan las salidas de los dos controladores y se escoge la
menor
if (fabs(salidaMotorPosiciones[i]) < salidaMotorVelocidades[i]){
//Si la salida del controlador de posición es menor que la de
velocidad

if (error0pos[i] > -0.15 && error0pos[i] < 0.15){ //Si el error se
encuentra dentro de un umbral cercano a la posición deseada no se
envía señal
analogWrite (enable[i], 0);

else{
```

```
  if (salidaMotorPosiciones[i] > 0.0){ //Se establece el límite
superior del comando de PWM a 255 (valor máximo admisible de PWM,
ya que su rango es de 0 a 255)
  if (salidaMotorPosiciones[i] >= 255.0){salidaMotorPosiciones[i]
= 255.0;
}
  digitalWrite (entrada2[i], LOW); //Se establecen las entradas
del puente en H para que el motor se extienda
  digitalWrite (entrada1[i], HIGH);

  analogWrite (enable[i], round(fabs(salidaMotorPosiciones[i])));
//Se envía la salida calculada en valor absoluto a los motores
}
else {
  if (salidaMotorPosiciones[i] <= -255.0){ //Se establece el
límite inferior del comando de PWM a -255
  salidaMotorPosiciones[i] = -255.0;
}
 digitalWrite (entrada2[i], HIGH); //Se establecen las entradas
del puente en H para que el motor se contraiga
  digitalWrite (entrada1[i], LOW);
  analogWrite (enable[i], round(fabs(salidaMotorPosiciones[i])));
//Se envía la salida calculada en valor absoluto a los motores
      }
    }
}

if (fabs(salidaMotorPosiciones[i]) > salidaMotorVelocidades[i]){
//Si la salida del controlador de posición es menor que la de
velocidad

if (error0pos[i] > -0.15 && error0pos[i] < 0.15){ //Si el error se
encuentra dentro de un umbral cercano a la posición deseada no se
envía señal
analogWrite (enable[i], 0);}
else{
   if (salidaMotorVelocidades[i] > 0.0){ //Se establece el límite
superior del comando de PWM a 255
      if (salidaMotorVelocidades[i] >= 255.0){
salidaMotorVelocidades[i] = 255.0;}
if (error0pos[i] > 0){ //Si el error de posición es mayor que 0 se
establecen las entradas del puente en H para que el motor se
extienda
digitalWrite (entrada2[i], LOW);
digitalWrite (entrada1[i], HIGH);}
```

```
if (error0pos[i] < 0){ //Si el error de posición es menor que 0 se
establecen las entradas del puente en H para que el motor se
contraiga
  digitalWrite (entrada2[i], HIGH);
   digitalWrite (entrada1[i], LOW);}
   analogWrite (enable[i],round(fabs(salidaMotorVelocidades[i])));
//Se  envía la salida calculada en valor absoluto a los motores
      }
     }
  }
}
inicioInterrup = inicioInterrup + 1; //Se aumenta en 1 el valor de
la  variable para tener constancia que ya se ha realizado la
primera interrupción
}

//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,//

void loop(){
if (inicio == 0){ //El cálculo de las consignas se ejecuta sólo
una vez  para aligerar el procesado
  int temp = longitudes();
  if (temp==0){ //Se comprueba que la consigna no supera los
límites  establecidos con la variable temp
//Se envían las consignas calculadas al controlador de posición
   for (int i=0; i<6; i++){
        consigna[i] = calcconsigna[i]*5.0/1023;
      }
    }
  int temp3 = 0;
  float consigna2[6] = {0,0,0,0,0,0};

  for (int g=0; g<6; g++){
    consigna2[g] = consigna1[g]/tiempo; //La consigna de
velocidad será el recorrido que tiene que hacer cada motor entre
el tiempo empleado en recorrerlo
    if (consigna2[g]<0 || consigna2[g]>3.3){ //Si la consigna no
se cuentra entre 0 y 3.3cm/s (velocidad máxima de los motores) se
aumenta temp3
     temp3 ++;
     }
   }
  if (temp3==0){ //Se comprueba que la consigna no supera los
límites  establecidos con la variable temp3
   for (int t=0; t<6; t++){
     consignavel[t] = consigna2[t]; //Se envían las consignas
calculadas  al controlador de velocidad
```

```
    }
  }
  inicio = inicio + 1; //Se aumenta en 1 la variable para reflejar
que ya  se ha efectuado el cálculo de las consignas
   }
 }
```