ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA CIVIL E INDUSTRIAL

Trabajo Fin de Grado

SISTEMA DE POSICIONAMIENTO XYZ

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Autor: Kevin Mínguez Olivero

Tutor: Jonay Tomás Toledo Carrillo

Codirector: Antonio Luis Morell González

Septiembre, 2014

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

1. Introducción

1.1 Objetivo del Trabajo de Fin de Grado.

1.1.1 Resumen.

El objetivo del trabajo fin de grado que se presenta es el de crear un sistema robótico de posicionamiento espacial, capaz de colocarse en cualquier punto XYZ y con una inclinación prestablecida dentro de su espacio de trabajo.

En el desarrollo del proyecto se ha empleado la plataforma de hardware libre Arduino, específicamente la placa Arduino Mega 2560 y su entorno de desarrollo, para implementar el control del sistema, mediante controlador PID.

Por otra parte se ha utilizado el software matemático MATLAB para realizar los cálculos necesarios que generarán las consignas de los actuadores del robot, a través del estudio de su cinemática inversa.

En el proyecto se incluye, por tanto, 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.

1.1.2 Abstract.

The objective of the final project presented is to create a robotic spatial positioning system, can be placed at any point XYZ and a specific inclination within your workspace.

In the project we have used the Arduino open hardware platform, specifically Arduino Mega 2560 and its development environment, to implement the control system using PID controller.

Furthermore we have used the mathematical software MATLAB to perform the calculations necessary to generate the setpoints for the robot’s actuators, through the study of its inverse kinematics.

The project therefore includes, robot programming; design, manufacture and installation of the additional electronics; as well as the design, creation and assembly of the robotic system.

1.2 Sistemas de posicionamiento XYZ. Plataforma Stewart-Gough.

Un sistema de posicionamiento XYZ es un sistema mecánico capaz de ubicarse en cualquier punto del espacio dentro de su rango de trabajo. En el presente proyecto se ha desarrollado uno de estos tipos de sistemas de posicionamiento espacial denominado plataforma de Stewart-Gough.

Una plataforma Stewart-Gough es un tipo de robot paralelo de 6 grados de libertad que cuenta con 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. Esto se consigue calculando cuánto debe retraerse o expandirse cada actuador mediante el estudio de la cinemática inversa de la plataforma, de la que obtenemos cada una de las consignas de los actuadores a partir de una traslación y orientación prefijada para la base móvil. Este tipo de robots permiten el movimiento longitudinal, rotacional así como movimientos combinados aportando al efector final una gran variedad de movimientos y ángulos.

Este tipo de sistemas se han utilizado en diversas aplicaciones como:

- Simuladores de conducción y de vuelo. Esta es la aplicación más extendida para este tipo de tecnología. En este caso se busca simular los movimientos que se generarían en estos vehículos a la hora de actuar sobre los mandos de los mismos o cuando interactúan con el ambiente (al aterrizar, simulando turbulencias o baches…). Este tipo de sistemas se centran en generar en los usuarios las sensaciones físicas de aceleración y orientación que experimentarían los pilotos al vivir estas maniobras, teniendo en cuenta los estímulos necesarios a recrear.
- Robots manipuladores. Por otra parte, también se ha usado este tipo de robots como actuadores para robots más sofisticados.
- Sistema de acoplamiento de bajo impacto para vehículos espaciales (LIDS). Esta es una aplicación particular desarrollada y empleada por la NASA para facilitar el acoplamiento entre vehículos espaciales, como por ejemplo entre transbordadores y estaciones espaciales.

Aparte de las aplicaciones generales en las que se ha empleado este tipo de robot, mencionadas anteriormente, es interesante remarcar que en nuestro caso, la plataforma fabricada en este proyecto será empleada para investigación y pruebas por parte del departamento de Ingeniería Informática de la facultad.

2. Diseño y fabricación de la plataforma Stewart

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.

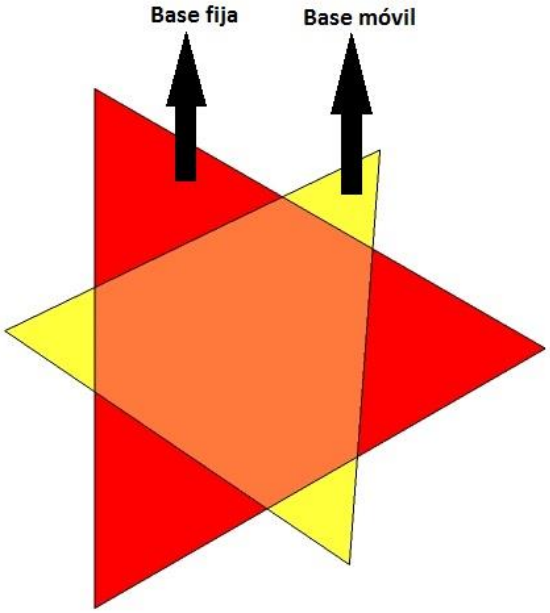


Figura 1. Forma y configuración de bases escogida

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:

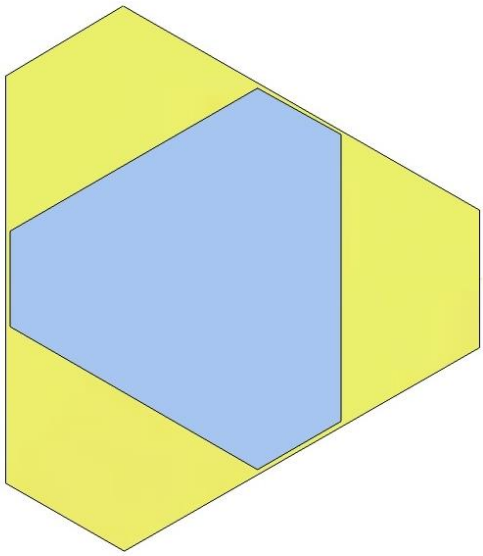


Figura 2. Forma final empleada en las bases

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:

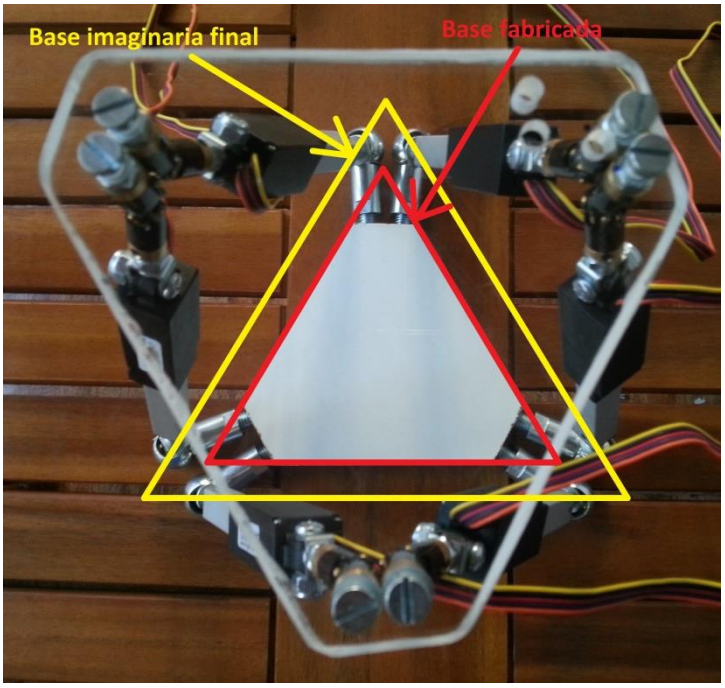


Figura 3. Base inferior imaginaria

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:

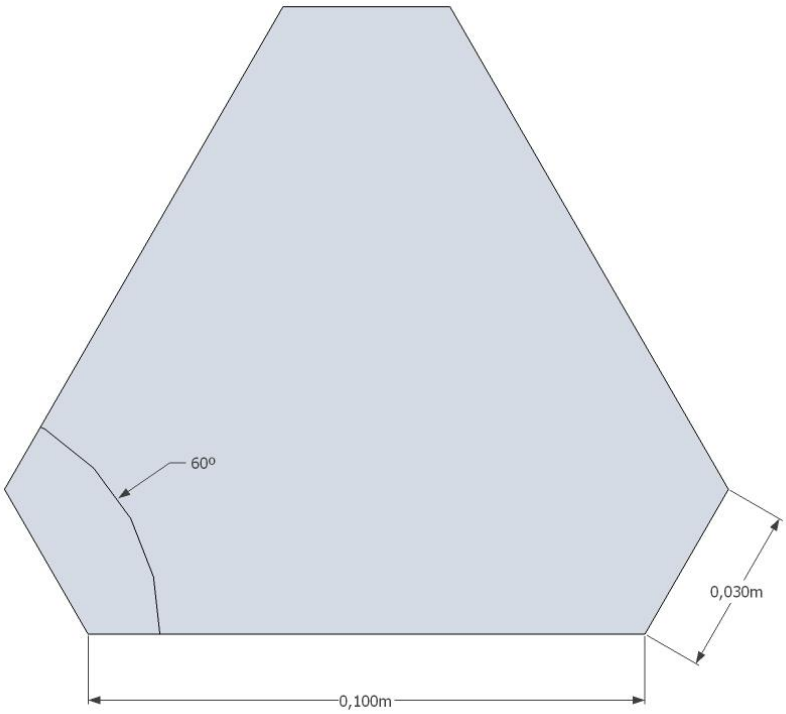


Figura 4. Dimensiones usadas para las bases

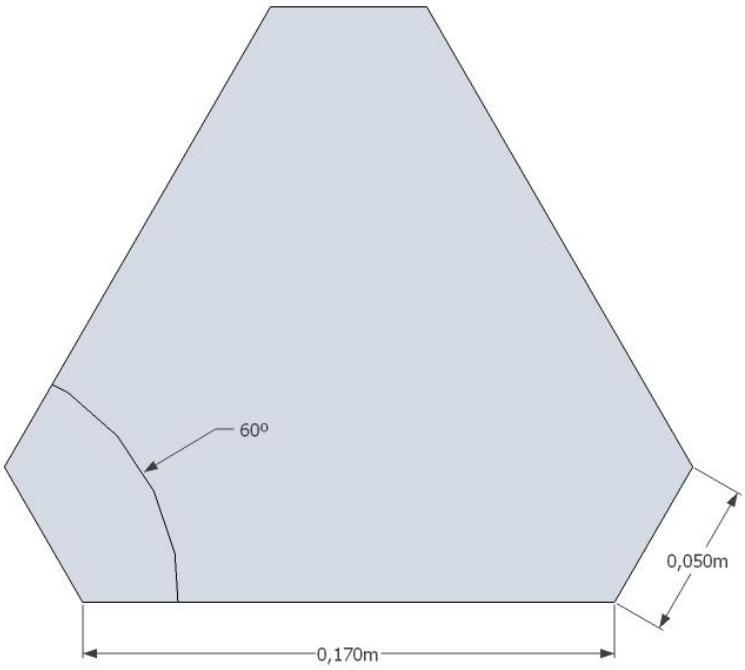


Figura 5. Dimensiones de la base inferior imaginaria

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 6. Motor lineal Firgelli, serie L16

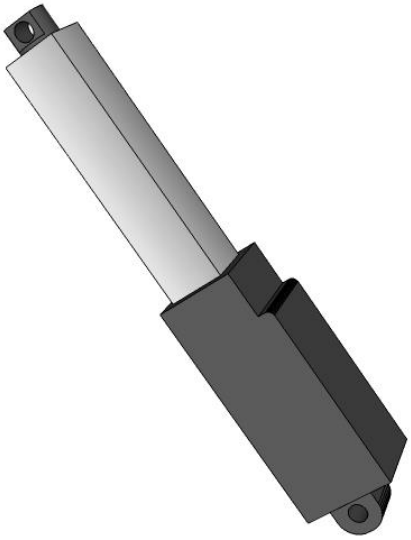


Figura 7. Diseño 3D (SketchUp) de motor empleado

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:

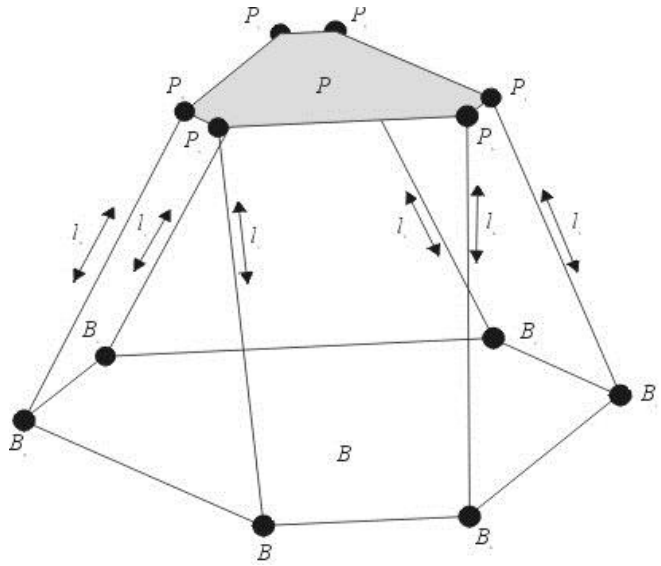


Figura 8. Ubicación de actuadores y articulaciones

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 9. 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 10. 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.

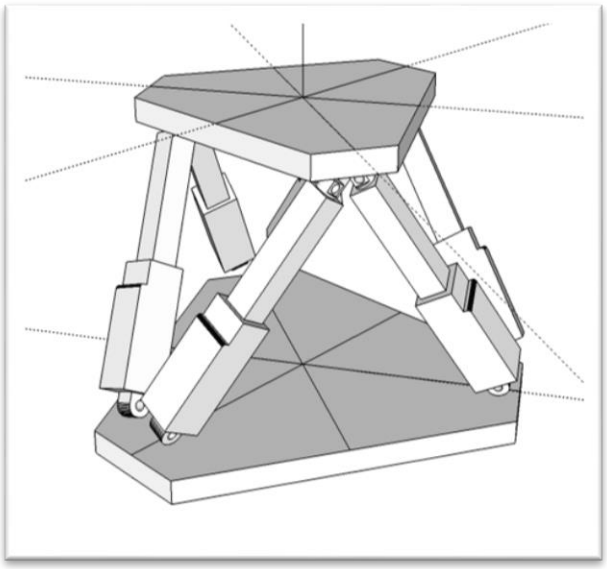


Figura 11. Diseño 3D básico, plataforma Stewart

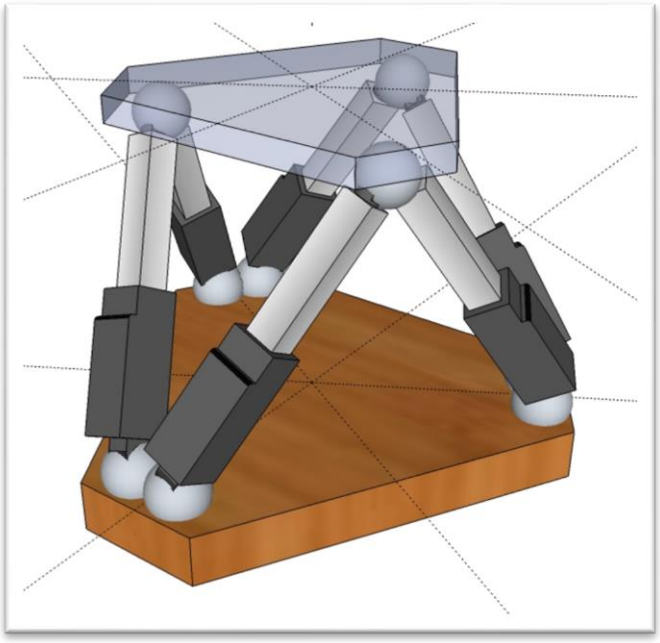


Figura 12. Diseño 3D final simplificado, plataforma Stewart

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:

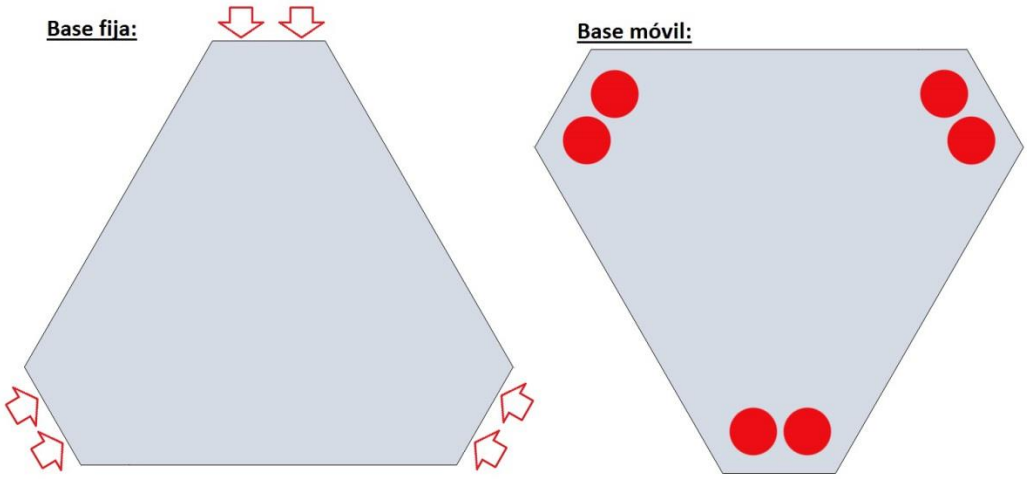


Figura 13. Ubicación de las articulaciones en las bases

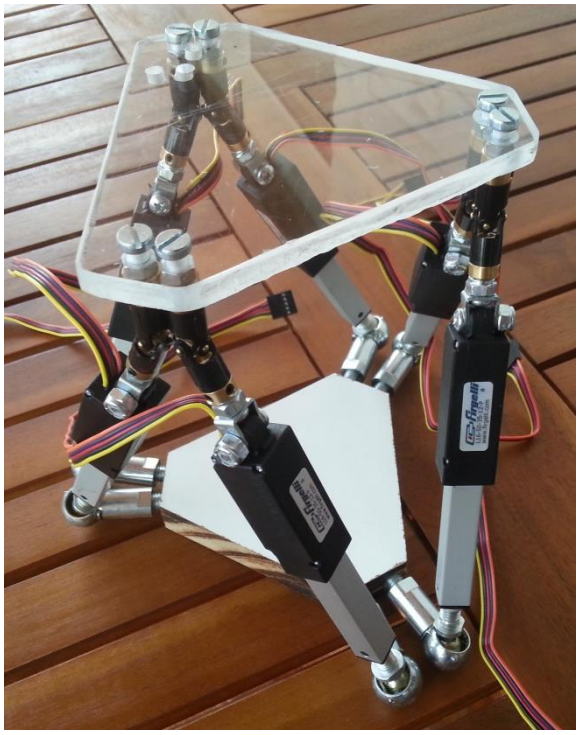


Figura 14. Plataforma robótica Stewart final

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

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.

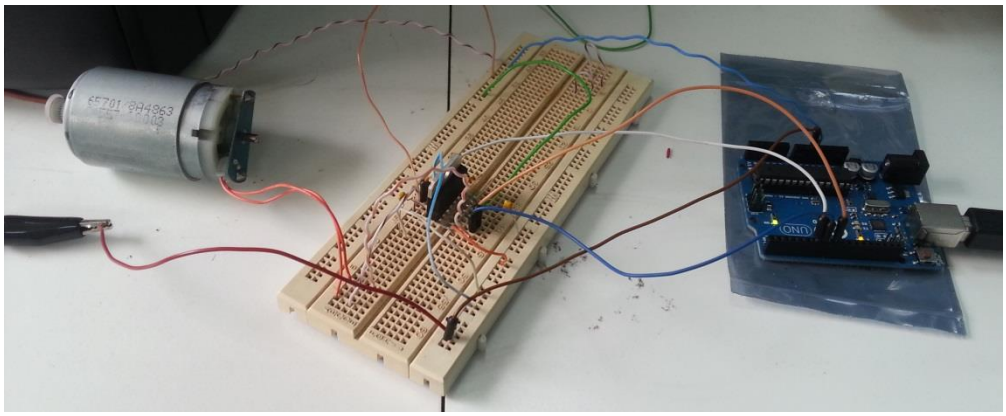


Figura 15. Primeras pruebas previas al diseño de la placa PCB

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.

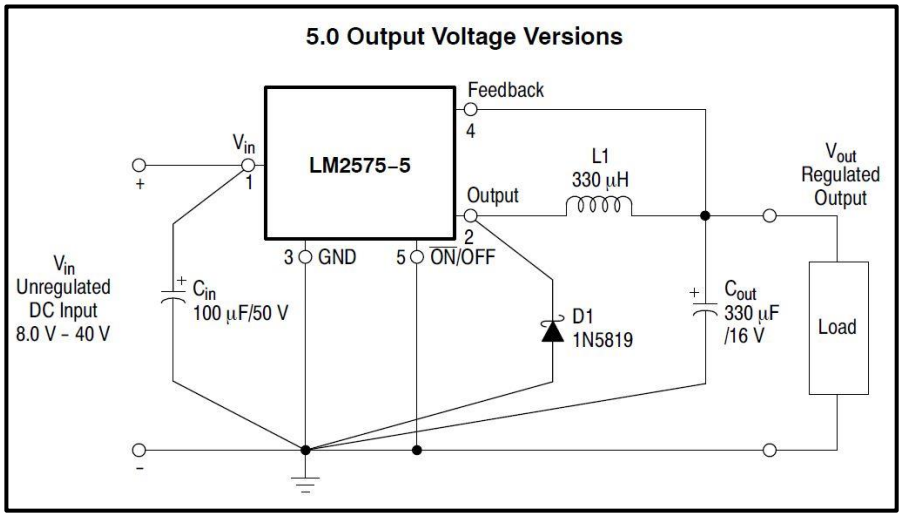


Figura 16. Configuración LM2575 para voltaje de salida de 5V

- 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.*

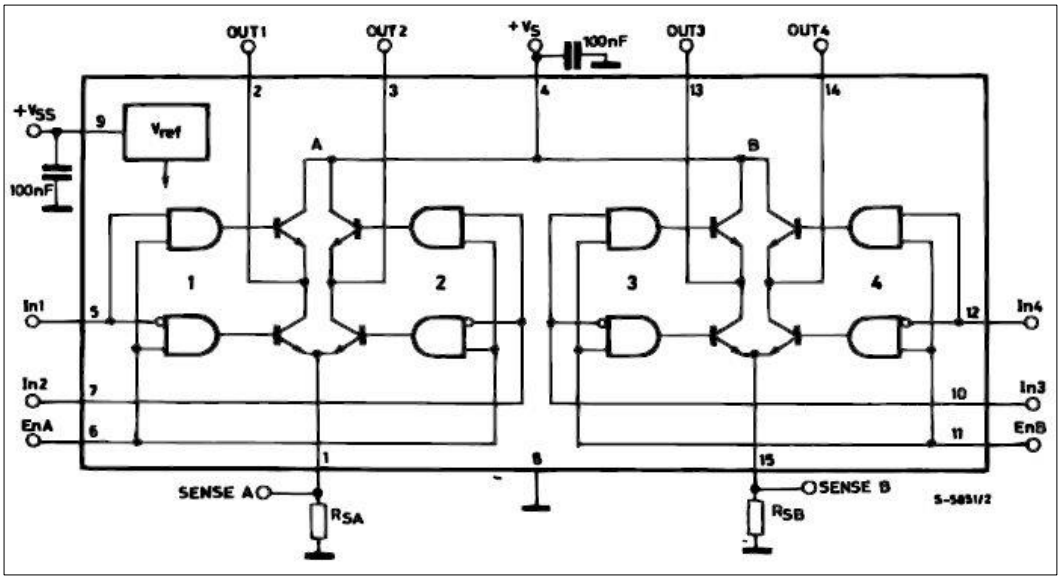


Figura 17. Composición de un L298 (2 puentes en H)

- 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:

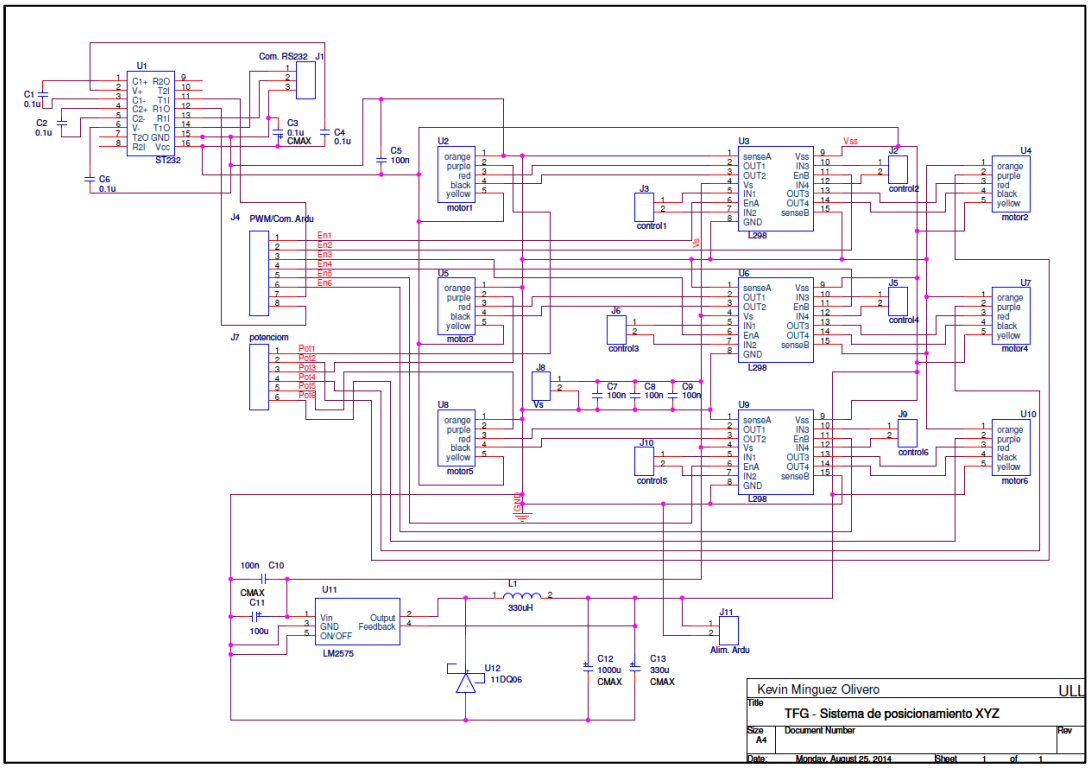


Figura 18. Diseño esquemático de la placa PCB

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 (mm)	Diámetro taladros (mm)
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 19. 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:

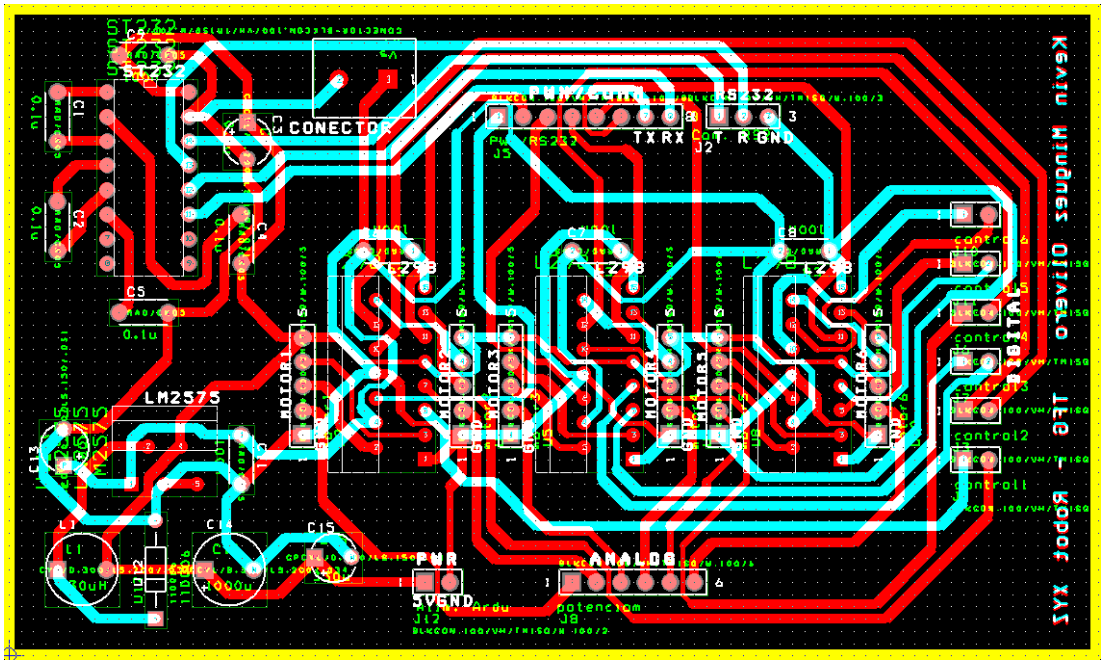


Figura 20. Diseño físico de la placa PCB

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.

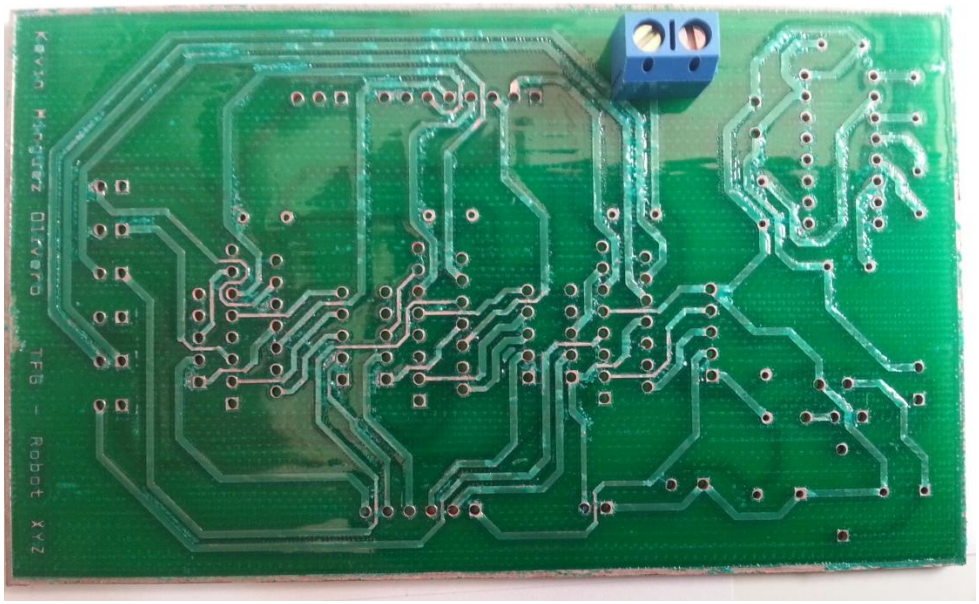


Figura 21. Placa PCB sin componentes (capa BOT)

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

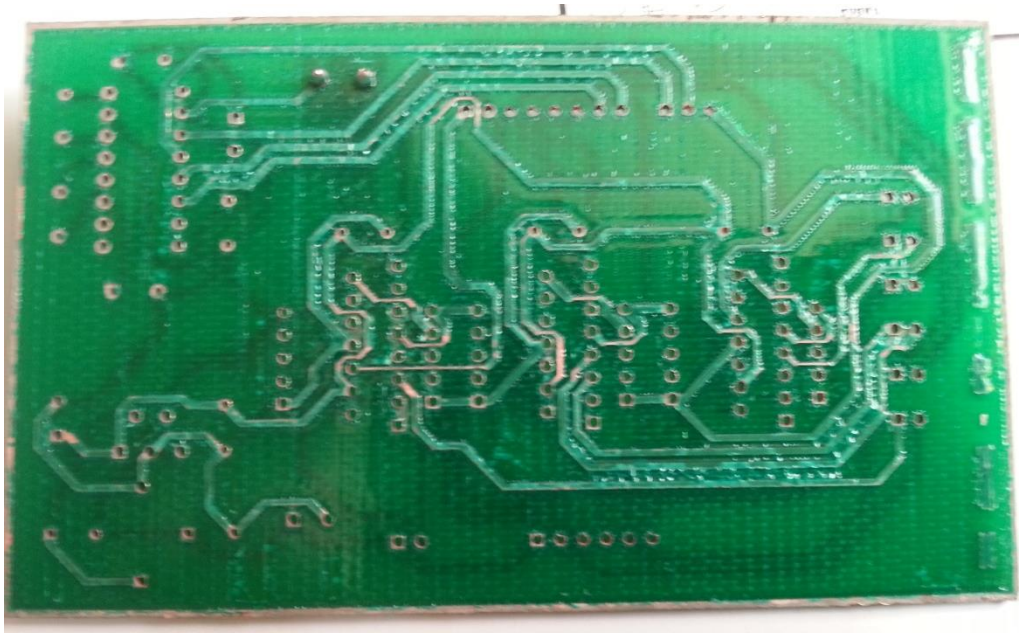


Figura 22. Placa PCB sin componentes (capa TOP)

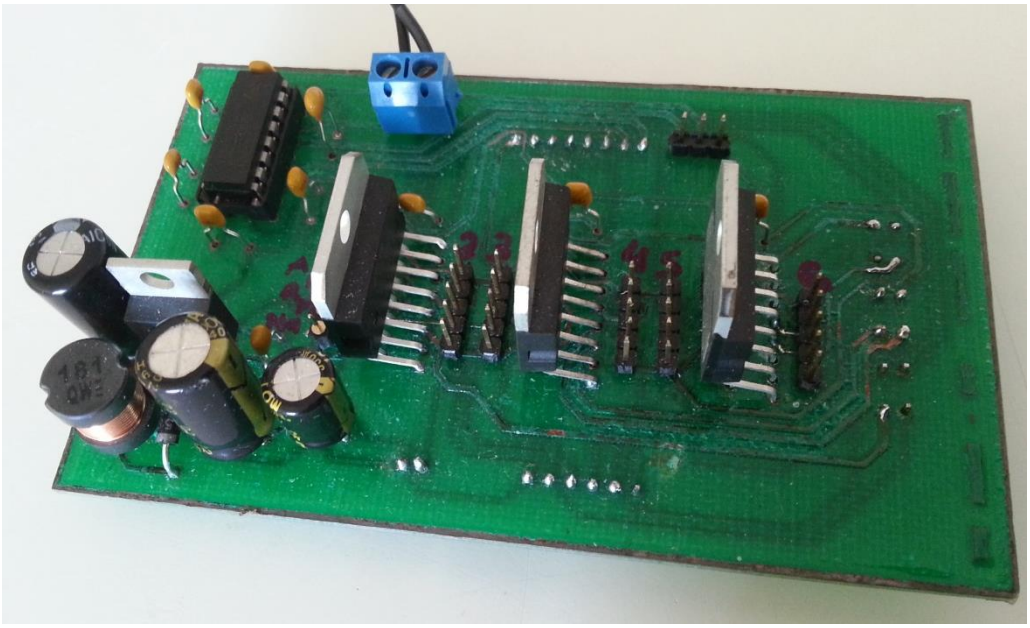


Figura 23. Placa PCB final (capa TOP)

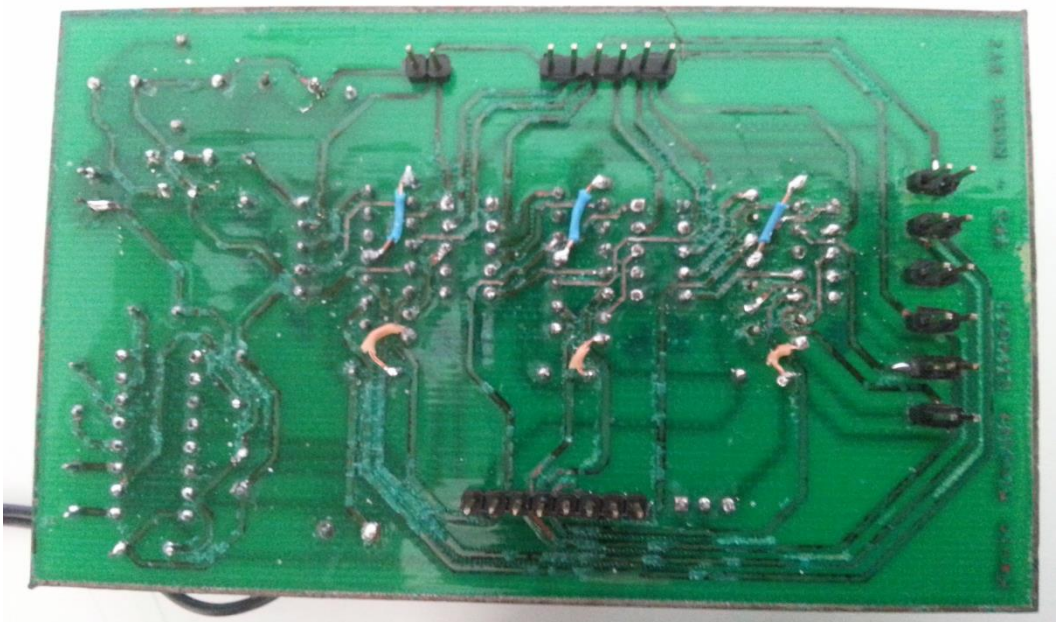


Figura 24. Placa PCB final (capa BOT)

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.

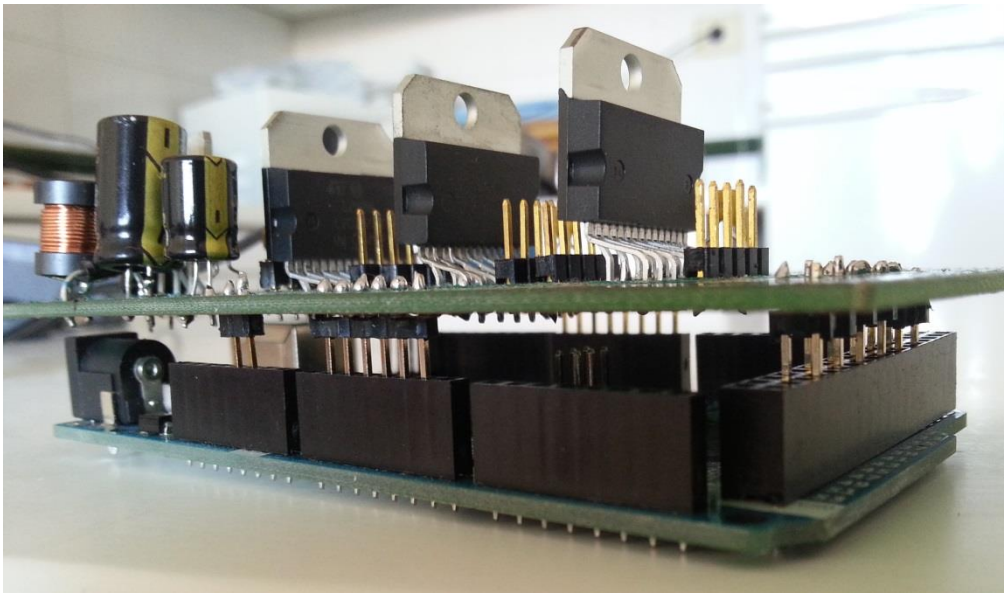


Figura 25. Comprobación de la correcta colocación de los pines de conexión con Arduino

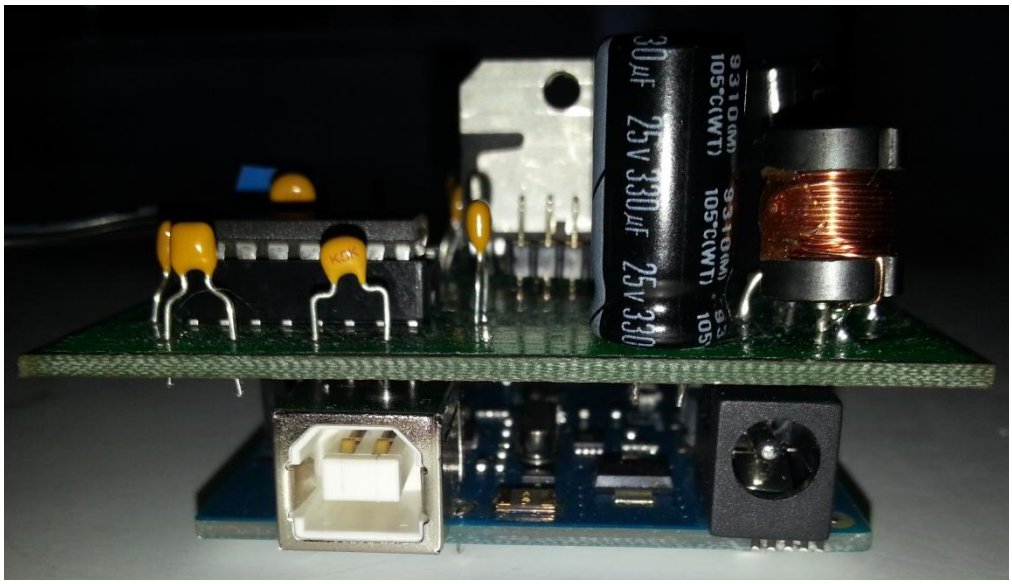


Figura 26. Sistema ensamblado de placa PCB y Arduino

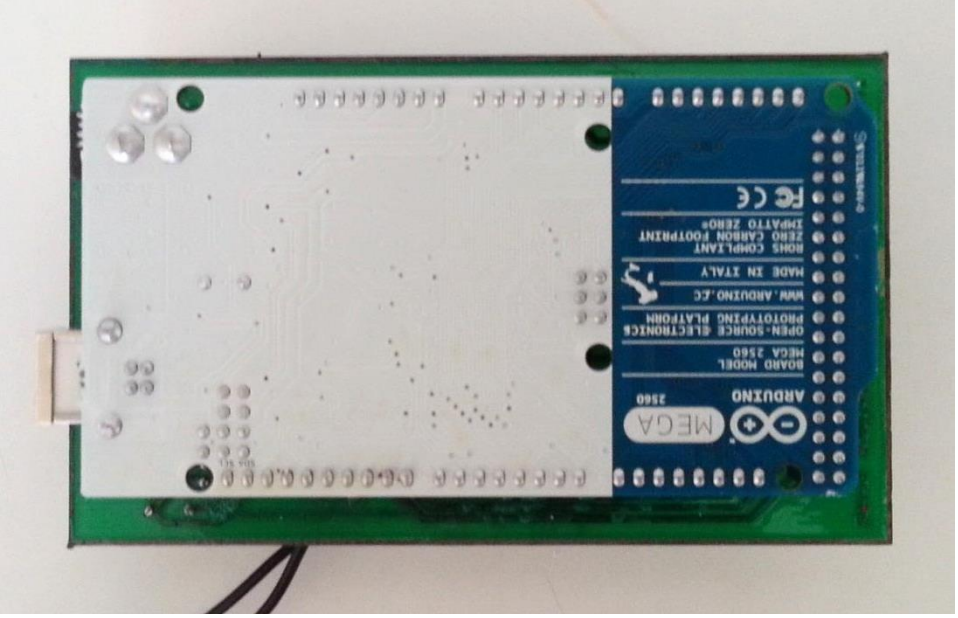


Figura 27. Comparación de tamaño entre placa PCB y Arduino

4. Programación de la plataforma robótica

La programación es uno de los aspectos más importantes a la hora de desarrollar cualquier sistema robótico o automático en general. En la programación debe incluirse, no solo el código necesario para conseguir la funcionalidad buscada en el robot, sino también, mecanismos de protección para el propio robot o para sus usuarios, así como métodos que eviten posibles errores de funcionamiento por un mal uso del mismo, entre otros aspectos. Para conseguir esto, el programador debe tomar en consideración muchas situaciones que pueden surgir durante el movimiento del dispositivo o la interactuación del mismo con los usuarios. Entre las posibles situaciones que se deben considerar pueden encontrarse:

- limitar el rango de actuación de los actuadores teniendo en cuenta sus posibilidades mecánicas y eléctricas.
- evitar contradicciones o incongruencias en el código con el fin de prevenir actuaciones indeseables
- minimizar los movimientos inútiles o poco eficientes. Se debe tratar de conseguir un movimiento suave y eficiente.
- reducir al máximo los posibles picos de consumo eléctrico, resultantes, por ejemplo, de movimientos demasiado bruscos o bloqueo de los actuadores, siempre y cuando esto no influya en el objetivo del sistema

- simplificar la interfaz de usuario –si la hubiese- para evitar confusiones a la hora de asignar órdenes al robot

Estas son algunas de las consideraciones a tener en cuenta a la hora de desarrollar un software eficiente y funcional para cualquier sistema automático.

En el presente proyecto, la programación llevada a cabo la podríamos dividir en dos partes: la parte consistente en el posicionamiento y control de los actuadores, llevada a cabo por la placa Arduino; y la parte en la que se calculan las consignas de cada actuador mediante la resolución de la cinemática inversa del sistema, llevada a cabo por el software matemático MATLAB.

4.1 Programación para Arduino. Control y posicionamiento de los actuadores.

Arduino es una plataforma de computación física de código abierto basado en una placa electrónica simple, y un entorno de desarrollo con el que escribir software para la placa. Arduino se puede utilizar para desarrollar una gran variedad de aplicaciones, tomando las entradas de una variedad de interruptores o sensores, y pudiendo controlar una variedad de salidas físicas como luces o motores. Los proyectos de Arduino pueden ser autónomos, pero también pueden comunicarse con software externo o incluso con otros dispositivos. El microcontrolador de las placas Arduino es programado usando el lenguaje de programación de Arduino (basado en Wiring) usando su entorno de desarrollo (basado en Processing).

El objetivo buscado mediante la programación de la placa Arduino fue el de realizar el control de los motores, es decir que cada uno se dirigiera de la forma más rápida y precisa posible a la posición que le indique su consigna en cada momento. El cálculo de la consigna no ha sido implementado en la Arduino debido a que la velocidad de su procesador no sería suficiente para realizar dicho cálculo y el control de los motores simultáneamente de manera óptima. Por esto se decidió implementar este cálculo mediante un programa en MATLAB que le enviaría directamente las consignas a la Arduino, para poder dedicar a esta última únicamente a controlar más rápida y eficientemente los actuadores. Sobre el cálculo de las consignas se hablará en el apartado siguiente.

Resumiendo, la función de la Arduino Mega consistiría en leer las consignas suministradas por MATLAB mediante puerto serial y enviar cada motor a estas posiciones controlando que se haga de manera rápida y precisa. El mecanismo de control empleado para conseguir esto se trata de un controlador PID digital.

4.1.1 Nociones sobre control con controlador PID y su inclusión en el código.

Un controlador PID (Proporcional Integral y Derivativo) basa su actuación en proporcionar señales a los actuadores basadas en el estudio del error entre el valor actual de la variable a controlar y su valor deseado o consigna. Al hacer uso del error, podemos ver que consiste en un mecanismo de control realimentado o de lazo cerrado. Un controlador PID se basa en tres acciones realizadas sobre el error de manera conjunta. Estas acciones son:

- 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 sobreocscilaciones. 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 tanto la acción de control puede ser modificada según los valores asignados a las constantes de sus aportaciones y sigue la forma siguiente:

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

Esta fórmula se correspondería con la de un controlador PID continuo pero debemos tener en cuenta que mediante Arduino solo sería posible obtener el error cada cierto tiempo y no de forma continua. Arduino lee los valores de posición de los motores utilizando el potenciómetro integrado en los mismos y el conversor analógico digital. A partir del voltaje se calcula la posición en la que se encuentra

posicionado el motor y, teniendo la consigna, se calcula el error. Esta lectura se realiza al inicio del ciclo del programa y no se efectúa de nuevo hasta el comienzo de un nuevo ciclo. Esto provoca que no podamos obtener información de los actuadores durante tiempo que tarda en ejecutarse el programa completo. Debido a esto, para esta aplicación se ha tenido que emplear un controlador PID discreto o digital. Este tipo de PID es similar al continuo pero con algunas variaciones en los cálculos de las distintas aportaciones. Para un PID discreto las diferentes acciones resultarían:

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

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

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

Como podemos ver, en estas fórmulas intervienen los valores del error en instantes de tiempo sucesivos (y con un periodo idéntico), pero no de manera continua, lo que nos permite implementar este tipo de controlador en nuestro código.

Las constantes proporcional, integral y derivativa fueron definidas como variables globales al inicio de nuestro código pero para escoger unos buenos valores hubo que hacer una sintonización del PID mediante ensayo y error. Esto fue realizado 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.

A pesar de poder incluir este tipo de controlador, la placa Arduino no tiene un tiempo fijo para ejecutar completamente el programa. Esto provoca que la información aportada por los motores y, por ende, el cálculo de los errores, no tendrían una sucesión periódica. Estos valores no periódicos provocarían que la actuación del controlador fuera inestable y por tanto, no óptima, si no se definía una forma de paliar esto. Debido a esto, a la hora de definir el controlador PID digital en nuestro código fue necesario implementar un método que forzara al procesador a realizar la medida de la posición de los motores así como el control de los mismos de manera periódica (con periodo predefinido). Esto tuvo que realizarse haciendo uso de interrupciones.

Para configurar dichas interrupciones fue necesario incluir en el programa una librería llamada ‘TimerOne’ que permite iniciar un contador al inicio del programa que, tras alcanzar la cuenta que hayamos predefinido, activa una interrupción que se puede emplear para realizar acciones periódicas. Tras alcanzar la cuenta, el contador se resetea para volver a realizar el ciclo. Es importante aclarar

que la interrupción antes comentada tiene más prioridad que el resto del programa, por lo que nos permite ejecutar un cierto 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 nuestro código, se configuró la interrupción para ejecutarse cada 3 ms.

Cómo se realiza esta función así como los particularidades del PID empleado se verán en el apartado 4.1.4.

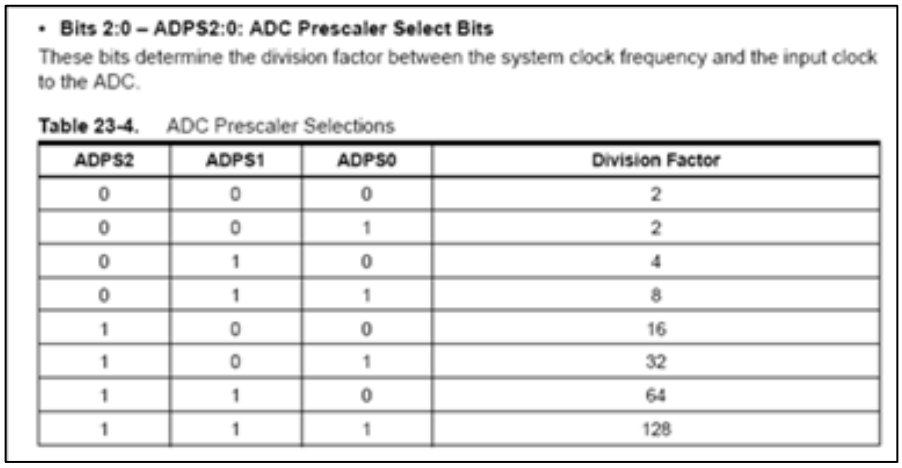
4.1.2 Adquisición de los valores de posición de los motores.

Para realizar el control de los actuadores es necesario conocer su posición a la hora de enviar la señal de control. Como se comentó en el apartado 2.1.2, los motores cuentan con un potenciómetro incorporado cuya resistencia medida varía en función de la distancia que se expande su parte móvil. Al aplicar una diferencia de voltaje de 5V a dicho potenciómetro sabemos que el voltaje medido correspondería con 0V con el pistón retraído completamente y de 5V cuando este se extienda completamente.

La placa Arduino Mega cuenta con ciertos pines capaces de leer voltajes analógicos. A pesar de esto, estas señales analógicas deben ser transformadas en señales digitales para poder ser procesadas. Para ello la placa cuenta con un conversor analógico-digital (CAD, o ADC en inglés) que se encarga de transformar valores analógicos entre 0 y 5 voltios a un número entero entre 0 y 1023. Esta lectura de valores puede llevarse a cabo mediante el comando ‘analogRead(x)’, siendo ‘x’ el pin que queremos leer.

El CAD de Arduino tiene una frecuencia de reloj de 16 MHz dividido por un factor de escala, que, por defecto es 128. Por tanto, la frecuencia por defecto del reloj del CAD sería 16MHz/128 = 125KHz, es decir, 1 ciclo cada 8 us. Una conversión toma 13 ciclos de reloj, por lo que la frecuencia de muestreo total sería de unos 9600 Hz. En nuestra aplicación, esta velocidad es insuficiente para proporcionar los valores al PID todo lo rápido que se debería. Por esto, fue necesario modificar este parámetro.

Para modificar la velocidad del reloj del CAD hay que modificar el factor de escala que se aplica sobre los 16 MHz. Para ello hay que cambiar tres bits del registro ADCSRA: los bits ADPS0, ADPS1 y ADPS2. Según cómo se configuren estos bits podemos cambiar el factor de escala. Aquí podemos ver una tabla:



• **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**

These bits determine the division factor between the system clock frequency and the input clock to the ADC.

Table 23-4. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figura 28. Tabla de configuración para cambiar frecuencia del CAD

En el código se cambió el factor a 64 por lo que la frecuencia de reloj se dobló resultado en 250 KHz, es decir 1 ciclo cada 4 us.

Otro aspecto modificado para optimizar el código fue el multiplexor del CAD. En la placa Arduino existe un multiplexor encargado de seleccionar el pin que debe leer y transformar el CAD. Por defecto, este multiplexor apunta al pin pertinente al realizar un 'analogRead', pero cuando se necesita una alta velocidad de medición, esto puede llegar a ser poco eficiente, ya que se precisa de un tiempo para cargar el valor antes de leerlo. Este tiempo previo se debe a que el CAD, para fijar el valor a convertir, utiliza un condensador que se carga con el voltaje del pin al que apunta el multiplexor. Este condensador tarda un tiempo en cargarse, lo que haría más lenta la conversión. Para mejorar la velocidad de conversión, en el código, tras realizar un 'analogRead', se incluyeron unas líneas que preparaban el multiplexor para la siguiente medición haciendo que apuntaran al siguiente pin, en lugar de esperar hasta realizar una siguiente lectura. De esta forma, mientras se realizan el resto de cálculos que forman el ciclo, se va cargando el condensador con el valor siguiente. Para ello se modificó el registro ADMUX tras cada lectura analógica.

4.1.3 Uso de señales PWM para el control de los motores.

Como se comentó en el apartado 3.1, es posible actuar sobre los motores, aplicando una señal PWM sobre el pin 'Enable' de sus puentes en H asociados. Ciertos pines de Arduino están capacitados para transmitir señales PWM. Estas señales PWM tienen un ciclo de trabajo variable según el valor que se envíe a dichos pines. Este valor debe estar 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.

El valor de las señales PWM empleadas para mover los motores es equivalente al valor de la acción de control resultante de los cálculos del PID. A pesar de esto, evidentemente este valor está limitado a 255 aunque la acción de control sea superior. De esta forma, conseguimos que para una acción de control grande, los motores siempre vayan a máxima velocidad, hasta el momento en que la acción de control tome un valor inferior a 255, momento en que el valor de la PWM y, por tanto, la velocidad de los motores disminuiría.

Al igual que la frecuencia de los CAD, la frecuencia de las señales PWM generadas también tuvo que modificarse. Por defecto, la frecuencia de estas señales está predefinida en 490Hz, excepto en el caso del pin 4, cuyo PWM tiene una frecuencia prefijada de 976Hz. Era necesario aumentar esta frecuencia y para conseguirlo fue necesario modificar ciertos registros, ya que no todos los pines están comandados por los mismos Timers. En nuestra aplicación hemos utilizado los pines 2,3,4,5,6 y 7 para emitir las PWM por lo que se tuvieron que modificar los registros de los Timer 0, 3 y 4, encargados de estos pines (entre otros). Para ello debemos cambiar los tres bits menos significativos de dicho registro denominados CS00, CS01 y CS02. En la siguiente tabla podemos ver las configuraciones posibles para estos pines y el factor de escala que aplica cada una:

Table 11-6. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Figura 29. Tabla de configuración para cambiar frecuencia de PWM

En nuestro caso tuvimos que cambiar el divisor a 8, resultando una frecuencia de PWM de 4000Hz, excepto para el pin 4, cuya frecuencia resultante fue de 7800Hz.

El código necesario para conseguir esto será expuesto en el apartado siguiente.

4.1.4 Descripción del código para Arduino.

Todo programa realizado para una placa de Arduino puede ser dividido en tres partes:

- Una parte inicial dedicada a la inclusión de librerías, definición de macros y declaración de variables globales que se usarán en el resto de partes del programa.
- El bloque 'setup', donde se realizan los configuraciones pertinentes en el procesador para la correcta ejecución del programa según nuestros objetivos. En este bloque se incluyen la definición de los pines necesarios 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 cíclico 'loop', por su parte, alberga el código del programa, es decir, 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. En este bloque también se realizan las comunicaciones mediante puerto serial en las aplicaciones que lo requieran.

A continuación se pasará a comentar todas las partes del código realizado en este proyecto, por separado, aclarando ciertos aspectos del mismo. El código completo también será incluido como anexo de la memoria.

4.1.4.1 Inclusión de librerías, definición de macros y declaración de variables.

En esta parte inicial del código se incluye la librería 'TimerOne', se definen varias macros empleadas en algunas partes del código posterior y se declaran las variables globales. A continuación se muestra este fragmento del código:

```
#include "TimerOne.h"

int entradal[6]={48,44,40,36,32,28};     //Entradal de cada motor
int entrada2[6]={49,45,41,37,33,29};     //Entrada2 de cada motor
int enable[6]={7,6,5,4,3,2};             //Enable de cada motor (PWM)

int saltarprint;         //Usada para mostrar la posicion del motor cada cierto tiempo

//Variables para el controlador:
float kp = 180;           //Constante proporcional
float qi = 0.1;           //Constante integral discreta
float qd = 180;           //Constante derivativa discreta
float P[6];                    //Parte proporcional
float I[6]={0,0,0,0,0,0};      //Parte integral
float D[6]={0,0,0,0,0,0};      //Parte derivativa

float consigna[6]={2.5,2.5,2.5,2.5,2.5,2.5};    //Consignas de posicion en voltios

float posicion[6];        //Voltajes obtenidos del sensor de posicion de cada motor

float error0[6];          //Errores previos e iniciales
float errorl[6];          //Errores actuales

float salidaMotor[6]={0,0,0,0,0,0};   //Salidas que controlan los motores

unsigned int consignaInt[6];
int indice=0;

//-----------Para cambiar la frecuencia del CAD---------
#define FASTADC 1

// defines for setting and clearing register bits
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif
```

4.1.4.2 Bloque ‘setup’.

En este bloque se realizan las configuraciones de los pines, declarándolos como entradas o salidas del sistema. Para cada puente en H de los motores son necesarias dos señales con las que indicar el sentido en que debe ir el motor (‘entrada1’ y ‘entrada2’ como output) y una última señal que transporte la PWM para regular la velocidad de estos (‘enable’ como output). Por otro lado se definen como inputs las señales analógicas que utilizaremos para conocer la posición de los motores. Al final del bloque ‘setup’ también podremos ver la inicialización de un timer de 100ms, necesario para definir las interrupciones cada 3ms en la siguiente línea del código. Podemos ver cómo estas interrupciones están asociadas a la ejecución de una función llamada ‘control’ encargada de leer las posiciones de los motores y realizar su control. Esta función será analizada en el apartado 4.1.4.3. Comprobar también, que en este bloque se inicializa el puerto serial con una velocidad de transmisión de 9600 baudios.

```
Serial.begin(9600);

pinMode(entrada1[0], OUTPUT);   //PIN28: entrada1 motor1
pinMode(entrada2[0], OUTPUT);   //PIN29: entrada2 motor1
pinMode(enable[0], OUTPUT);     //PIN2: enable motor1

pinMode(entrada1[1], OUTPUT);   //PIN32: entrada1 motor2
pinMode(entrada2[1], OUTPUT);   //PIN33: entrada2 motor2
pinMode(enable[1], OUTPUT);     //PIN3: enable motor2

pinMode(entrada1[2], OUTPUT);   //PIN36: entrada1 motor3
pinMode(entrada2[2], OUTPUT);   //PIN37: entrada2 motor3
pinMode(enable[2], OUTPUT);     //PIN4: enable motor3

pinMode(entrada1[3], OUTPUT);   //PIN40: entrada1 motor4
pinMode(entrada2[3], OUTPUT);   //PIN41: entrada2 motor4
pinMode(enable[3], OUTPUT);     //PIN5: enable motor4

pinMode(entrada1[4], OUTPUT);   //PIN44: entrada1 motor5
pinMode(entrada2[4], OUTPUT);   //PIN45: entrada2 motor5
pinMode(enable[4], OUTPUT);     //PIN6: enable motor5

pinMode(entrada1[5], OUTPUT);   //PIN48: entrada1 motor6
pinMode(entrada2[5], OUTPUT);   //PIN49: entrada2 motor6
pinMode(enable[5], OUTPUT);     //PIN7: enable motor6

pinMode(A1, INPUT);
pinMode(A2, INPUT);
pinMode(A3, INPUT);
pinMode(A4, INPUT);
pinMode(A5, INPUT);
pinMode(A6, INPUT);

Timer1.initialize(100000);        //Inicializamos un timer de 100ms
Timer1.attachInterrupt(control, 3000);
}
```

4.1.4.3 Función ‘control’.

Esta función es ejecutada cada 3ms ya que esta comandada por la interrupción generada en el bloque ‘setup’. La función ‘control’ es la encargada de realizar las lecturas de los potenciómetros de los actuadores, así como de realizar su control mediante PID. Esta función la estudiaremos de forma desglosada para ir explicando cada una de sus partes.

Inicialmente se genera un bucle for de 6 iteraciones con el fin de resumir el código, ya que con él, escribiendo las acciones que se realizarán en un solo motor, será suficiente para ejecutarlo en los 6 motores. La primera operación a realizar en el bucle es la lectura del valor analógico del actuador para, a continuación, actuar sobre el registro del multiplexor de los CAD para dejarlo preparado para la lectura siguiente.

Finalmente, se realiza una conversión de escala para el valor medido ‘posicion’. Con esto pasamos de tener valor mínimo 0 y máximo 1023 a trabajar en un rango de valores entre 0 y 5. Esto no es realmente necesario para hacer el cálculo

del control, simplemente se ha hecho para manejar valores comparables a los valores reales de posición en centímetros o al voltaje obtenido del potenciómetro de los actuadores.

```
void control(){

  int temp;
  for(int i=0; i<6; i++){

  //Leemos sensor y transformamos valor entre 0 y 5
  //------------Para ir preparando la siguiente medida analogica--------------
  temp=analogRead(i+1);
  if ( i == 6)
    ADMUX = (ADMUX & 0xE0) | 1;
  else
    ADMUX = (ADMUX & 0xE0) | ((i+2));
  posicion[i] = temp*5.0/1024.0;
```

El siguiente paso consiste en realizar el cálculo de la acción de control del PID (‘salidaMotor’) de la forma que se comentó en el apartado 4.1.1. Como información adicional comentar que la aportación de la acción integral ha sido limitada a 25 para evitar un aumento indefinido de la misma a lo largo del tiempo. Además de esto, se ha bloqueado las acciones integral y derivativa en los casos en que la acción integral sea inferior al límite, es decir, menor de 255, con el fin de minimizar el cálculo realizado por el procesador. Recordar que el valor de las tres constantes necesarias para este cálculo (‘kp’, ‘qi’ y ‘qd’) fue aportado en la declaración de estas variables, al principio del código, y que fue obtenido mediante ensayo y error, como se explicó en el apartado 4.1.1.

```
  //Lazo de control (controlador PID)
  error0[i] = posicion[i] - consigna[i];
  P[i] = kp*error0[i];

  if(fabs(salidaMotor[i])<255){    //Aplicamos parte integral solo cuando estemos cerca de la consigna
  I[i] = I[i]+qi*error0[i];

  if (I[i] >= 25)
    I[i] = 25;
  else if (I[i] <= -25)
    I[i] = -25;
  }

  if(fabs(salidaMotor[i])<255){
  D[i] = qd*(error0[i]-error1[i]);   //Aplicamos parte derivativa solo cuando estemos cerca de la consigna
  }

  salidaMotor[i] = P[i]+I[i]-D[i];

  error1[i]=error0[i];
```

Para finalizar, se genera la señal PWM que controlará la velocidad del motor así como las señales que indicarán el sentido de giro, en función de la acción de control obtenida en el cálculo previo. Este código se divide en dos, uno para el avance del motor (con ‘entrada1’ a nivel bajo y ‘entrada2’ a alto), y otro para su

retroceso (con las entradas de forma inversa al de avance). En este paso también se limita el valor 'salidaMotor' a 255 para no sobrepasar el valor máximo que puede tomar la señal PWM.

```
  //----------------Salida Motores-------------------------

 if (salidaMotor[i] > 0.0) { // Mover el motor en avance
    if (salidaMotor[i] >= 255.0)
      salidaMotor[i] = 255.0;

    digitalWrite(entrada2[i], HIGH);
    digitalWrite(entrada1[i], LOW);

  analogWrite(enable[i], round(fabs(salidaMotor[i])));
 }//end if(avance)

 else {   //Mover el motor en retroceso
   if (salidaMotor[i] <= -255.0)
     salidaMotor[i] = -255.0;

   digitalWrite(entrada2[i], LOW);
   digitalWrite(entrada1[i], HIGH);

  analogWrite(enable[i], round(fabs(salidaMotor[i])));
 }// end else(retroceso)

 }//end for(i)

}//end void control
```

4.1.4.4 Bloque 'loop'.

Este bloque está destinado a la recolección de las consignas que serán calculadas y aportadas por el programa diseñado en MATLAB. Para esto se utilizará comunicación serial mediante un cable USB entre el PC y Arduino. Los valores de las consignas están formados por dos bytes que son interpretados y traducidos por Arduino a valores decimales con los que realizar el cálculo de la acción de control.

Para guardar los valores recibidos por serial se he declarado una variable vectorial 'buffer'. En esta variable los datos recibidos se almacenan por bytes, por lo que albergaría doce bytes de datos para las consignas de los seis motores y un byte extra con el valor '10'. El código se ha realizado de forma que solo se procesen los datos cuando el byte 13 tenga el valor '10'. De esta forma podemos comprobar si los datos se han enviado de forma correcta.

Al recibir una secuencia correcta de valores, se procede a reescribir las consignas como números enteros, acoplando los dos bytes de cada consigna almacenados en 'buffer'. Las consignas reformuladas como enteros son guardadas en la variable 'consignaInt'. Como estos valores se encuentran en un rango entre 0 y l023, el último paso es realizar la conversión de escala de estos a un rango entre 0 y

5, como se realizó con la variable 'posicion'. Estos valores finales son guardados en la variable 'consigna'.

```
void loop(){

  //Establecemos consignas
   int buffer[25];

  int i=0;

   if (Serial.available() > 0) { // envia datos solamente cuando recibe datos

        for (int j = 0; j < 13; j++) {
          while (Serial.available() == 0)
            ;

          buffer[j] = Serial.read();
          delayMicroseconds(100);

          Serial.print("Buffer ");
          Serial.print(j);
          Serial.print(": ");
          Serial.println(buffer[j]);
        }

        if (buffer[12] == 10) { // Paquete correcto.
          for (int j=0; j < 6; j++) {
            unsigned int temp;
            temp = buffer[j*2];
            temp = temp << 8;
            temp = temp + buffer[j*2+1];
            consignaInt[j] = temp;
          }


         for (int j=0; j < 6; j++) {
             consigna[j] = 5.0*consignaInt[j]/1024.0;
          }
          i = 0;
        }

    }//end if(Serial)
}
```

4.2 Programación en MATLAB. Cálculo de consignas y comunicación con Arduino.

MATLAB es un lenguaje de alto nivel y un entorno interactivo para el cálculo numérico, la visualización y la programación. Mediante MATLAB, es posible manipular matrices, analizar datos, desarrollar algoritmos y crear modelos o aplicaciones. MATLAB emplea un lenguaje de programación propio denominado lenguaje M y se puede utilizar en una gran variedad de aplicaciones, tales como procesamiento de señales y comunicaciones, procesamiento de imagen y vídeo,

sistemas de control, pruebas y medidas, finanzas computacionales y biología computacional. Entre otras de sus capacidades se encuentra la posibilidad de crear interfaces de usuario y la comunicación con software o hardware externo.

El código realizado en MATLAB para nuestra plataforma consta de dos partes: una encargada de realizar la comunicación por serial con Arduino y otra con el fin de calcular las consignas que deberían alcanzar los actuadores para mover la plataforma móvil a un punto, cuya posición y orientación sería escogida por el usuario.

4.2.1 Comunicación entre Arduino y MATLAB.

Esta parte del código es la encargada de llamar a la función que calcula las consignas, para después enviar estos valores a Arduino.

En la primera parte del código se procede a crear un objeto serial de nombre 'arduserial', indicando el puerto al que está conectada la placa Arduino y la velocidad de comunicación establecida en el código de dicha placa. En la siguiente línea se abre el puerto serial mediante el comando 'fopen'.

```
clear all;
delete(instrfind({'Port'},{'COM9'}));
arduserial = serial('COM9','BaudRate',9600);
warning('off','MATLAB:serial:fscanf:unsuccessfulRead');
fopen(arduserial);
```

En el siguiente fragmento, se declaran las variables externas que se deben proporcionar a la función 'ik'(cuyo funcionamiento se explicará en el apartado siguiente) como argumentos de entrada para el cálculo de las consignas. El significado de las variables se puede ver comentado en el código. Es importante comentar que sólo una de las variables ('pose') es la que incluye los valores de posición y ángulo que tomaría la base móvil. Las otras variables (A y B) sirven para aportar información física de nuestra plataforma a la función y, por tanto, siempre tendrían el mismo valor. Para asignar los valores a 'A' y 'B' debemos medir los terminales superiores de los motores con respecto a la base móvil, para 'A' y los inferiores con respecto a la base fija, para 'B'. Aquí podemos ver dicho fragmento del código:

```
%A=coordenadas del terminal superior de cada pata desde
%plataforma superior (matriz 3*6)

%B=coordenadas del terminal inferior de cada pata desde
%plataforma inferior (matriz 3*6)

%pose=posicion final del centro de la plataforma superior
%respecto a la inferior. Traslacion y rotacion (3 coordenadas y 3 angulos)

A=[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]; %valor fijo
B=[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]; %valor fijo
pose=[0; 0; 16; 0; 0; 0]; %valor variable
```

Los valores aportados a las variables 'A' y 'B' se han medido de forma aproximada. En la siguiente imagen podemos ver las coordenadas medidas así como la altura entre las bases (en la posición más baja alcanzable) y la longitud mínima de cada 'pata' del motor. En esta imagen se muestran los anclajes de cada motor, numerados. Números iguales indican que los anclajes pertenecen al mismo motor.

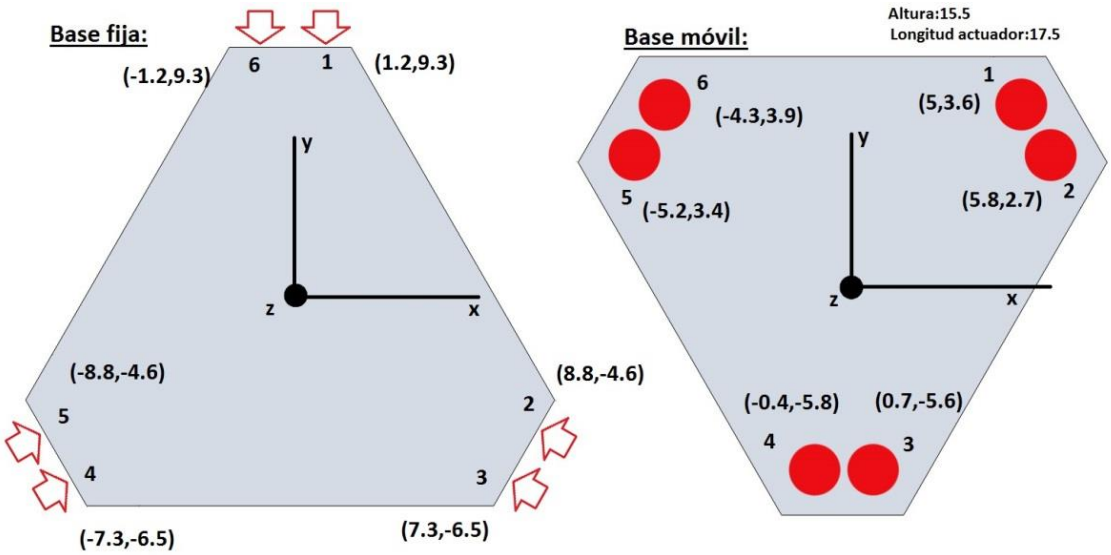


Figura 30. Coordenadas aproximadas de los anclajes de los actuadores en cm

Estos datos también se muestran tabulados a continuación, para una mejor localización:

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 31. Tabla de coordenadas de los anclajes y longitudes importantes del robot

Como podemos comprobar, los valores asignados a las variables 'A' y 'B' se corresponden con las coordenadas expuestas:

- A=[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=[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];

La siguiente línea del código tiene como objetivo llamar a la función 'ik', encargada de realizar el cálculo de las consignas mediante el estudio de la cinemática inversa del robot. El vector devuelto por la función 'ik' es guardado en la variable 'consigna'. En este momento, la variable 'consigna' tendría los valores de las longitudes que deberían tomar las 6 'patas' del robot. Como siempre existe una longitud mínima de 17,5 cm derivada de la longitud del actuador y las articulaciones, debemos restar este valor a los aportados por la función 'ik'. Al hacer esto, nos quedaríamos solo con la longitud que debería extenderse (o retraerse) cada motor. Inmediatamente después, se cambia de escala los valores de estas consignas para que se encuentren entre 0 y 1023, en lugar de entre 0 y 5. Si el resultado de la cinemática inversa devuelve valores fuera de este rango, dichos valores se descartarán por ser inalcanzables, como veremos en el posterior fragmento de código.

```
consigna = ik(A, B, pose);
for i = 1:6
    consigna(i) = consigna(i)-17.5;
    consigna(i) = 1024*consigna(i)/5;
end
```

A continuación se realiza la comprobación de seguridad, para asegurarnos de enviar las consignas solo cuando estas se encuentren entre 0 y 1023.

```
temp = 0;
for i = 1:6
    if (consigna(i)<0 || consigna(i)>1023)
        temp++;
    end
end
```

El siguiente paso consiste en escribir por el puerto serial, el valor de las consignas obtenidas en la línea anterior (solo si todas las consignas son válidas). Para ello se emplea el comando 'fwrite'. En nuestro caso, debemos enviar las consignas en forma de bytes individuales. Por ello, se realizan dos cálculos con los que obtener

el byte más significativo y menos significativo de cada consigna. La palabra clave 'uint8' indica que el valor que se envía es un número de 8 bits sin signo.

```
if temp == 0
    pause(1);
    for i = 1:6
        fwrite(arduserial, floor(consigna(i)/256),'uint8');
        fwrite(arduserial, bitand(consigna(i),255),'uint8');
    end
    fwrite(arduserial, 10,'uint8');
end
```

Para finalizar, se cierra el puerto serial y se elimina el objeto serial 'arduserial'.

```
fclose(arduserial);
delete(arduserial);
clear arduserial;
```

4.2.2 Cálculo de las consignas para los actuadores.

Para realizar el cálculo de las consignas se emplea un programa que hace uso de la cinemática inversa de la plataforma. Por tanto, antes de pasar a comentar el código se realizará una introducción a la cinemática inversa del robot.

4.2.2.1 Nociones sobre cinemática inversa de la plataforma de Stewart.

La cinemática inversa en robótica es un problema que busca encontrar los ángulos y desplazamientos que deben tomar los actuadores para alcanzar un punto y rotación específicos en el actuador final. Por esto, la cinemática inversa es la herramienta más útil para conseguir nuestros objetivos en este proyecto ya que, nos permitiría averiguar cuánto se debe extender cada motor para alcanzar la posición y rotación que deseemos con la base móvil. En nuestro caso, el cálculo de la cinemática inversa es llevado a cabo por la función 'ik' mencionada anteriormente y que será explicada en profundidad en el siguiente apartado.

Para nuestro problema de cinemática inversa, es necesario calcular una matriz de transformación con la que podamos relacionar la articulación final con la inicial. Esta matriz de transformación permite referenciar un sistema de referencia solidario a la base móvil con el sistema de referencia de la base fija mediante rotaciones y traslaciones. Con esto, se puede deducir que la matriz de transformación está formada por una matriz de rotación y un vector de posición, así como por un vector de cambio de perspectiva y un elemento denominado factor de escala. Tanto el cambio de perspectiva como el factor de escala no tienen sentido en esta aplicación por lo que tomarían el valor 0 y 1 respectivamente.

Por tanto, una matriz de transformación que describa un sistema de referencia '1' con respecto a otro '0' debería quedar de forma genérica como se muestra a continuación:

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

donde: **Rojo** = matriz de rotación

Verde = matriz de traslación

Azul = vector de perspectiva

Violeta = factor de escala

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. Para obtener la matriz de rotación final, se puede estudiar por separado cada uno de los giros, para finalmente incluir 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).

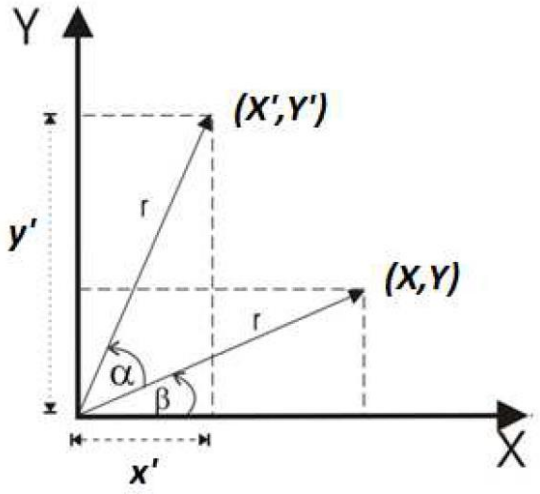


Figura 32. Giro de un vector en torno al eje z (roll)

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.

4.2.2.2 Código en MATLAB.

El cálculo de las consignas es 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 Estabilización 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.

Este programa se encarga de calcular la cinemática inversa de la plataforma, por lo que solo es necesario aportarle el punto final que queremos alcanzar, así como los ángulos de rotación deseados (alabeo, cabeceo y giro). El código también precisa de cierta información de la plataforma para el cálculo, como la posición de las articulaciones inferiores respecto a la base fija y la posición de las articulaciones superiores respecto a la base móvil. A continuación se procederá a la descripción del código. A pesar de esto, se aportará la totalidad del código como un anexo de la memoria.

Al comienzo del código se inicia la función 'ik' que realiza el cálculo de las consignas. En la parte inicial de la función se realizan la definición de las variables locales empleando los argumentos de entrada de la función, anteriormente descritos.

```
function [L] = ik(A, B, pose)    % A y B en coordenadas homogeneas
    A = A(1:3,:);
    B = B(1:3,:);
    R = pose(4:6,:);
    D = pose(1:3,:);
    Rrad = (R.*pi)./180; % De grados a radianes
    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;
```

La variable 'R' se correspondería con los valores angulares del actuador final y 'D' con sus coordenadas espaciales. El vector 'R', inicialmente en grados, es transformado a radianes dando lugar a la variable 'Rrad'. Como podemos ver, también se realizan algunos cálculos previos. Estos cálculos también se podrían hacer posteriormente al calcular la matriz de rotación pero, guardando los diferentes resultados en variables nos ahorramos muchos cálculos repetitivos, pudiendo conseguir velocidades de procesamiento mayores.

En el fragmento final del código se calcula la matriz de rotación y las longitudes que deberían tener las 6 articulaciones de la plataforma.

```
    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];

    L = sqrt(sum( (RR * A + D(:,ones(6,1)) - B).^2 ));
end
```

Como se puede apreciar, la matriz de rotación toma una forma similar a la expuesta en el apartado 4.2.2.1. Las variaciones entre ambas matrices se deben a las convenciones utilizadas a la hora de diseñar el programa, como el sentido de giro que se tomaría como positivo.

Finalmente se calculan las 6 longitudes que deberían tomar las articulaciones, que, como ya sabemos, varían en función de la extensión de los motores. Estos valores se guardan en un vector 'L', que sería devuelto, como resultado de la función 'ik'. El cálculo de 'L' consiste en aplicar la distancia euclídea, derivada del teorema de Pitágoras, para obtener las longitudes, ya que se conocerían los puntos que tomarían los extremos superiores de los motores, gracias a la matriz de rotación anterior, y, por supuesto se conocen sus extremos inferiores. Este cálculo consistiría en la raíz cuadrada de la suma de sus componentes al cuadrado, siempre con sistema de referencia en la base inferior. También podemos comprobar que en este cálculo final se incluye el vector de traslación, por lo que también se tomaría en cuenta este tipo de movimiento (si lo hubiera) para el resultado final.

5. Conclusiones del Trabajo Fin de Grado.

5.1 Conclusión.

El desarrollo de este proyecto, personalmente, me ha servido como un mayor acercamiento al ámbito profesional relacionado con la ingeniería, en general, y la ingeniería electrónica y automatización, en particular. El desarrollo de actividades relacionadas con la investigación, experimentación, diseño, fabricación y realización de los documentos técnicos ha sido una gran forma de aplicar los conocimientos adquiridos en los últimos años de manera práctica.

En este proyecto, se han empleado y desarrollado habilidades y conocimientos muy variados como:

- El estudio analítico de la cinemática del robot y el desarrollo de un algoritmo de control con el que posicionarlo de forma correcta.
- La programación en diferentes lenguajes y la abstracción necesaria que conlleva el hecho de plasmar el funcionamiento de un sistema a través de estos medios.
- El diseño 2D y 3D de piezas y mecanismos en general.
- El diseño esquemático y físico de circuitos electrónicos, con los procesos que esto conlleva, como la búsqueda de los mejores componentes, realizar los cálculos necesarios para su interconexión de forma correcta y segura, o algo tan simple como mejorar la técnica de soldadura.
- La búsqueda de la documentación con la que recopilar la información necesaria para la redacción de la memoria.

En definitiva, ha sido una excelente forma de asentar las bases aprendidas a lo largo de la carrera.

5.2 Conclusion.

The realization of this project, personally, has served me as closer to the professional field related to engineering, in general, and electronic engineering and automation, in particular. The development of activities related with research, experimentation, design, manufacture and writing of technical documents has been a great way to apply the knowledge gained in recent years practically.

In this project, have been used and developed skills and knowledge varied as:

- The analytical study of the robot kinematic and the development of a control algorithm with which position it correctly.
- Programming in different languages and abstraction needed to capture the performance of a system through these means.
- The design of pieces and mechanisms in 2D and 3D.
- The schematic and physical design of electronic circuits, with the processes that this entails, such as finding the best components, perform the calculations necessary for its interconnection properly and safely, or something as simple as improving the soldering technique.
- The search for the documentation with which to collect the information necessary for the writing of the memory.

In short, it was a great way to set the bases learned throughout the career.

6. Referencias.

- Arduino. Plataforma de hardware libre.
  - Sitio web: http://www.arduino.cc/
  - Arduino Mega 2560: http://arduino.cc/en/Main/ArduinoBoardMega2560
  - Referencia para lenguaje de programación de Arduino: http://arduino.cc/en/Reference/HomePage
  - Descarga del entorno de desarrollo: http://arduino.cc/en/Main/Software
- Matlab. Software matemático. Empresa: MathWorks.
  - Sitio web: http://www.mathworks.es/
  - Documentation Center: http://www.mathworks.es/es/help/index.html
  - Descarga del programa: http://www.mathworks.es/products/matlab/
- OrCAD. Suite de herramientas para el diseño esquemático de circuitos electrónicos y creación de PCB's. Empresa: Cadence Design Systems.
  - Sitio web: http://www.orcad.com/
- Ignacio Peñarrocha. "Implementación digital de controladores PID". 16 de Enero de 2006.
- Zafer Bingul, Oguzhan Karahan. "Dynamic Modeling and Simulation of Stewart Platform". Mechatronics Engineering, Kocaeli University (Turkey).
- Cambiar frecuencia de PWM en Arduino Mega 2560: http://forum.arduino.cc/index.php/topic,72092.0.html http://sobisource.com/arduino-mega-pwm-pin-and-frequency-timer-control/
- Cambiar frecuencia del conversor analógico digital en Arduino Mega 2560: http://forum.arduino.cc/index.php/topic,6549.0.html
- 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.

- Datasheet articulaciones empleadas.
    - Rótulas:

      http://docs-europe.electrocomponents.com/webdocs/005b/0900766b8005b198.pdf

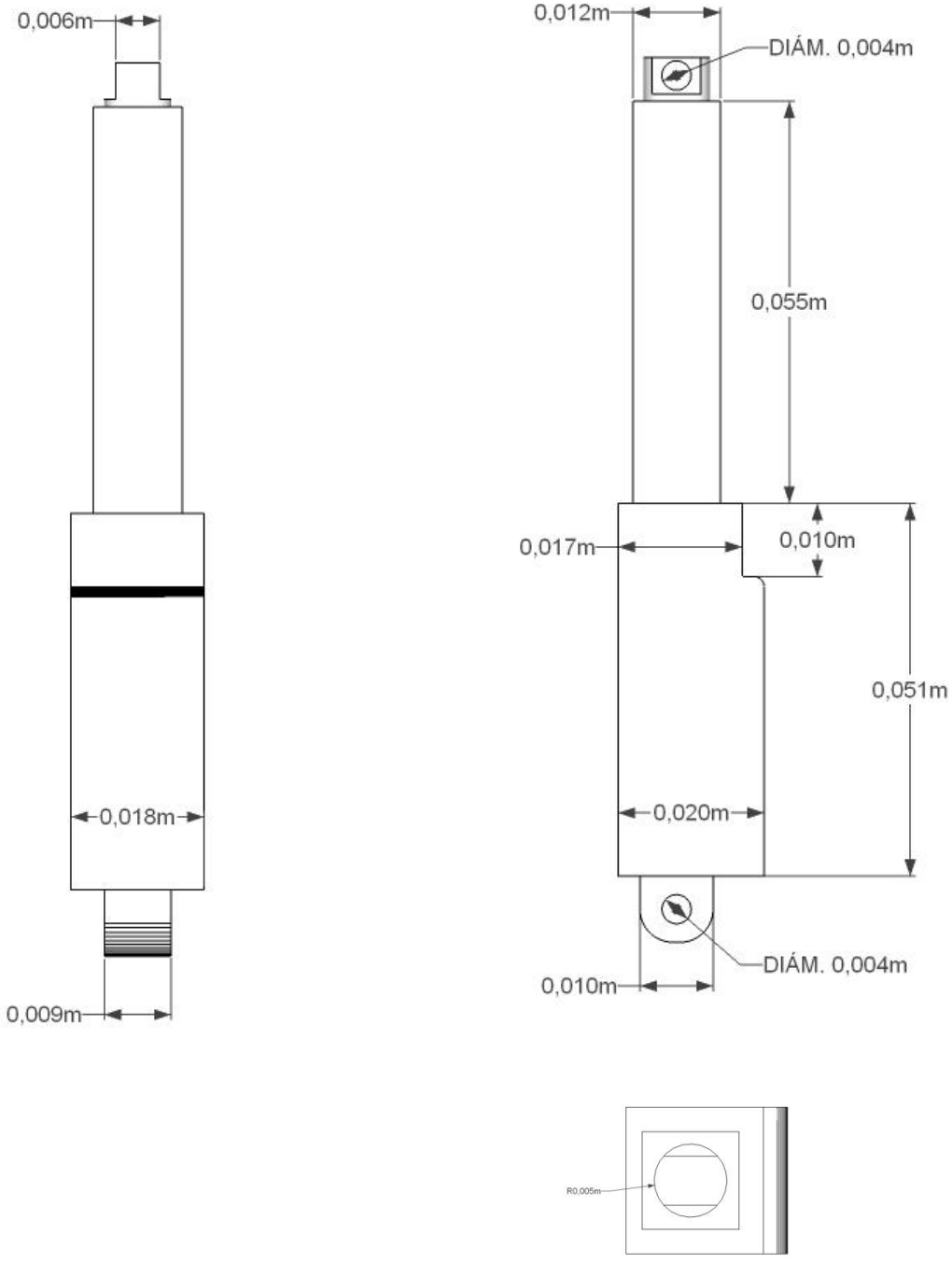
    - Cardanes:

      http://docs-europe.electrocomponents.com/webdocs/008b/0900766b8008b7de.pdf
- Figura 6. Motor lineal Firgelli, serie L16: http://www.firgelli.com/Uploads/L16_datasheet.pdf
- Figura 8. Ubicación de actuadores y articulaciones: http://paginas.fe.up.pt/~aml/Stewart.htm
- Figura 16. Configuración LM2575 para voltaje de salida de 5V: http://www.onsemi.com/pub_link/Collateral/LM2575-D.PDF (página 7)
- Figura 17. Composición de un L298 (2 puentes en H): https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf (página 1)
- Figura 28. Tabla de configuración para cambiar frecuencia del CAD / Figura 29. Tabla de configuración para cambiar frecuencia de PWM / Cambiar frecuencia de CAD y PWM: https://cdn.sparkfun.com/datasheets/Components/General%20IC/mega2560.pdf

7. Anexos.

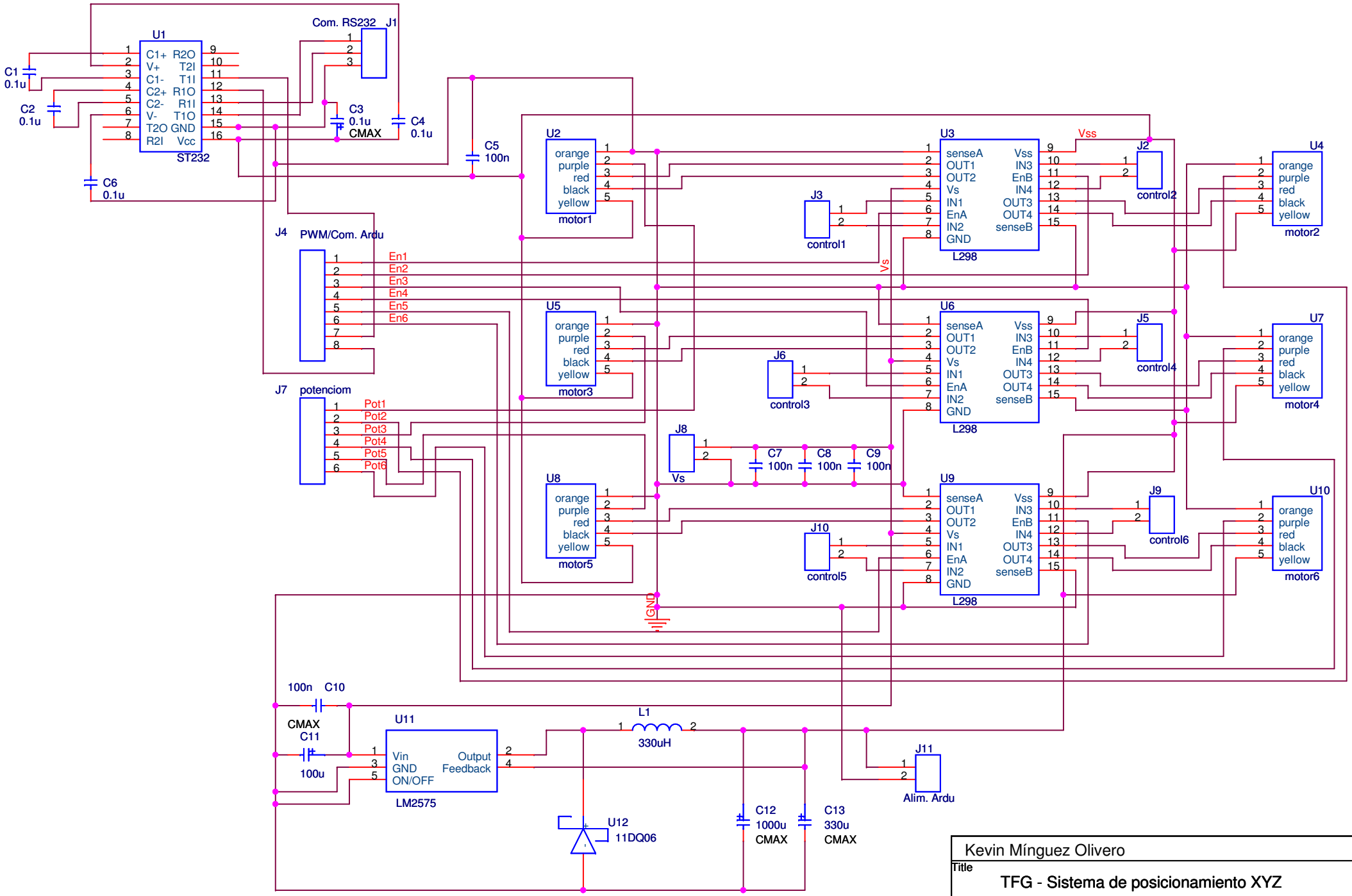
Anexo 1

Plano de motor empleado


0,006m
0,018m
0,009m
0,012m
DIÁM. 0,004m
0,055m
0,017m
0,010m
0,051m
0,020m
DIÁM. 0,004m
0,010m
R0,005m

Anexo 2

Diseño esquemático de placa PCB


U1
C1+ R2O
V+ T2I
C1- T1I
C2+ R1O
C2- R1I
V- T1O
T2O GND
R2I Vcc
ST232
C1 0.1u
C2 0.1u
C6 0.1u
Com. RS232 J1
C3 0.1u CMAX
C4 0.1u
C5 100n
J4 PWM/Com. Ardu
En1
En2
En3
En4
En5
En6
J7 potenciom
Pot1
Pot2
Pot3
Pot4
Pot5
Pot6
U2
orange
purple
red
black
yellow
motor1
U5
motor3
U8
motor5
J3
control1
J6
control3
J10
control5
U3
senseA
OUT1
OUT2
Vs
IN1
EnA
IN2
GND
Vss
IN3
EnB
IN4
OUT3
OUT4
senseB
L298
U6
U9
J2
control2
J5
control4
J9
control6
U4
motor2
U7
motor4
U10
motor6
Vss
Vs
GND
J8
Vs
C7 100n
C8 100n
C9 100n
100n C10
CMAX
C11
100u
U11
Vin
GND
ON/OFF
Output
Feedback
LM2575
L1
330uH
U12
11DQ06
C12 1000u CMAX
C13 330u CMAX
J11
Alim. Ardu
Kevin Mínguez Olivero
ULL
Title
TFG - Sistema de posicionamiento XYZ
Size A4
Document Number
Rev
Date: Monday, August 25, 2014
Sheet 1 of 1

Anexo 3

Diseño físico de placa PCB

Anexo 3.1

Capa Bottom

XYZ Robot - TFB - Olivero Mingues Kevin

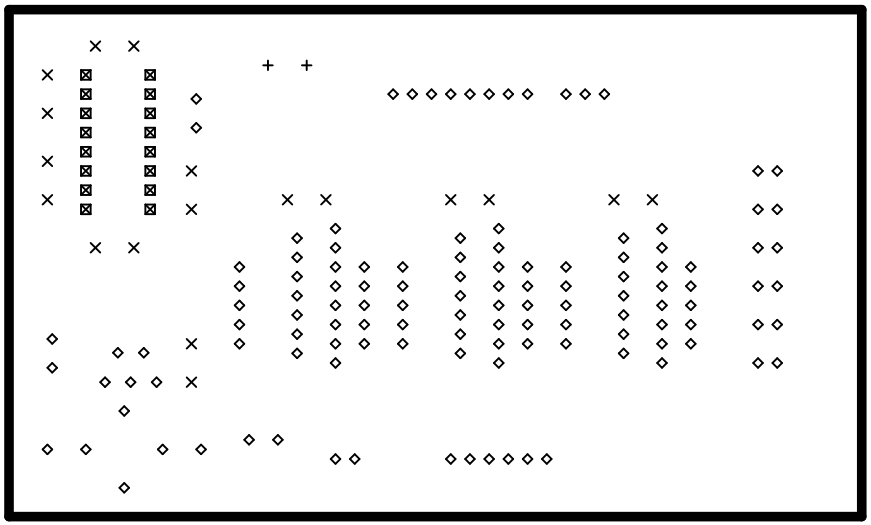
Anexo 3.2

Capa Top

Kevin Mingues Olivero - TFG - Robot XY1

Anexo 3.3

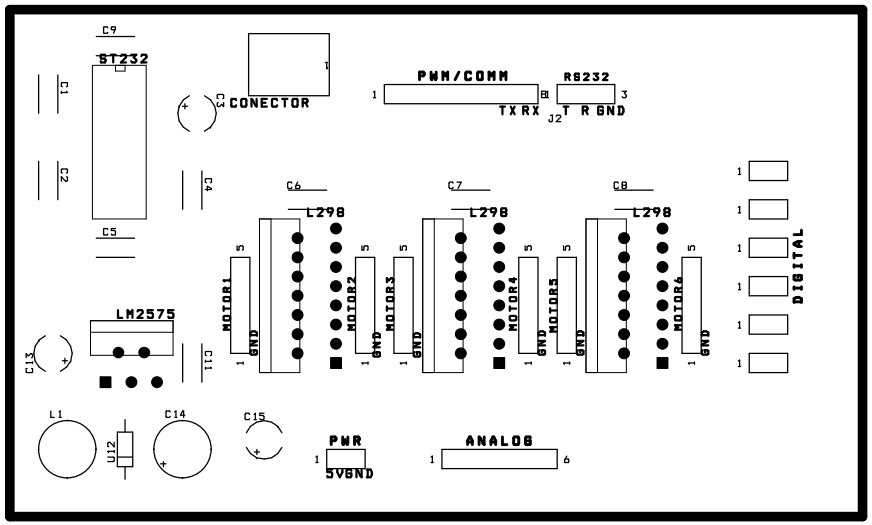
Plano de taladros



DRILL CHART				
SYM	DIAM	TOL	QTY	NOTE
×	0.800 mm		18	
⊠	1.000 mm		16	
◇	1.200 mm		123	
+	1.300 mm		2	
TOTAL			159	

Anexo 3.4

Plano de Serigrafía (capa Top)


C9
ST232
C1
C3
C2
C4
C5
CONECTOR
PWM/COMM
RS232
TX RX
J2
T R GND
C6
C7
C8
L298
L298
L298
MOTOR1
MOTOR2
MOTOR3
MOTOR4
MOTOR5
MOTOR6
GND
DIGITAL
LM2575
C13
C11
L1
C14
C15
U12
PWR
5VGND
ANALOG

Anexo 4

Código Arduino

```
#include "TimerOne.h"

intentrada1[6]={48,44,40,36,32,28};      //Entrada1 de cada motor
intentrada2[6]={49,45,41,37,33,29};      //Entrada2 de cada motor
int enable[6]={7,6,5,4,3,2};             //Enable de cada motor (PWM)

int          saltarprint;//Usada para mostrar la posicion del motor cada cier

//Variables para el controlador:
float    kp      =     180;//Constante proporcional
float    qi      =     0.1;//Constante integral discreta
float    qd      =     180;//Constante derivativa discreta
float                    P[6];//Parte proporcional
float      I[6]={0,0,0,0,0,0};//Parte integral
float      D[6]={0,0,0,0,0,0};//Parte derivativa

float    consigna[6]={2.5,2.5,2.5,2.5,2.5,2.5};//Consignas de posicion en vo

float         posicion[6];//Voltajes obtenidos del sensor de posicion de cada

float           error0[6];//Errores previos e iniciales
float           error1[6];//Errores actuales

float   salidaMotor[6]={0,0,0,0,0,0};//Salidas que controlan los motores

unsigned int consignaInt[6];
int indice=0;

//-----------Para cambiar la frecuencia del CAD---------
#define FASTADC 1

// defines for setting and clearing register bits
#ifndef cbi
#define  cbi(sfr,bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define  sbi(sfr,bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif


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

void setup() {

//-----------Para cambiar la frecuencia de los CAD---------
#if FASTADC
  // set prescale to 64
  sbi(ADCSRA,ADPS2) ;
  sbi(ADCSRA,ADPS1) ;
```

```
  cbi(ADCSRA,ADPS0) ;
#endif
//-------------------------------------------------------------

//------------Para cambiar la frecuencia de las PWM---------
int   myEraser   =   7;// this is 111 in binary and is used as an eraser
TCCR4B  &=  ~myEraser;// this operation (AND plus NOT),  set the three bits

int  myPrescaler   =   2;// this could be a number in [1 , 6]. In this case,
TCCR0B  |=  myPrescaler;//this operation (OR), replaces the last three bits in
TCCR3B  |=  myPrescaler;//this operation (OR), replaces the last three bits in
TCCR4B  |=  myPrescaler;//this operation (OR), replaces the last three bits in
//-------------------------------------------------------------

Serial.begin(9600);

pinMode(entrada1[0], OUTPUT);    //PIN28: entrada1 motor1
pinMode(entrada2[0], OUTPUT);    //PIN29: entrada2 motor1
pinMode(enable[0], OUTPUT);     //PIN2: enable motor1

pinMode(entrada1[1], OUTPUT);    //PIN32: entrada1 motor2
pinMode(entrada2[1], OUTPUT);    //PIN33: entrada2 motor2
pinMode(enable[1], OUTPUT);     //PIN3: enable motor2

pinMode(entrada1[2], OUTPUT);    //PIN36: entrada1 motor3
pinMode(entrada2[2], OUTPUT);    //PIN37: entrada2 motor3
pinMode(enable[2], OUTPUT);     //PIN4: enable motor3

pinMode(entrada1[3], OUTPUT);    //PIN40: entrada1 motor4
pinMode(entrada2[3], OUTPUT);    //PIN41: entrada2 motor4
pinMode(enable[3], OUTPUT);     //PIN5: enable motor4

pinMode(entrada1[4], OUTPUT);    //PIN44: entrada1 motor5
pinMode(entrada2[4], OUTPUT);    //PIN45: entrada2 motor5
pinMode(enable[4], OUTPUT);     //PIN6: enable motor5

pinMode(entrada1[5], OUTPUT);    //PIN48: entrada1 motor6
pinMode(entrada2[5], OUTPUT);    //PIN49: entrada2 motor6
pinMode(enable[5], OUTPUT);     //PIN7: enable motor6

pinMode(A1, INPUT);
pinMode(A2, INPUT);
pinMode(A3, INPUT);
pinMode(A4, INPUT);
pinMode(A5, INPUT);
pinMode(A6, INPUT);

Timer1.initialize(100000);        //Inicializamos un timer de 100ms
Timer1.attachInterrupt(control, 3000);
```

```
}

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

void control(){

  int temp;
  for(int i=0; i<6; i++){

  //Leemos sensor y transformamos valor entre 0 y 5
  //------------Para ir preparando la siguiente medida analogica-------------
  temp=analogRead(i+1);
  if ( i == 6)
    ADMUX = (ADMUX & 0xE0) | 1;
  else
    ADMUX = (ADMUX & 0xE0) | ((i+2));
  posicion[i] = temp*5.0/1024.0;
  //--------------------------------------------------------------------------

  //Lazo de control (controlador PID)
  error0[i] = posicion[i] - consigna[i];
  P[i] = kp*error0[i];

  if(fabs(salidaMotor[i])<255){    //Aplicamos parte integral solo cuando est
  I[i] = I[i]+qi*error0[i];

  if (I[i] >= 25)
    I[i] = 25;
  else if (I[i] <= -25)
    I[i] = -25;
  }

  if(fabs(salidaMotor[i])<255){
   D[i]  =  qd*(error0[i]-error1[i]);//Aplicamos parte derivativa solo cuando
  }

  salidaMotor[i] = P[i]+I[i]-D[i];

 error1[i]=error0[i];

    //----------------SalidaMotores-----------------------

  if (salidaMotor[i] > 0.0) {// Mover el motor en avance
     if (salidaMotor[i] >= 255.0)
       salidaMotor[i] = 255.0;

     digitalWrite(entrada2[i], HIGH);
     digitalWrite(entrada1[i], LOW);
```

```
   analogWrite(enable[i], round(fabs(salidaMotor[i])));
  }//end if(avance)

  else    {//Mover el motor en retroceso
    if (salidaMotor[i] <= -255.0)
      salidaMotor[i] = -255.0;

    digitalWrite(entrada2[i], LOW);
    digitalWrite(entrada1[i], HIGH);

   analogWrite(enable[i], round(fabs(salidaMotor[i])));
  }// end else(retroceso)

  }//end for(i)

}//end void control

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

void loop(){

  //Establecemos consignas
   int buffer[25];

  int i=0;

   if (Serial.available() > 0) {// envia datos solamente cuando recibe datos

        for (int j = 0; j < 13; j++) {
          while (Serial.available() == 0)
            ;

           buffer[j] =Serial.read();
          delayMicroseconds(100);

          Serial.print("Buffer ");
          Serial.print(j);
          Serial.print(": ");
          Serial.println(buffer[j]);
        }

        if (buffer[12] == 10) {// Paquete correcto.
          for (int j=0; j < 6; j++) {
            unsigned int temp;
           temp = buffer[j*2];
           temp = temp << 8;
           temp = temp + buffer[j*2+1];
           consignaInt[j] = temp;
          }
```

```
          for (int j=0; j < 6; j++) {
             consigna[j] = 5.0*consignaInt[j]/1024.0;
             }
            i = 0;
                }

    }//end if(Serial)
}
```

Anexo 4.1

Librería TimerOne (.h y .cpp)

```
/*
 *  Interrupt and PWM utilities for 16 bit Timer1 on ATmega168/328
 *  Original code by Jesse Tane forhttp://labs.ideo.comAugust 2008
 *  Modified March 2009 by Jérôme Despatis and Jesse Tane for ATmega328 suppor
 *  Modified June 2009 by Michael Polli and Jesse Tane to fix a bug in setPeri
 *  Modified June 2011 by Lex Talionis to add a function to read the timer
 *  Modified Oct 2011 by Andrew Richards to avoid certain problems:
 *  - Add (long) assignments and casts to TimerOne::read() to ensure calculati
 *  - Ensure 16 bit registers accesses are atomic - run with interrupts disabl
 *  - Remove global enable of interrupts (sei())- could be running within an i
 *  - Disable interrupts whilst TCTN1 == 0.  Datasheet vague on this, but exp
 *    flag gets set whilst TCNT1 == 0, resulting in a phantom interrupt.  Coul
 *    at very short durations
 *  - startBottom() added to start counter at 0 and handle all interrupt enabl
 *  - start() amended to enable interrupts
 *  - restart() amended to point at startBottom()
 * Modiied 7:26 PM Sunday, October 09, 2011 by Lex Talionis
 *  - renamed start() to resume() to reflect it's actual role
 *  - renamed startBottom() to start(). This breaks some old code that expects
 *
 *  This program is free software: you can redistribute it and/or modify
 *       it under the terms of the GNU General Public License as published by
 *       the Free Software Foundation, either version 3 of the License, or
 *       (at your option) any later version.
 *
 *       This program is distributed in the hope that it will be useful,
 *       but WITHOUT ANY WARRANTY; without even the implied warranty of
 *       MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *       GNU General Public License for more details.
 *
 *       You should have received a copy of the GNU General Public License
 *       along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 *   See Google Code projecthttp://code.google.com/p/arduino-timerone/for lat
 */
#ifndef TIMERONE_h
#define TIMERONE_h

#include <avr/io.h>
#include <avr/interrupt.h>

#define   RESOLUTION   65536// Timer1 is 16 bit

class TimerOne
{
  public:

    // properties
    unsigned int pwmPeriod;
```

```
    unsigned char clockSelectBits;
         char oldSREG;                                    // To hold Status Reg

    // methods
    void initialize(long microseconds=1000000);
    void start();
    void stop();
    void restart();
         void resume();
         unsigned long read();
    void pwm(char  pin, int  duty, long microseconds=-1);
    void disablePwm(char pin);
    void attachInterrupt(void  (*isr)(), long microseconds=-1);
    void detachInterrupt();
    void setPeriod(long microseconds);
    void setPwmDuty(char  pin, int duty);
    void (*isrCallback)();
};

extern TimerOne Timer1;
#endif
```

```
/*
 *  Interrupt and PWM utilities for 16 bit Timer1 on ATmega168/328
 *  Original code by Jesse Tane for http://labs.ideo.com August 2008
 *  Modified March 2009 by Jérôme Despatis and Jesse Tane for ATmega328 suppor
 *  Modified June 2009 by Michael Polli and Jesse Tane to fix a bug in setPeri
 *  Modified June 2011 by Lex Talionis to add a function to read the timer
 *  Modified Oct 2011 by Andrew Richards to avoid certain problems:
 *  - Add (long) assignments and casts to TimerOne::read() to ensure calculati
 *  - Ensure 16 bit registers accesses are atomic - run with interrupts disabl
 *  - Remove global enable of interrupts (sei())- could be running within an i
 *  - Disable interrupts whilst TCTN1 == 0.  Datasheet vague on this, but exp
 *    flag gets set whilst TCNT1 == 0, resulting in a phantom interrupt.  Coul
 *    at very short durations
 *  - startBottom() added to start counter at 0 and handle all interrupt enabl
 *  - start() amended to enable interrupts
 *  - restart() amended to point at startBottom()
 * Modiied 7:26 PM Sunday, October 09, 2011 by Lex Talionis
 *  - renamed start() to resume() to reflect it's actual role
 *  - renamed startBottom() to start(). This breaks some old code that expects
 *
 *  This program is free software: you can redistribute it and/or modify
 *       it under the terms of the GNU General Public License as published by
 *       the Free Software Foundation, either version 3 of the License, or
 *       (at your option) any later version.
 *
 *       This program is distributed in the hope that it will be useful,
 *       but WITHOUT ANY WARRANTY; without even the implied warranty of
 *       MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *       GNU General Public License for more details.
 *
 *       You should have received a copy of the GNU General Public License
 *       along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 *   See Google Code project http://code.google.com/p/arduino-timerone/ for lat
 */
#ifndef TIMERONE_cpp
#define TIMERONE_cpp

#include "TimerOne.h"

TimerOne               Timer1;// preinstatiate

ISR(TIMER1_OVF_vect)          // interrupt service routine that wraps a user d
{
  Timer1.isrCallback();
}


void TimerOne::initialize(long microseconds)
```

```
{
          TCCR1A      =      0;// clear control register A
      TCCR1B    =    _BV(WGM13);// set mode 8: phase and frequency correct pwm,
  setPeriod(microseconds);
}


void TimerOne::setPeriod(long microseconds)              // AR modified for at
{

  long    cycles      =      (F_CPU      /      2000000)      *      microsecor
  if(cycles  <  RESOLUTION)                          clockSelectBits  =  _BV(CS1
  else if((cycles   >>=   3)   <   RESOLUTION)   clockSelectBits   =   _BV(CS1
  else if((cycles >>= 3) < RESOLUTION) clockSelectBits = _BV(CS11) | _BV(CS1
  else if((cycles   >>=   2)   <   RESOLUTION)   clockSelectBits   =   _BV(CS1
  else if((cycles >>= 2) < RESOLUTION) clockSelectBits = _BV(CS12) | _BV(CS1
  else        cycles = RESOLUTION - 1, clockSelectBits = _BV(CS12) | _BV(CS1

  oldSREG = SREG;
  cli();                                                 // Disable interrupt
              ICR1        =        pwmPeriod        =        cycles;// ICR1
  SREG = oldSREG;

  TCCR1B &= ~(_BV(CS10) | _BV(CS11) | _BV(CS12));
                    TCCR1B             |=             clockSelectBits;// reset
}

void TimerOne::setPwmDuty(char  pin,int duty)
{
  unsigned long dutyCycle = pwmPeriod;

  dutyCycle *= duty;
  dutyCycle >>= 10;

  oldSREG = SREG;
  cli();
  if(pin == 1 || pin == 9)       OCR1A = dutyCycle;
  else if(pin == 2 || pin == 10) OCR1B = dutyCycle;
  SREG = oldSREG;
}

void TimerOne::pwm(char  pin,int  duty,long  microseconds)// expects duty cy
{
  if(microseconds > 0) setPeriod(microseconds);
  if(pin == 1 || pin == 9) {
                           DDRB       |=       _BV(PORTB1);// sets data direct
                          TCCR1A       |=       _BV(COM1A1);// activates the ou
  }
  else if(pin == 2 || pin == 10) {
```

```
    DDRB |= _BV(PORTB2);
    TCCR1A |= _BV(COM1B1);
  }
  setPwmDuty(pin, duty);
  resume();                     // Lex - make sure the clock is running.  We
                                // and the first one is in the middle
}

void TimerOne::disablePwm(char pin)
{
  if(pin == 1 || pin == 9)        TCCR1A &= ~_BV(COM1A1);// clear the bit th
  else if(pin == 2 || pin == 10)  TCCR1A &= ~_BV(COM1B1);// clear the bit th
}

void TimerOne::attachInterrupt(void  (*isr)(),long microseconds)
{
  if(microseconds > 0) setPeriod(microseconds);
                   isrCallback            =            isr;// register the use
                  TIMSK1           =           _BV(TOIE1);// sets the timer c
        // might be running with interrupts disabled (eg inside an ISR), so c
//  sei();
  resume();
}

void TimerOne::detachInterrupt()
{
                 TIMSK1          &=          ~_BV(TOIE1);// clears the time

}

void TimerOne::resume()                         // AR suggested
{
  TCCR1B |= clockSelectBits;
}

void TimerOne::restart()        // Depricated - Public interface to start at
{
        start();
}

void TimerOne::start()   // AR addition, renamed by Lex to reflect it's actual
{
  unsigned int tcnt1;

     TIMSK1   &=   ~_BV(TOIE1);//  AR  added
   GTCCR  |=  _BV(PSRSYNC);              // AR added - reset prescaler (NB: sh

  oldSREG = SREG;                                // AR - save status register
  cli();                                         // AR - Disable interrupts
```

```
        TCNT1      =      0;
      SREG    =    oldSREG;              // AR - Restore status register
        resume();
  do {   // Nothing -- wait until timer moved on from zero - otherwise get a p
        oldSREG = SREG;
        cli();
        tcnt1 = TCNT1;
        SREG = oldSREG;
   }while  (tcnt1==0);

//        TIFR1     =     0xff;          // AR - Clear interrupt flags
//  TIMSK1 = _BV(TOIE1);              // sets the timer overflow interrupt ena
}

void TimerOne::stop()
{
    TCCR1B   &=   ~(_BV(CS10)   |   _BV(CS11)   |   _BV(CS12));// clears all clock
}

unsigned long TimerOne::read()          //returns the value of the timer in m
{                                                                    //re
        unsigned long tmp;                              // AR amended to hold
        unsigned int tcnt1;                             // AR added

        oldSREG= SREG;
        cli();
        tmp=TCNT1;
        SREG = oldSREG;

        char scale=0;
        switch (clockSelectBits)
        {
        case 1:// no prescalse
                scale=0;
                break;
        case 2:// x8 prescale
                scale=3;
                break;
        case 3:// x64
                scale=6;
                break;
        case 4:// x256
                scale=8;
                break;
        case 5:// x1024
                scale=10;
                break;
        }
```

```
        do {    // Nothing -- max delay here is ~1023 cycles.  AR modified
                oldSREG = SREG;
                cli();
                tcnt1 = TCNT1;
                SREG = oldSREG;
        } while  (tcnt1==tmp);//if the timer has not ticked yet

        //if we are counting down add the top value to how far we have counte
        tmp = (  (tcnt1>tmp) ? (tmp) : (long)(ICR1-tcnt1)+(long)ICR1   );
        return ((tmp*1000L)/(F_CPU /1000L))<<scale;
}

#endif
```

Anexo 5

Código MATLAB

```
clear all;
delete(instrfind({'Port'},{'COM9'}));
arduserial = serial('COM9','BaudRate',9600);
warning('off','MATLAB:serial:fscanf:unsuccessfulRead');
fopen(arduserial);

%A=coordenadas del terminal superior de cada pata desde
%plataforma superior (matriz 3*6)

%B=coordenadas del terminal inferior de cada pata desde
%plataforma inferior (matriz 3*6)

%pose=posicion final del centro de la plataforma superior
%respecto a la inferior. Traslacion y rotacion (3 coordenadas y 3 angulos)

A=[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]; %valor fijo
B=[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]; %valor fijo
pose=[0; 0; 16; 0; 0; 0]; %valor variable

consigna = ik(A, B, pose);
for i = 1:6
    consigna(i) = consigna(i)-17.5;
    consigna(i) = 1024*consigna(i)/5;
end

temp = 0;
for i = 1:6
    if (consigna(i)<0 || consigna(i)>1023)
        temp++;
    end
end

if temp == 0
    pause(1);
    for i = 1:6
        fwrite(arduserial, floor(consigna(i)/256),'uint8');
        fwrite(arduserial, bitand(consigna(i),255),'uint8');
    end
    fwrite(arduserial, 10,'uint8');
end

fclose(arduserial);
delete(arduserial);
clear arduserial;
```

```
%A=coordenadas del terminal superior de cada pata desde plataforma superior (matriz 3*6)
%B=coordenadas del terminal inferior de cada pata desde plataforma inferior (matriz 3*6)
%pose=posicion final del centro de la plataforma superior respecto a la inferior. Traslacion y rotacion (3 coordenadas y 3 angulos)


function [L] = ik(A, B, pose)   % A y B en coordenadas homogeneas
    A = A(1:3,:);
    B = B(1:3,:);
    R = pose(4:6,:);
    D = pose(1:3,:);
    Rrad = (R.*pi)./180; % De grados a radianes
    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];

    L = sqrt(sum( (RR * A + D(:,ones(6,1)) - B).^2 ));
end
```