

**MÁSTER DE AUTOMÁTICA E
INFORMÁTICA INDUSTRIAL**

**Instituto Universitario de Automática e
Informática Industrial**

Universitat Politècnica de Valencia



TESINA FINAL DE MÁSTER

**IMPLEMENTACIÓN BASADA EN EL
MIDDLEWARE OROCOS DE
CONTROLADORES DINÁMICOS
PARA UN ROBOT PARALELO**

Autor: José Ignacio Casalilla Morenas

Dirigida por: Dra. Dña. Marina Vallés Miquel

Dr. D. Ángel Valera Fernández

Valencia, Julio de 2012

Agradecimientos

Mis agradecimientos, en primer lugar, a mis directores de Tesina, Ángel y Marina, por haberme dado la oportunidad de realizar este proyecto con ellos, formando parte de sus proyectos de investigación. Gracias a los dos por vuestra dedicación y entusiasmo tanto en el campo de la investigación como en el de la docencia, y porque junto a vosotros he crecido no sólo como investigador, sino también como persona. Siempre seréis mi modelo a seguir.

Gracias también al personal del Departamento de Ingeniería Mecánica y Materiales, en especial al Profesor Vicente Mata por su interés mostrado en el proyecto y por toda la ayuda recibida.

No puedo olvidarme tampoco de mis padres, mi hermana y mis tíos, que siempre habéis estado y estaréis ahí cuando os necesite. Por fingir como nadie que entendíais algo cuando os contaba algo referente al *famoso* robot paralelo, gracias. Os quiero.

Finalmente, no me queda más que agradecer a uno de los pilares de mi vida, Irene, que sin tu apoyo incondicional en estos casi cuatro años nada hubiera sido igual. Gracias por tus sabios consejos cuando lo veía todo cuesta arriba.

A todos, gracias.

Índice

1. Introducción, estado del arte, objetivos y justificación	1
2. Desarrollo teórico	7
2.1. Introducción a la robótica. Robot Paralelo	7
2.1.1. El Robot paralelo	7
2.1.2. Estructuras y componentes de un robot	9
2.1.3. Cinemática en robots	11
Modelo cinemático directo	11
Modelo cinemático inverso	11
2.1.4. Control Dinámico de robots	12
2.2. Middleware de tiempo Real	12
2.2.1. Desarrollo de software basado en componentes	14
2.3. Orocos	14
2.3.1. Historia y proyecto	14
2.3.2. Kinematics and Dynamics Library (KDL)	16
2.3.3. Real Time Toolkit (RTT)	17
2.3.4. Bayesian Filtering Library (BFL)	18
2.3.5. Orocos Component Library (OCL)	19
2.3.6. OrocosToolChain	19
3. Desarrollo práctico	21
3.1. Plataforma de desarrollo hardware	21
3.1.1. Equipo industrial	21
3.1.2. Advantech PCI 1720	22
3.1.3. Advantech PCL 833	23
3.1.4. ATI Force Sensor	24
3.2. Plataforma de desarrollo software	25
3.2.1. Sistema Operativo	25
3.2.2. Lectura de los valores del encóder	26
3.2.3. Envío de la acción de control a los actuadores	27
3.2.4. Lectura de los valores de fuerza	28
3.3. Componentes en Orocos	30
3.3.1. Partes de un componente	30
3.3.2. Estructura interna de un componente	31
3.3.3. Tipos y uso de puertos	32
3.3.4. Interconexión entre componentes	35
3.4. Modelos cinemáticos del robot 3PRS	37
3.4.1. Cinemática directa	37
3.4.2. Cinemática inversa	39
3.5. Modelo dinámico del robot 3PRS	40
3.6. Algoritmos de control	41
3.6.1. Controladores basados en la pasividad	41
Control PD+G	42
Control PID	43

Control de Paden-Panja	44
3.6.2. Controladores por dinámica inversa	44
Control punto a punto	46
Control por trayectoria	47
3.6.3. Controlador adaptativo	47
3.7. Configuraciones modulares usando Orocos	51
3.7.1. Solución inicial (un sólo módulo)	52
3.7.2. Solución modular	54
3.7.3. Solución modular con supervisor	56
3.7.4. Solución modular con generador de trayectorias	59
3.7.5. Inclusión del módulo de fuerza	61
3.7.6. División en submódulos del módulo de control	62
4. Resultados experimentales	65
4.1. Seguimiento de trayectoria de posición	65
4.1.1. Controlador PID	65
4.1.2. Controlador Paden-Panja	66
4.1.3. Comparativa PID vs Paden-Panja	68
4.1.4. Control Adaptativo	69
4.2. Seguimiento de trayectoria de fuerza	70
5. Conclusiones y trabajos futuros	73
Referencias	78
A. Instalación de Orocos ToolChain	79
B. Creación y funcionamiento de componentes	81

Índice de figuras

1.	Fases principales de una tecnología	3
2.	Cohesión y coupling de módulos	4
3.	Formulación de un CBSS	4
4.	Estimación de coste para modelos de CBSD	5
5.	Robot Paralelo 3PRS utilizado	9
6.	Estructura general de un robot	10
7.	Middleware	13
8.	Brazo robot en serie	16
9.	RTT Services	17
10.	RTT Stack	18
11.	Estimación mediante BFL	18
12.	Comunicación entre varios componentes	19
13.	Equipo industrial	21
14.	PCI 1720	22
15.	PCL 833	23
16.	Ati Force Sensor	24
17.	Uname -a	26
18.	Tabla DeviceNet	29
19.	Partes de un componente	31
20.	Estructura interna	32
21.	Puerto normal	34
22.	Puerto evento	34
23.	Puerto de evento asociado a función	35
24.	Conexiones de módulos	36
25.	Modelo CAD del Robot paralelo 3PRS	37
26.	Localización sistemas de coordenadas	38
27.	Esquema de control PD+G	43
28.	Esquema de control PID	43
29.	Esquema de control Paden	44
30.	Esquema de control punto a punto	46
31.	Esquema de control por trayectoria	47
32.	Esquema de control adaptativo	51
33.	Solución inicial.	53
34.	Solución modular.	54
35.	Solución modular con supervisor.	57
36.	Tiempo de ejecución control	58
37.	Tiempo de ejecución total	59
38.	Esquema con generador de referencias	60
39.	Esquema con el módulo de fuerza	62
40.	Esquema con el módulo de fuerza	63
41.	Esquema con el módulo de fuerza	63
42.	Articulación 1 Control PID	65
43.	Articulación 2 Control PID	66

44.	Articulación 3 Control PID	66
45.	Articulación 1 Control Paden-Panja	67
46.	Articulación 2 Control Paden-Panja	67
47.	Articulación 3 Control Paden-Panja	68
48.	Comparativa control PID vs Paden-Panja	69
49.	Error Paden y Adaptativo	70
50.	Errores cuadráticos adaptativo	70
51.	Robot 3PRS con sensor de fuerza	71
52.	Seguimiento trayectoria de fuerza	71
53.	Calibración dinámica de la plataforma	75
54.	Sistema de rehabilitación	75
55.	Instalación Orocós	80
56.	Estructura ficheros componente	81
57.	Cmake	82
58.	Cmake	82
59.	Cmake	83

1. Introducción, estado del arte, objetivos y justificación

Como se sabe, los robots industriales están presentes cada vez más en diversos procesos de fabricación actuales, habiendo una gran cantidad de robots que son programados para que realicen diferentes tareas repetitivas y de la forma más eficaz posible. En el mundo (y época) en el que vivimos, donde la dura competencia, la calidad del producto, los plazos de entrega y, sobre todo, el precio del producto final marcan el éxito o fracaso de cualquier multinacional o pequeña y mediana empresa, es muy importante optimizar los procesos de fabricación. En este último punto es donde está muy presente el campo de la robótica, siendo los robots industriales los que tienen un papel muy importante en el logro de un producto de alta calidad a bajo precio.

Debido a la dura competencia, los continuos cambios en elementos hardware y software y, sobre todo, la complejidad de las tareas sobre las que los robots han de interactuar, es necesario un nuevo enfoque en la programación de los mismos. Por ejemplo, en la actualidad muchas aplicaciones de robots industriales, integran visión artificial (sector alimentario), reconocimiento de formas, planificación de trayectorias. . . Todos estos conceptos son de un alto nivel, y provocan que la fusión de todos los elementos que los hacen posible en una aplicación que los gobierne adecuadamente para que junto con el robot se pueda realizar la tarea objetivo de forma adecuada, sea complicado. Supongamos, tal y como pasa en la realidad, que cada elemento citado anteriormente puede ser de una marca determinada, o bien está programado en un determinado lenguaje, las unidades de medida corresponden a otra región, etc. Sin duda, esto es una dificultad extra a la hora de realizar una aplicación que gobierne al robot en su tarea.

La aplicación de técnicas avanzadas de control permiten mejorar sustancialmente las prestaciones de sistemas mecánicos complejos sin alterar significativamente su estructura mecánica. Un buen ejemplo de ello es la incorporación del control dinámico a robots industriales que, hasta hace pocos años, tenían un sistema de control puramente cinemático. Sin embargo, hay que tener en cuenta que, debido a las características dinámicas de este tipo de técnicas, los controladores resultantes derivan en aplicaciones de tiempo real que requieren tener un cierto cuidado en su implementación para que los resultados obtenidos sean satisfactorios, ya que el coste computacional se incrementa significativamente.

En el caso de los sistemas mecánicos complejos, su implementación puede llevar a la aparición de múltiples bucles de control, con distintas variables a medir o estimar (según la magnitud sea medible o no a través de sensores), tales como la velocidad o aceleración a partir de la posición, apareciendo distintos requerimientos tanto temporales como de recursos de computación y que, además, necesiten una cierta sincronización a la hora de compartir información entre ellos.

Por lo tanto, en la presente tesina se pretende evaluar la mejora en prestaciones que puede suponer la implementación en tiempo real de controladores

para un robot paralelo de 3 grados de libertad, previamente diseñados, usando el entorno de desarrollo Orocos.

Por otro lado, puesto que la tesina aborda la implementación de software basada en componentes, a continuación se va a realizar una breve revisión del estado del arte en relación a los distintos artículos, autores e instituciones que han abordado algún tipo de proyecto de similares características al expuesto en esta tesina. Concretamente, se centrará básicamente en aspectos relacionados con el *desarrollo de software basado en componentes (CBSD)*, tratando de obtener una visión de los orígenes, situación actual y, finalmente, cuáles son las principales tendencias para el futuro.

Como paso previo a revisar distintos artículos relacionados con el desarrollo de software basado en componentes, es necesario comentar los diferentes puntos de vista sobre cómo definir el CBSD.

En [1] se comenta que el CBSD *es una alternativa de diseño que favorece la reutilización de elementos software y facilita el desarrollo de sistemas a partir de elementos preexistentes, siendo un componente software una unidad de composición con interfaces bien definidas y un contexto de uso explícito*. Como complemento a esta definición, en [19] se expone que el CBSD *se basa en la descomposición de un software en componentes lógicos o funcionales con interfaces bien definidas, con el fin de facilitar el tiempo de desarrollo, y mejorar el mantenimiento del sistema*.

Centrándonos en el marco histórico sobre cuándo fue el comienzo del desarrollo del software basado en componentes, en [3] se tienen algunas de las respuestas. Según los autores, la arquitectura del software (así como cualquier tecnología), tiene distintas fase por las que se va desarrollando o *madurando* la tecnología en cuestión (figura 1). Estas fases son:

1. Investigación básica, en la exponen las ideas básicas y el problema.
2. Formulación de conceptos, donde se asientan las primeras ideas y se resuelven pequeños problemas
3. Desarrollo y expansión.
4. Mejora interna y exploración, de forma que los propios creadores extienden las propuestas iniciales hacia otros campos, usándolos en problemas reales.
5. Mejora externa y exploración, de forma que gente que era totalmente ajena al proyecto, se da cuenta de que es una tecnología útil y aplicable, por lo que comienza a hacer sus propios desarrollos.
6. Divulgación, que consiste en que la tecnología es conocida y aceptada por buena parte de la sociedad, creando distintas versiones y mejoras.

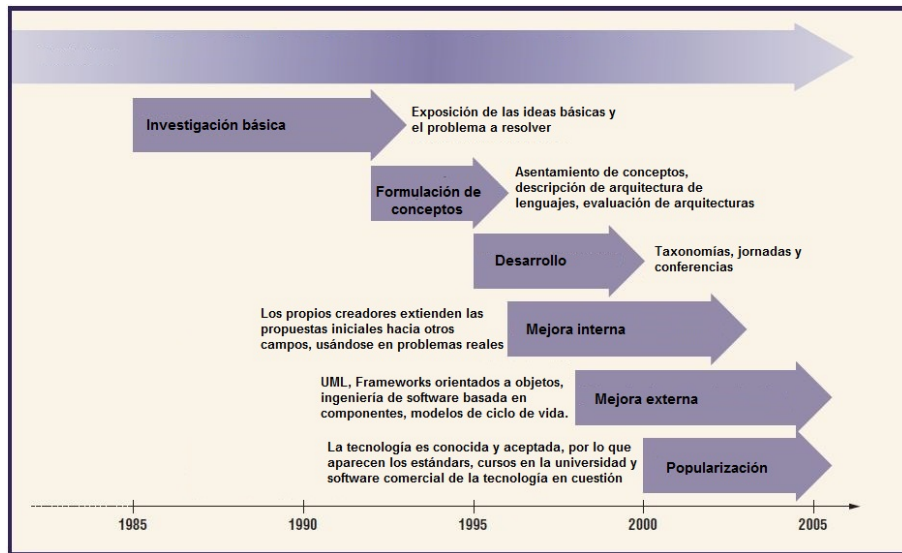


Figura 1: Fases principales de una tecnología

Como se puede ver en la figura anterior, el desarrollo de software basado en módulos ha sido en unas de las últimas fases y, relativamente, reciente. Según el autor, el CBD ha comenzado a tener un auge importante cuando se ha llegado a conocer del todo el software orientado a objetos, ya que un componente puede entenderse como un objeto, destacando su interoperabilidad y reusabilidad.

Puesto que en la última década ha habido un interés creciente en los sistemas software basados en componentes, lejos de la pura definición, aplicaciones, ventajas o inconvenientes, un aspecto importante a tratar es la optimización de los distintos módulos.

En [19] se aborda el tema de la optimización, impulsado según los autores porque *hay muy pocos trabajos de investigación que separen los aspectos teóricos de la optimización en la creación de componentes*. Los autores comentan que una de sus propuestas es que internamente un módulo esté fuertemente unido o *cohesionado*, pero que cada uno de los módulos sea lo más independiente posible de los otros (figura 2).

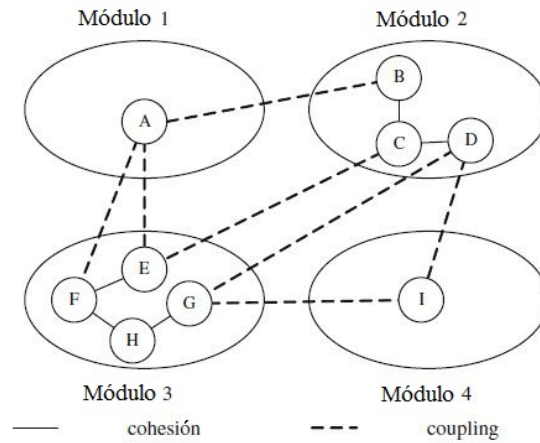


Figura 2: Cohesión y coupling de módulos

Por ello, el principal objetivo para conseguir módulos bien estructurados es maximizar la *cohesion* y minimizar el *coupling*, llegando a escribir en su trabajo de investigación (Abreu & Goulão, 2001) una fórmula (fórmula 1) referente a la *intra-modular coupling density (ICD)*, con el fin de medir la relación entre *cohesion* y *coupling* de los módulos de un diseño propuesto.

$$ICD = \frac{CI_{IN}}{CI_{IN} + CI_{OUT}} \quad (1)$$

donde CI_{IN} es el número de interacciones de clases con módulos, mientras que CI_{OUT} es el número de interacciones con clases de distintos módulos.

Otro aspecto importante que nos presentan los autores de [19] es que los módulos software tienen que ser identificados lo primero de todo, puesto que cada módulo contiene, al menos, un componente software (figura 3).

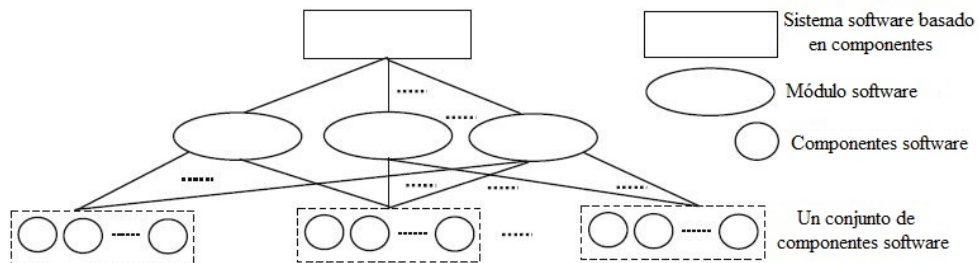


Figura 3: Formulación de un CBSS

Finalmente, un conjunto de módulos software formarán el sistema software basado en componentes (CBSS).

Por otro lado, y tal como dicen en [22], la estimación del coste de desarrollo del software, es un aspecto realmente importante dentro de la disciplina de la ingeniería del software. Pese a que hay un gran número de estudios acerca de la planificación del coste de desarrollo del software, el creciente aumento de CBSD (según los autores, el 53% de 118 empresas participantes en el estudio usan CBSD, siendo una tendencia ascendente) ha supuesto que se hayan tenido que realizar estudios de coste, pero específicos referentes a software basado en componentes. Comentan los autores que *en esta planificación hay que tener presente el particionado del sistema, la modularización, la abstracción de datos y la información oculta, siendo cada componente similar a una caja negra*.

De la revisión que han realizado en [22], han englobado en 3 grupos las propuestas para la predicción del coste y planificación de un CBSD (figura 4).

- **BASADAS EN MODELOS ALGORÍTMICOS.** Son las más comunes, y relacionan aspectos como la dependencia entre distintos modelos, costes de licencias, coste estimado por componente, así como modelos probabilísticos.
- **BASADAS EN MODELOS INTEGRALES O COMPUESTOS.** La más conocida es COCOTS y se basa en la combinación de varias técnicas de modelado, usando después la más apropiada para la estimación del coste.
- **OTRAS MEJORAS.** Son técnicas que no pueden ser clasificadas en ningún grupo, como la técnica basada en vectores (Yakimovich, 1999).

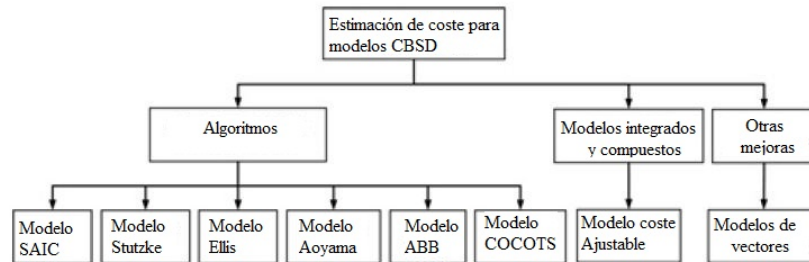


Figura 4: Estimación de coste para modelos de CBSD

Como conclusión a la extensa y minuciosa revisión hecha en [22], los autores opinan que la estimación del coste de un CBSD es un área muy poco avanzada, y en la que se debería profundizar más, no sólo en el mero coste de creación, sino también en la eficacia y productividad.

Otro aspecto referente a un CBSD, que no se ha comentado, son los *Frameworks* (o espacios de trabajo), los cuales facilitan enormemente el trabajo cuando se está desarrollando un software basado en componentes.

En [1] destacan algunos referentes al contexto de la robótica, siendo los más importantes los que se comentan a continuación:

- ORCA. Es de código abierto, y viene con un repositorio de componentes ya implementados.
- SmartSoft. Destaca por el enfoque a patrones de comunicación.
- GenoM. En este paquete se implementan algoritmos y dispositivos hardware típicos de la robótica
- Orocos. Que será comentado en detalle posteriormente.

Los autores de [1] dan a entender que una problemática de los frameworks es que son dependientes de la plataforma, puesto que se utilizan librerías dinámicas en su implementación, hecho con el que no estoy de acuerdo, puesto que con Orocos, por ejemplo, se puede implementar un componente desde una arquitectura determinada, especificándole en el momento de la compilación para qué arquitectura va a ser y las librerías que se quieren utilizar.

Respecto a las tendencias y la evolución, todos los artículos están de acuerdo en que el uso de componentes va en claro ascenso año tras año, comentando en [1] que ya hay *marcos conceptuales y metodológicos para la definición y uso de lenguajes para el modelado específico de frameworks* (FSML) (Antkiewicz and Stephan, 2009).

Por tanto, la justificación de por qué se ha decidido trabajar en esta tesina de Máster sobre la implementación de controladores dinámicos usando el entorno Orocos es, sobre todo, el auge de la robótica industrial y la programación en tiempo real para procesos complejos.

También, como se verá en los apartados posteriores, Orocos tiene como objeto permitir desarrollar controladores para robots y máquinas en un entorno modular, de software libre y de propósito general. La programación se realiza en lenguaje C++ usando las librerías Real-Time Toolkit, Kinematics and Dynamics, Bayesian Filtering y Orocos Component. La ventaja del software libre es que todo el código fuente es accesible, por lo que, para adaptar nuevas funcionalidades al mismo, no existen grandes restricciones pudiéndose modificar los fuentes.

Finalmente, cabe destacar la principal característica de la programación usando la librería Orocos, que es su programación modular, en la que cada módulo es totalmente independiente a los demás, proporcionando ventajas tales como la reusabilidad del código, ejecución de forma distribuida o configuración y reconfiguración en tiempo de ejecución, entre otras muchas ventajas.

2. Desarrollo teórico

En esta sección se van a describir de forma teórica distintos aspectos relacionados con el presente trabajo. En primer lugar se describirá el robot de configuración paralela, así como sus principales características, estructuras y componentes del mismo.

Posteriormente se describirá en qué consiste un middleware de tiempo real (y el desarrollo de software basado en componentes), para terminar explicando de forma más detallada el middleware usado en este proyecto, Orocos.

2.1. Introducción a la robótica. Robot Paralelo

Es complicado definir formalmente el término de robot industrial, puesto que, dependiendo cuál sea la situación (por ejemplo, entre el mercado japonés y el euro-americano), puede haber distintas diferencias en el mismo. Una de las principales discrepancias es que mientras que para los japoneses un robot industrial es cualquier dispositivo mecánico dotado de articulaciones móviles destinado a la manipulación, el mercado occidental es más restrictivo exigiendo una mayor complejidad, sobre todo en lo relativo al control.

La definición del Robotics Institute of América (RIA), define al robot industrial como *un manipulador multifuncional reprogramable, capaz de mover materias, piezas, herramientas, o dispositivos especiales, según trayectorias variables, programadas para realizar tareas diversas.*

Sin embargo, la Asociación Japonesa de Robótica Industrial (JIRA), con una definición más genérica que la americana, lo define como *dispositivos capaces de moverse de modo flexible análogo al que poseen los organismos vivos, con o sin funciones intelectuales, permitiendo operaciones en respuesta a las órdenes humanas.*

Por último, la Federación Internacional de Robótica (IFR) distingue entre robot industrial de manipulación y otros robots: *Por robot industrial de manipulación se entiende una máquina de manipulación automática, reprogramable y multifuncional con tres o más ejes que pueden posicionar y orientar materias, piezas, herramientas o dispositivos especiales para la ejecución de trabajos diversos en las diferentes etapas de la producción industrial, ya sea en una posición fija o en movimiento.*

Por tanto, pese a todas estas definiciones, es imposible cubrir todos los tipos de robots o manipuladores, ya que, además de que año tras año aparecen nuevos tipos de robots debido a cada configuración u objetivo, cada uno de estos tiene una acepción diferente.

2.1.1. El Robot paralelo

Dado que el objetivo de este proyecto está relacionado con la implementación de controladores para un robot paralelo, a continuación se explicarán los conceptos más importantes y generales de un robot de este tipo.

Su distinción *paralela* es porque el extremo final del miembro de robot está conectado a su base por una serie de (por lo general tres o seis) miembros independientes, con la particularidad de que trabajan en paralelo. Por ello, *paralelo* se utiliza aquí en el sentido topológico, en lugar de la geometría, puesto que los miembros pueden trabajar juntos, y no tienen por qué estar alineados en forma de líneas paralelas.

Un robot paralelo consiste en una plataforma móvil que se conecta a una base fija por medio de varias cadenas cinemáticas. Estos robots tienen un elemento final conectado a la plataforma móvil. Los robots paralelos tienen ventajas respecto a los robots serie que consisten, esencialmente, en que la carga se reparte entre varias articulaciones que conectan la plataforma móvil a la base. Así, este tipo de robots tienen una serie de ventajas que los hacen muy útiles para diversas tareas:

- Presentan alta rigidez y robustez.
- Alta capacidad de transportar carga.
- Alta velocidad y precisión.
- La cinemática inversa es relativamente sencilla de calcular, lo que permite la simulación de modelos, para posteriormente aplicarlo al modelo real.

Sin embargo, como principales inconvenientes están los siguientes:

- Tienen espacios de trabajo reducidos.
- Problemas de singularidad.
- Problemas a la hora de planificar su control.

La elección de la arquitectura y movimiento del robot paralelo han venido determinadas por la necesidad de desarrollar un robot de bajo coste capaz de generar rotación angular en dos ejes (roll y pitch) y elevación como movimiento lineal. Finalmente en este proyecto se optó por la arquitectura 3-PRS (prismática-revolución-esférica), siendo una de las ventajas de la arquitectura PRS que los actuadores se localizan en la base fija, mientras que en la arquitectura 3-RPS, los actuadores se mueven junto con las articulaciones de revolución.

En la figura 5 se puede ver el robot paralelo utilizado. Se puede ver como tiene tres miembros que conectan la plataforma móvil con la base. Cada miembro consiste en:

1. Un motor, que mueve un husillo a bola.
2. Un deslizador.
3. Un vástago de conexión.

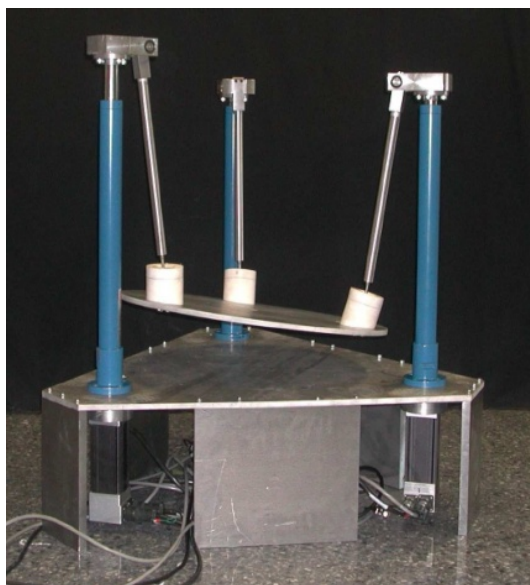


Figura 5: Robot Paralelo 3PRS utilizado

2.1.2. Estructuras y componentes de un robot

En esta sección, se presentará la estructura general de un robot, así como sus componentes básicos, de forma que en apartados posteriores se entienda la utilidad de cada uno de ellos.

De forma general, la estructura de un robot se puede dividir en tres unidades funcionales que permiten al usuario influir en el entorno de trabajo del robot:

- **UNIDAD DE PROGRAMACIÓN.** La unidad de programación se utiliza para programar y operar con el robot. A través de la unidad de programación, los resultados de los programas ejecutados y los comandos, pueden ser analizados. Normalmente suele consistir en un teclado industrial, una pantalla y un control remoto.
- **SISTEMA DE CONTROL.** El sistema de control recibe los comandos y las instrucciones de la unidad de programación. Después de interpretar los comandos e instrucciones, su tarea es generar las acciones de control adecuadas. Las acciones de control deben manejar correctamente el sistema mecánico, dependiendo de su estado, con el objetivo de minimizar el error. Además, el sistema de control informará al usuario sobre el resultado de las acciones de control a través de la unidad de programación. El sistema de control está formado típicamente por un equipo industrial, que incluye tarjetas de adquisición y conversión para la comunicación con el robot mecánico, herramientas y otros dispositivos.
- **MECÁNICA DEL ROBOT.** La mecánica del robot convierte las acciones de

control en movimientos del robot, con el fin de interactuar con el medio y realizar una actividad determinada. Además, los componentes mecánicos informan periódicamente al sistema de control sobre el estado del robot. La mecánica del robot se compone de varios elementos individuales que están conectados por articulaciones, produciéndose un movimiento de éstas mediante motores eléctricos, hidráulicos o actuadores neumáticos. Por otro lado, los sensores, tales como los encóders, permiten conocer en cada instante la posición de una articulación.

En la figura 6 se puede apreciar la estructura general del robot y las diferentes interacciones entre los diversos componentes.

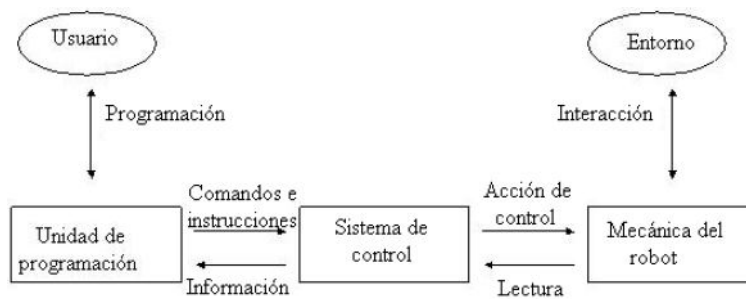


Figura 6: Estructura general de un robot

Como se ha comentado anteriormente, el sistema de control consiste en un equipo industrial y los elementos adicionales siguientes:

- Convertidor D/A para proporcionar las acciones de control.
- Transformadores para las acciones de control.
- Tarjetas de adquisición de datos A/D para obtener información de los sensores.
- Tarjetas de comunicación (serie, paralelo, Ethernet).

Las principales funciones del mismo son las siguientes:

- Ser capaz de controlar la interfaz de comunicación con el usuario, así como interpretar comandos enviados por el usuario, tales como activaciones de las salidas digitales o analógicas, o alguna situación excepcional como una parada de emergencia. Los comandos correspondientes son interpretados y ejecutados de inmediato. El usuario, además puede introducir los programas en la unidad de programación, que serán interpretados y ejecutados por el sistema de control.
- Generar y controlar los movimientos del robot mediante la creación de una señal de referencia que dirige los actuadores del robot. Durante todo el

proceso, la referencia se compara con la posición obtenida por los sensores. El sistema de control debe intentar reducir la diferencia entre la referencia y la salida (es decir, el error) por la aplicación de un algoritmo de control.

Finalmente, el movimiento de las articulaciones del robot es producido por los actuadores que actúan de acuerdo a las acciones de control que genera el sistema de control. A pesar de que hay varios tipos de actuadores, en función a su fuente de energía, en este caso se han utilizado actuadores eléctricos.

2.1.3. Cinemática en robots

La cinemática es la ciencia del movimiento que trata a éste sin importarle las fuerzas que lo causan. Dentro de la cinemática se estudia la posición, la velocidad, aceleración y todas las derivadas de las variables de posición de mayor orden con respecto al tiempo o cualquier otra variable. El estudio de la cinemática de los manipuladores se refiere a todas las propiedades geométricas basadas en el tiempo del movimiento.

Los robots consisten en un conjunto de eslabones conectados mediante articulaciones que permiten el movimiento relativo entre los eslabones vecinos. El número de grados de libertad que un robot posee es el número de variables de posición independientes que deberían ser especificadas para localizar todas las partes del mecanismo. En el caso de los robots industriales el número de grados de libertad suele equivaler al número de articulaciones siempre y cuando cada articulación tenga un solo grado de libertad.

Modelo cinemático directo El modelo cinemático directo es el problema geométrico que calcula la posición y orientación del efector final del robot. Dados una serie de ángulos entre las articulaciones, el problema cinemático directo calcula la posición y orientación del marco de referencia del efector final con respecto al marco de la base.

Modelo cinemático inverso Dada la posición y orientación del efector final del robot, el problema cinemático inverso consiste en calcular todos los posibles conjuntos de ángulos entre las articulaciones que podrían usarse para obtener la posición y orientación deseada.

El problema cinemático inverso es más complicado que la cinemática directa ya que las ecuaciones no son lineales, sus soluciones no son siempre fáciles o incluso posibles en una forma cerrada. También surge la existencia de una o de diversas soluciones. La existencia o no de la solución lo define el espacio de trabajo de un robot dado. La ausencia de una solución significa que el robot no puede alcanzar la posición y orientación deseada porque se encuentra fuera del espacio de trabajo del robot o fuera de los rangos permisibles de cada una de sus articulaciones.

2.1.4. Control Dinámico de robots

La dinámica se ocupa de la relación entre las fuerzas que actúan sobre un cuerpo y el movimiento que en él se origina. Por lo tanto, el modelo dinámico de un robot tiene como objetivo conocer la relación entre el movimiento del robot y las fuerzas implicadas en el mismo.

Esta relación se obtiene mediante el denominado modelo dinámico, que relaciona matemáticamente:

- La localización del robot definida por las variables de las articulaciones o por las coordenadas de localización de su extremo, así como la velocidad y aceleración.
- Las fuerzas de par aplicadas en las articulaciones (o en el extremo del robot).
- Los parámetros dimensionales del robot, como longitud, masa e inercias de sus elementos.

La obtención del modelo para sistemas con pocos grados de libertad (1 o 2) no es demasiado complicado, aunque esta tarea aumenta de dificultad a medida que ese número se incrementa.

El problema de la obtención del modelo dinámico de un robot es, por lo tanto, uno de los aspectos más complejos de la robótica, lo que ha llevado a ser obviado en numerosas ocasiones. Sin embargo, el modelo dinámico es imprescindible para conseguir fines tales como:

- Simulación del movimiento del robot.
- Diseño y evaluación de la estructura mecánica del robot.
- Dimensionamiento de los actuadores.
- Diseño y evaluación del control dinámico del robot.

Este último fin es, evidentemente, el más importante de todos, ya que el buen funcionamiento del robot depende de la precisión y velocidad de sus movimientos.

2.2. Middleware de tiempo Real

El Middleware es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, software, redes, hardware y/o sistemas operativos. Éste simplifica el trabajo de los programadores en la compleja tarea de generar las conexiones que son necesarias en los distintos sistemas. De esta forma se provee una solución que mejora la calidad de servicio (QoS), seguridad, envío de mensajes, directorio de servicio, etc.

Funciona como una capa de abstracción de software, que se sitúa entre la capa de aplicación y el sistema operativo. El middleware abstrae de la complejidad y heterogeneidad de las redes de comunicaciones subyacentes, así como de

los sistemas operativos y lenguajes de programación, proporcionando una API para la fácil programación y manejo de aplicaciones distribuidas, entre otras. Dependiendo del problema que se tenga que resolver y de las funciones necesarias, serán útiles diferentes tipo de servicios de middleware. En la figura se puede observar lo expuesto anteriormente.

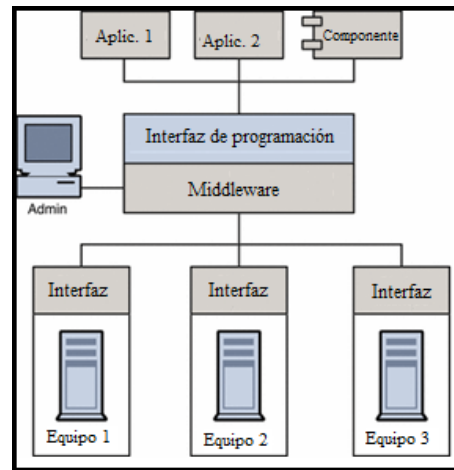


Figura 7: Middleware

Dependiendo de la categoría de integración, existen middlewares orientados a distintos aspectos:

- Orientados a procedimiento o procesos
- Orientados a objetos
- Orientados a mensajes
- Orientados a componentes

En este caso, y tal y como se comentará en los apartados posteriores, el middleware (Orocós) usado en este trabajo es orientado a componentes, siendo una de sus características más importantes que *es configurable y reconfigurable*. La reconfiguración se puede realizar en tiempo de ejecución, lo que ofrece una gran flexibilidad para satisfacer las necesidades de un gran número de aplicaciones. Por ello, esta característica ofrece una gran flexibilidad para satisfacer las necesidades de distintas aplicaciones.

Por otro lado, dependiendo de la aplicación que se desee realizar, puede que requiera un middleware con características de tiempo real, el cual soporta peticiones sensibles al tiempo y políticas de planificación. La ventaja principal de esta clase de middleware es que provee de un proceso de decisión, de forma que se determina el mejor criterio para resolver procesos sensibles en el tiempo, así como también facilita la localización de recursos libres cuando se tienen tiempos límites de operación.

2.2.1. Desarrollo de software basado en componentes

Como complemento a lo descrito en el apartado del estado del arte, el desarrollo de software basado en componentes aparece con el objetivo de dar un paso más allá, respecto a la programación orientada a objetos. Un aspecto bastante importante respecto al desarrollo de componentes es que se obtienen entidades reutilizables y con una interfaz bien definida. De esta forma, al desarrollar nuevas aplicaciones usando estos componentes existentes bien definidos, el coste (sobre todo temporal) se reduce considerablemente. También, puesto que los componentes son unidades independientes respecto a la configuración, se pueden crear otras (con funcionalidades distintas) reemplazables, lo que permite más flexibilidad en el proceso de implementación y mantenimiento.

En conclusión, los objetivos principales del desarrollo de software basado en componentes son:

- Reutilización de los elementos de código fuente.
- Ganancia en costes de tiempo y desarrollo.
- Mejora de la calidad (puesto que se reutilizan componentes existentes).
- Necesidad de escribir menos código y así minimizar las probabilidades de cometer errores.
- Facilitar el mantenimiento y reemplazamiento de componentes (o su adaptación a nuevos requisitos de la aplicación).

2.3. Orocos

En este apartado se comentará todo lo relacionado con el middleware de tiempo real, el cual se ha utilizado para realizar el proyecto: OROCOS (“Open Robot Control Software”). Orocos, implementado en su totalidad en C++, permite la creación de módulos capaces de operar en tiempo real, siendo un aspecto clave a la hora de realizar el control del robot paralelo del presente proyecto.

Ya que el entorno Orocos está compuesto por varias librerías y funcionalidades, se comentará el potencial de cada una de ellas.

2.3.1. Historia y proyecto

La idea de iniciar un proyecto de Software Libre para el control del robots nació en diciembre de 2000, motivada por más de dos décadas de experiencias más bien decepcionantes, y fracasos al tratar de utilizar software comercial de control de robots para la investigación robótica avanzada. La idea, junto con un borrador de una posible propuesta de proyecto, se puso en marcha en la lista de correo de EURON, la Red Europea de Robótica. Este correo electrónico dio lugar a una gran cantidad de respuestas, a pesar de que fue enviado durante el período de Navidad. Dentro de unas dos semanas, una propuesta fue preparada

y enviada a la Unión Europea. Los contactos con el responsable de temas de software en la Unión Europea dejó claro que el tamaño del proyecto tenía que ser muy modesto, por lo que sólo se seleccionaron tres socios o responsables: Universidad Católica de Lovaina en Bélgica, LAAS Toulouse en Francia y la KTH de Estocolmo, en Suecia. Cada uno de estos tres grupos recibió únicamente la ayuda de una sola persona durante todo un año. El proyecto patrocinado por la UE se inició en septiembre de 2001, y tuvo una duración de dos años.

La patrocinación de la Unión Europea también proporcionó algunas ayudas, con el objetivo de invitar a reuniones del proyecto Orocós, a diferentes personas que no estaban implicados en el proyecto. Esto, junto con las herramientas clásicas de una página web y una lista de correo, generaron diversos debates e intercambio de ideas.

Una primera versión de lo que sería el núcleo en tiempo real del proyecto Orocós fue lanzado en el verano de 2002, pero fue muy preliminar y difícil de usar. En noviembre de 2002, la primera versión fue lanzada con la que se podía controlar la posición y velocidad de un robot manipulador con seis grados de libertad.

Después de que el proyecto patrocinado por la UE hubiera terminado, los socios del proyecto continuaron mejorando el software ya entregado. Tal fue el avance, que incluso el marco de desarrollo de tiempo real en Orocós de la KU Leuven, había conseguido lanzar 7 versiones importantes sobre septiembre de 2003. Por otro lado, en la KTH de Estocolmo, se realizaron diversas versiones basadas en los componentes más comunes en el marco de la robótica.

Debido a su aplicabilidad a las aplicaciones industriales, el marco Orocós en tiempo real se ha instaurado más en el campo de control de la máquina, y no en lo que es en sí la robótica, dejado atrás sus inicios en la robótica. La modularidad de los paquetes Orocós refleja una versatilidad y portabilidad aún desconocida. El Centro Tecnológico de mecatrónica de Flandes, patrocinó y financió el desarrollo desde 2005 hasta el año 2009, facilitando la integración de Orocós en máquinas industriales.

PAL Robotics financió el esfuerzo requerido para la creación de la versión 2.0 de la Orocós ToolChain (explicada en siguientes apartados), siendo necesaria una reunión de desarrolladores en Barcelona, con el objetivo de preparar el lanzamiento de esa versión.

Actualmente, SourceWorks es el principal patrocinador a la infraestructura de tiempo real Orocós ToolChain, mientras que muchas otras organizaciones contribuyen en la generación de las capas superiores. Por ejemplo, DFKI en Bremen (Alemania), ofrece a Rock Robotics las herramientas para la generación de código y componentes Orocós.

Los usuarios y colaboradores de todo el mundo, usan el software Orocós para el procesamiento de datos obtenido por sensores, la cinemática de una máquina, el control del robot. Como último apunte, los derechos de autor del código Orocós son compartidos por más de 20 colaboradores de diferentes países.

2.3.2. Kinematics and Dynamics Library (KDL)

Un esqueleto de un brazo robot en serie con seis articulaciones de revolución como el que se aprecia en la figura 8 es un ejemplo de una estructura cinemática. Mediante la utilidad que nos proporciona la KDL (librería de cinemática y dinámica) de Orocos, se consigue reducir el modelado y especificación del movimiento a un problema meramente geométrico (aunque con varios sistemas de referencia) con cálculos matemáticos, pudiéndose resolver un movimiento en base a unas especificaciones en cada una de las articulaciones.

La librería de Cinemática y Dinámica (KDL) desarrolla un marco de aplicación independiente para el modelado, y otro para el cálculo de cadenas cinemáticas en robots, modelos biomecánicos humanos, figuras animadas por ordenador, máquinas herramienta, etc. Además, la KDL proporciona librerías predefinidas de clases de objetos geométricos (como el punto o la línea), cadenas cinemáticas de varias familias (en serie, humanoide, paralelas o móviles) así como su especificación de movimiento e interpolación.

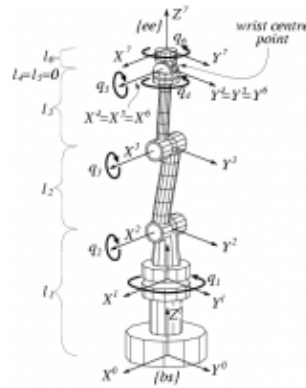


Figura 8: Brazo robot en serie

El modo de funcionamiento es tan sencillo como definir los segmentos de las articulaciones y el “solver”:

```
//Definition of a kinematic chain & add segments to the chain
KDL::Chainchain;
chain.addSegment(Segment(Joint(Joint::RotZ),Frame(Vector(0.0,0.0,1.020)));
chain.addSegment(Segment(Joint(Joint::RotX),Frame(Vector(0.0,0.0,0.480)));
chain.addSegment(Segment(Joint(Joint::RotX),Frame(Vector(0.0,0.0,0.645)));
chain.addSegment(Segment(Joint(Joint::RotZ)));
chain.addSegment(Segment(Joint(Joint::RotX),Frame(Vector(0.0,0.0,0.120)));
chain.addSegment(Segment(Joint(Joint::RotZ)));
// Create solver based on kinematic chain
ChainFkSolverPos_recursivefksolver=ChainFkSolverPos_recursive(chain);
```

Indicarle las posiciones finales de algunas articulaciones y, finalmente, calcular la cinemática inversa:


```

kinematics_status=fksolver.JntToCart(jointpositions, cartpos);
if(kinematics_status>=0)
{ std::cout<<cartpos<<std::endl;
printf("%s \n", "Suces, thanks KDL!"); }
else
{ printf("%s \n", "Error: could not calculate forward Kinematics"); }

```

2.3.3. Real Time Toolkit (RTT)

Dentro del proyecto Orocos, se encuentra uno de los paquetes más importantes, siendo el *motor* de tiempo real de todo el sistema. Concretamente, la *Real Time Toolkit* aporta un marco en C++, con el objetivo de aplicarlo a sistemas de control, ya sean de tiempo real como no. En la figura 9 se puede apreciar la idea principal.

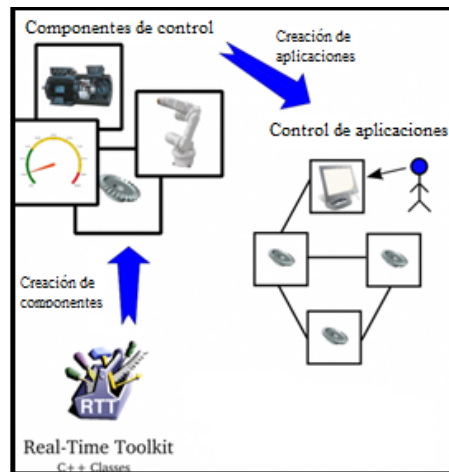


Figura 9: RTT Services

Entre otras cosas, la Real Time Toolkit permite a los desarrolladores la creación de componentes totalmente configurables (incluso en tiempo de ejecución) e interactivos basados en el control de aplicaciones de tiempo real (figura 10), pudiéndose usar en aplicaciones con características tales como:

- Capturar y graficar el flujo de datos entre los componentes.
- Reasignar prioridades y modificar el algoritmo de control en tiempo de ejecución.
- Configurar los componentes y la aplicación a partir de archivos XML.
- Interactuar con otros dispositivos directamente desde una interfaz gráfica de usuario o mediante el prompt de Linux.

- Ampliar las aplicaciones con estructuras de datos propias.
- Ejecutar la aplicación tanto en sistemas operativos estándar como sistemas de tiempo real.

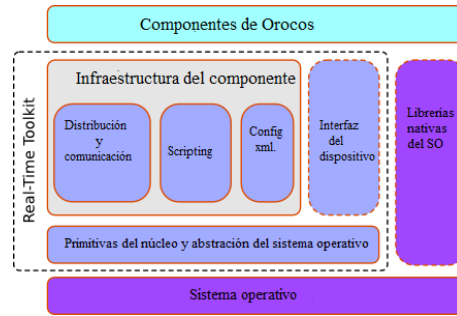


Figura 10: RTT Stack

2.3.4. Bayesian Filtering Library (BFL)

La Bayesian Filtering Library (BFL) proporciona un marco de aplicación independiente para, por ejemplo, el procesamiento de la información recursiva y algoritmos de cálculo basado en la regla de Bayes, tales como filtros de Kalman o filtros de partículas. Estos algoritmos pueden, por ejemplo, ser ejecutados como una aplicación en tiempo real, o ser utilizado para la estimación de las aplicaciones de Cinemática y Dinámica (figura 11).

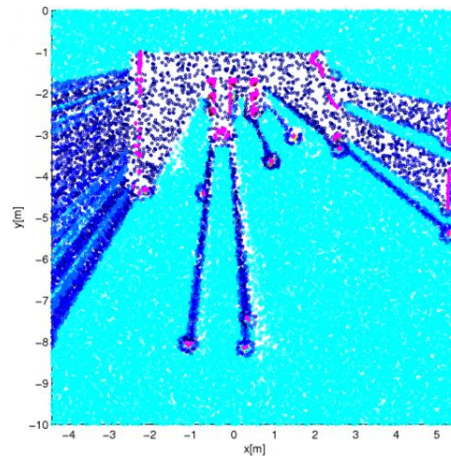


Figura 11: Estimación mediante BFL

Cabe destacar que en la actualidad existen aplicaciones similares a la que nos propone OrocOS, llegando a ser, considerablemente mejores (debido a que

son específicas en aspectos relacionado con los filtros) y más sencillas en su manejo. Por ello, en desarrollo en los últimos años de la BFL es mínimo, ya que prácticamente ni se usa.

2.3.5. Orocos Component Library (OCL)

La *Orocos Component Library* es una parte importante del proyecto Orocos, ya que mediante ésta, se crean todos los componentes (apoyándose de en la RTT, anteriormente comentada).

Como se explicará en posteriores apartados, cada uno de los componentes está definido a partir de una primitiva llamada “TaskContext”, la cual define el entorno (contexto) mínimo o básico que debe tener un componente en Orocos, que luego se modifica y configura según las preferencias del usuario.

Un componente en sí es una unidad básica de funcionalidad que puede ejecutar uno o más programas (en tiempo real) en un único hilo o “thread”. Por otro lado, el sistema está libre de prioridades (pese a que un usuario avanzado las puede cambiar) y todas las operaciones son no bloqueantes. Otro aspecto importante es que los componentes en tiempo real pueden comunicarse con los componentes que no lo son, y viceversa, de forma transparente.

En el siguiente ejemplo (figura 12) se puede observar como varios componentes (unos pertenecientes a un sistema de tiempo real, y otros no) se pueden comunicar gracias a que la propia Real Time Toolkit se encarga de gestionar esa comunicación de la manera más eficiente posible.

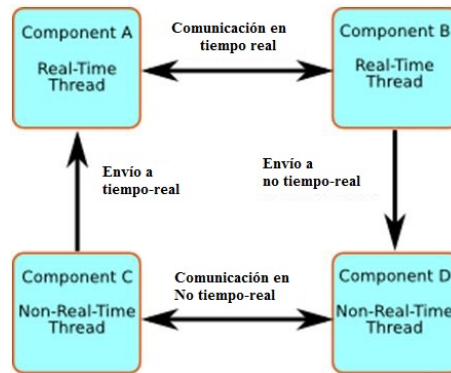


Figura 12: Comunicación entre varios componentes

2.3.6. OrocosToolChain

La Orocos ToolChain es la creación más reciente del proyecto Orocos. Su primera versión apareció por julio de 2010 y, poco a poco, se está convirtiendo en la herramienta más usada.

La particularidad de la ToolChain es que en esta herramienta vienen incluidas otras herramientas como:

- `AUTOPROJ`, que es una herramienta para descargar y compilar las librerías necesarias de forma automática.
- `REAL TIME TOOLKIT` (anteriormente descrita).
- `OROCOS COMPONENT LIBRARY` (anteriormente descrita).
- `OROGEN` Y `TYPEGEN`, que son dos herramientas para generar código OrocOS a partir de unas cabeceras descritas, o de un fichero de descripción del componente (con una sintaxis ya predefinida).

La última versión que salió fue la 2.5.0, en octubre de 2011, y es la que se está utilizando actualmente en este proyecto.

De este modo, y en un único paquete, está todo lo necesario para comenzar a realizar componentes que tengan distintas funcionalidades. Además, una de las ventajas de la OrocOS ToolChain es su soporte multiplataforma, ya que se puede trabajar tanto en Linux como en Windows o MAC (en algunos casos, sin utilizar la parte del tiempo real).

3. Desarrollo práctico

En esta sección se comentará todo lo relacionado con el desarrollo práctico que se ha realizado para la elaboración del proyecto.

En primer lugar se comentará el entorno de desarrollo (tanto hardware como software), para posteriormente pasar a explicar la estructura de los componentes en OrocOS.

Finalmente, se abordarán las diferentes propuestas modulares que se han usado, así como sus ventajas e inconvenientes.

3.1. Plataforma de desarrollo hardware

A continuación se pasará a comentar los elementos hardware que han sido necesarios para el desarrollo del proyecto.

3.1.1. Equipo industrial

El equipo industrial utilizado ha sido un IEC RACK-360GW-R20 (figura 13). Este PC Industrial tiene los siguientes componentes:

- Intel(R) Core(TM)2 Quad (4 núcleos) CPU Q8300 a 2.5 GHZ.
- 4GB de memoria RAM.
- 500 GB de Disco Duro.
- 7 Slots para tarjetas PCI.
- 5 Slots para tarjetas ISA.

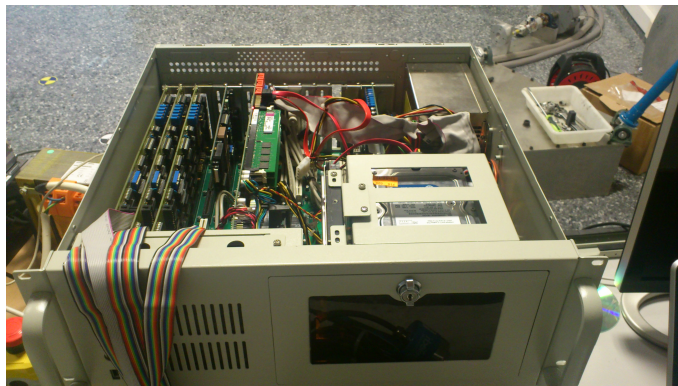


Figura 13: Equipo industrial

Es preciso un ordenador potente y robusto, puesto que, además de tener que realizar un elevado número de operaciones en el algoritmo de control, es preciso que funcione correctamente en tiempo real.

Además, como se apreciará en secciones posteriores, habrá situaciones en las que el control no puede fallar (por ejemplo, cuando esté interactuando un ser humano), de ahí un PC Industrial de estas características.

3.1.2. Advantech PCI 1720

La PCI 1720 es una tarjeta de conversión digital/analógica, por el bus PCI, creada por Advantech (figura 14). Esta tarjeta contiene cuatro canales de 12 bits para la salida analógica, además de la posibilidad de elegir entre cuatro rangos de tensión. Estos rangos son los siguientes:

- 0~5V
- 0~10V
- $\pm 5V$
- $\pm 10V$



Figura 14: PCI 1720

La posición de las articulaciones del robot irán cambiando dependiendo de la tensión que se le esté aplicando, por lo que, mediante la tarjeta PCI 1720, se irán aplicando distintos valores de tensión, que previamente habrán sido calculados mediante el algoritmo de control.

En este caso, puesto que hay que controlar los tres ejes del robot, como mínimo se necesitarán tres de las cuatro salidas que tiene la tarjeta, escogiendo un rango de tensión de $\pm 10V$, pese a que en el algoritmo final se saturará a $\pm 6V$ para evitar movimientos demasiado rápidos del robot.

Por tanto, si se tienen 12 bits para un rango de $\pm 10V$, la precisión será de 0.0048 V, por lo que sumado a un error de $\pm 0.024\%$ que indica el fabricante, es una buena aproximación.

En la siguiente tabla se pueden observar los pines más importantes, así como las conexiones que se han realizado usando la placa de conexiones PCLD-880 para, posteriormente, poder realizar la conexión con el robot.

Terminal	Descripción
A19	GND
A9	Channel 0
A15	Channel 1
B1	Channel 2
B7	Channel 3

Cuadro 1: Conexiones PCI-1720

Finalmente, otra de las razones por las que se ha elegido esta tarjeta ha sido porque la frecuencia de trabajo era suficiente acorde con las características del algoritmo de control.

3.1.3. Advantech PCL 833

La PCL 833 es una tarjeta de encoders de 3 ejes de cuadratura. A diferencia de la tarjeta 1720, ésta se conecta al bus ISA (y no al PCI). La PCL 833 de Advantech tiene tres contadores independientes de 24 bits (figura 15).



Figura 15: PCL 833

Puesto que los encoders de los motores del robot generan señales que indican información acerca de la posición de los ejes, la principal función de la tarjeta de encoders es recibir esa información e interpretarla, haciendo la conversión de esa secuencia de pulsos a cuentas del motor. Una vez se tengan los pulsos del motor, haciendo una conversión de pulsos a posición (de modo experimental o basándose en la especificación de los encoders) se puede controlar la cinemática y dinámica del robot.

De la misma forma que se ha hecho con la PCI 1720, para una mejor maniobrabilidad a la hora de cablear, se ha conectado a una placa las entradas de los tres encoders, siendo los pines más importantes lo que se pueden apreciar en la siguiente tabla:

Terminal	Descripción	Tipo
1	GND	
2	Counter 1	A+
3	Counter 1	B+
5	Counter 2	A+
6	Counter 2	B+
8	Counter 3	A+
9	Counter 3	B+
14	Counter 1	A-
15	Counter 1	B-
17	Counter 2	A-
18	Counter 2	B-
20	Counter 3	A-
21	Counter 3	B-

Cuadro 2: Conexiones PCL-833

3.1.4. ATI Force Sensor

Otro elemento hardware que se está utilizando actualmente en el proyecto es un sensor de fuerza, concretamente, el modelo Delta de la marca ATI (figura).



Figura 16: Ati Force Sensor

El sensor de fuerza ATI nos permite medir las seis componentes de fuerza y par. Se trata de un transductor, con un cable blindado de alta flexibilidad, y un sistema inteligente de adquisición de datos con un controlador de *DeviceNet*. Los sensores de fuerza se suelen usar en la industria para pruebas de productos, montaje robótico o pulido.

Por otro lado (y como se comentará posteriormente en esta tesina), en el campo de la investigación, este tipo de sensores se están usando cada vez más para tareas como cirugía, neurología o rehabilitación.

A pesar de ser un sensor tan sensible, las cargas máximas que indica el fabricante son realmente interesantes, tal y como se puede ver en la siguiente tabla.

Single Axis Overload	
Fxy	±2600 N
Fz	±8600 N
Txy	±290 Nm
Tz	±400 Nm

Cuadro 3: Cargas Sensor de Fuerza Ati

3.2. Plataforma de desarrollo software

Tras haber comentado en la sección anterior la plataforma de desarrollo hardware, en esta sección se explicará cómo, con esos elementos hardware, se consigue acceder a ellos mediante distinto software. También se justificará el sistema operativo utilizado, así como el núcleo de tiempo real.

3.2.1. Sistema Operativo

Un aspecto relevante en este proyecto es la elección del sistema operativo (Orocos permite tanto Windows como Linux), ya que tiene que cumplir con ciertas restricciones temporales (por lo que debe ser en tiempo real) y computacionales.

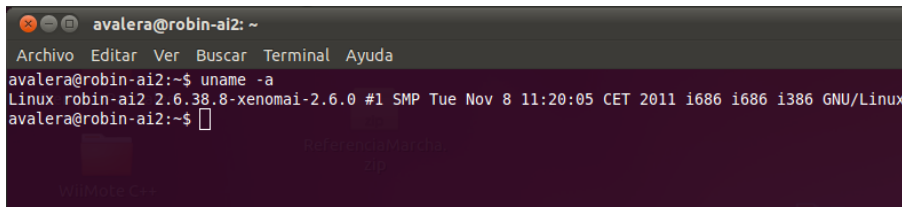
Por esta razón, ninguna versión de Windows es válida ya que, además de no ser en tiempo real, el código es totalmente cerrado, por lo que no se pueden hacer modificaciones.

Finalmente, la opción elegida ha sido un Linux con un *parcheado* que añada la parte de tiempo real. Concretamente, se ha instalado Ubuntu 11.04, un sistema operativo que utiliza un núcleo Linux, estando su origen basado en Debian, orientado a un usuario de nivel medio y con un fuerte enfoque en la facilidad de uso.

Por otro lado, una vez instalado el sistema operativo, hacía falta un parche para dotarlo de la funcionalidad típica de un RTOS (Real Time Operating System).

Se ha optado por Xenomai, que se basa en un núcleo de RTOS, de forma que se puede utilizar para la construcción de cualquier tipo de aplicación en tiempo real, sobre un núcleo genérico, como es el que tiene la versión 11.04 de Ubuntu.

Finalmente, tras la instalación del sistema operativo y el parche en tiempo real, al teclear la orden `$uname -a`, se obtiene la versión del núcleo y distribución que se está utilizando (figura 17).

A terminal window with a dark background and light text. The title bar shows 'avalera@robin-ai2: ~'. The menu bar includes 'Archivo', 'Editar', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'. The command 'uname -a' has been executed, resulting in the output: 'Linux robin-ai2 2.6.38.8-xenomai-2.6.0 #1 SMP Tue Nov 8 11:20:05 CET 2011 i686 i686 1386 GNU/Linux'. The prompt 'avalera@robin-ai2:~\$' is visible at the end of the line.

```
avalera@robin-ai2: ~
Archivo Editar Ver Buscar Terminal Ayuda
avalera@robin-ai2:~$ uname -a
Linux robin-ai2 2.6.38.8-xenomai-2.6.0 #1 SMP Tue Nov 8 11:20:05 CET 2011 i686 i686 1386 GNU/Linux
avalera@robin-ai2:~$
```

Figura 17: Uname -a

3.2.2. Lectura de los valores del encóder

Para la lectura de los valores del encóder, se ha usado la tarjeta PCL 833 (comentada en el apartado 3.1.3), que trabaja sobre el bus ISA.

Uno de los principales inconvenientes ha sido que el fabricante no aportaba los drivers para Linux, mientras que sí que los aportaba para Windows. Por tanto, puesto que se sabe que una tarjeta ISA trabaja directamente leyendo y escribiendo registros, si se conocía cuál era el registro base, con la ayuda del manual y los drivers escritos para Windows, se podría implementar los propios drivers.

De esta forma, mediante unos *switches* que incorpora la propia tarjeta, se seleccionó como dirección base de la misma la 0x220. A partir de ahí se implementó una librería para linux con funciones tales como:

- int vInitialize833(void);
- int vCh_Read(int option);
- int vCh_SetInputMode(int ChannelNo, int option);
- int vCh_DefineResetValue(int ChannelNo, int option);
- int vSetCascadeMode(int option);
- int vCh_SetLatchSource(int ChannelNo, int option);

Mediante estas funciones ya se podía inicializar la tarjeta, para indicarle la velocidad del reloj, por ejemplo, o leer el valor de los encóders.

Por tanto, para comenzar a utilizar la tarjeta, se debía incluir en el código fuente las siguientes líneas:

```
iopl(3);
Base= 0x0220;
pcl833(Initialize833,NA);
```

De esta manera, con la primera orden, el usuario tenía permisos de lectura y escritura sobre registros del sistema, por lo que ya podía inicializar la tarjeta con la dirección base especificada.

Posteriormente, tras haberla inicializado, para cada canal se debía configurar de la siguiente manera (en este ejemplo está el canal 1).

```

pcl833(Ch1_SetInputMode,x1);
pcl833(Ch1_DefineResetValue,middle);
pcl833(SetCascadeMode,c24bits); //no cascada

```

De esta forma se indica que no se quiere en cascada, y que el reset se haga al valor medio del contador. Esto es realmente importante, puesto que el robot puede comenzar su movimiento hacia una posición inferior (por lo que el contador del encóder haría un *underflow*). Así se puede asegurar que el contador no desborda ni por arriba ni por abajo.

Finalmente, para leer los valores de los canales, se procede con las siguientes instrucciones.

```

pcl833(Ch_Read,ch1);
ch1Dec=InReg[2]*65536 + InReg[1]*256 + InReg[0];
ch1Dec=ch1Dec-8388608;
ch1Dec=ch1Dec*0.02/1000;

```

Como se puede observar, cuando se lee el valor del canal 1, se obtiene en 3 bytes (puesto que el contador es de 24 bits). Lo que se hace para obtener el valor en decimal es multiplicar por 2^0 el byte de menor peso, por el 2^8 el siguiente, y por 2^{16} el de mayor peso, y después sumarlo.

Tras esto, como en la inicialización de la tarjeta se han puesto los contadores al valor medio (que es 8388608), ahora se resta para poder saber la posición verdadera sin tener que preocuparnos por un desbordamiento.

Finalmente, al multiplicar ese valor por $2 \cdot 10^{-5}$ se obtiene el desplazamiento lineal, en metros, de cada articulación.

Por lo tanto, de la misma forma que se ha hecho para el canal 1 (que corresponde a la articulación 1) también se hace para los canales 2 y 3, que representan a la segunda y tercera articulación.

3.2.3. Envío de la acción de control a los actuadores

Tras haber calculado la acción de control mediante el algoritmo de control correspondiente (que se explicará en posteriores apartados), es necesario enviar esa acción de control a los actuadores, con el objetivo de que se muevan las articulaciones del robot.

A diferencia de la tarjeta PCL833, que estaba conectada al bus ISA, la tarjeta PCI-1720 que se ha descrito en la sección 3.1.2 está conectada al bus PCI, por lo que el acceso a la misma es mucho más complicado en caso de no tener los drivers del fabricante, ya que el simple acceso a unos determinados registros no es válido.

La solución se encontró en el proyecto Comedi (“Control and Measurement Interface”), que desarrolla drivers, herramientas y librerías en código abierto para la adquisición de datos. Comedi es una colección de drivers para una gran variedad de tarjetas relacionadas con la adquisición de datos. Además, los drivers están implementados como un “módulo del núcleo de Linux”, teniendo la funcionalidad típica de un módulo

Por otro lado, la “Comedilib” es una librería implementada en C, en la que se incluyen todas las funciones y métodos para poder manejar todo tipo de dispositivos. A parte de proporcionar al usuario todas las funciones y programas de configuración de las tarjetas, se incluyen diversos programas de testeo y prueba, sirviendo de ejemplo en algunos casos.

Por tanto, insertando el módulo de nuestra tarjeta en cuestión, se consigue que reconozca la tarjeta D/A y así poder enviar la acción de control.

Lo primero que hay que realizar es configurar la tarjeta de la siguiente manera.

```
int configuraTarjetapci1720(void)
{ cf = comedi_open("/dev/comedi0");
  maxdata = comedi_get_maxdata(cf, subdev, chan);
  cr = comedi_get_range(cf, subdev, chan, range);
  return 1; }
```

De esta forma, se abre el descriptor de fichero de la tarjeta y se configura, para así poder sacar un valor de tensión posteriormente, mediante la siguiente función.

```
int sendVolts(double val, int canal)
{ data=comedi_from_phys(val, cr, maxdata);
  printf("Canal %d Data %d volts: %g\n", canal, data, val);
  result= comedi_data_write(cf, subdev, canal, range, AREF_GROUND, data);
  if (result==-1)
  {comedi_perror("/dev/comedi0");
   exit(1);}
  return 1;}
```

Como se puede observar, a la función *sendVolts* se le pasa como parámetros de entrada el valor de tensión (en voltios) y el canal por el que se quiere sacar ese valor. Dentro de la función es necesario hacer una conversión mediante la función *comedi_from_phys* que, según el subcanal que se haya elegido (en este caso ha sido $\pm 10V$) será un valor u otro.

Finalmente, mediante la función *comedi_data_write* se envía ese valor de tensión por el canal especificado.

3.2.4. Lectura de los valores de fuerza

Además de leer los valores de posición, mediante una tarjeta A/D, y enviar la acción de control, a través de una tarjeta D/A, en las fases finales de este proyecto se ha incluido un sensor de fuerza, por lo que ha sido necesario leer esos valores del sensor.

A diferencia de los dos puntos anteriores, en esta ocasión esa lectura se ha realizado mediante la toma de red del PC, puesto que el protocolo para esa toma de datos era DeviceNet.

DeviceNet es un protocolo de comunicación usado en la industria de la automatización para interconectar dispositivos de control para intercambio de datos. Éste usa Bus CAN como tecnología en la que se apoya y define una capa de aplicación para cubrir un rango de perfiles de dispositivos.

Lo primero que hay que realizar es configurar el sensor y realizar una puesta a cero.

```

*(uint16*)&request[0] = htons(0x1234); /* standard header. */
*(uint16*)&request[2] = htons(COMMAND); /* COMMAND=0X0042 */
*(uint32*)&request[4] = htonl(NUM_SAMPLES); /* NUM_SAMPLES=1 */
/* Sending the request. */
he = gethostbyname("192.168.1.1");
memcpy(&addr.sin_addr, he->h_addr_list[0], he->h_length);
addr.sin_family = AF_INET; addr.sin_port = htons(PORT);
err = connect( socketHandle, (struct sockaddr *)&addr, sizeof(addr) );
send( socketHandle, request, 8, 0 );

```

Como se puede ver en la figura 18, dependiendo del valor de COMMAND se realiza una acción u otra.

Command	Command Name	Command Response
0x0000	Stop streaming	none
0x0002	Start high-speed real-time streaming	RDT record(s)
0x0003	Start high-speed buffered streaming	RDT record(s)
0x0004	Start multi-unit streaming (synchronized)	RDT record(s)
0x0041	Reset Threshold Latch	none
0x0042	Set Software Bias	none

Figura 18: Tabla DeviceNet

En primer lugar, el campo de la cabecera es un valor estándar (0x1234). Posteriormente, el campo del comando debe ser, en este caso, 0x0042 ya que se quiere hacer un reset. Este reset es necesario porque a medida que el sensor se va calentando, los valores leídos difieren en cierta medida. Después se indica la dirección IP de la tarjeta de red por la que se van a enviar y recibir los datos, y se hace la conexión del socket. Finalmente, mediante ese socket se envía la orden.

Una vez se ha configurado, no hay más que llamar a la función *leer_fuerza* para obtener los valores de fuerza y par (Fx, Fy, Fz, Tx, Ty, Tz).

```

void leerFuerza(double ret_fuerza[6]) //Salida en Newtons: Fx, Fy, Fz, Tx, Ty, Tz
{
*(uint16*)&request[0] = htons(0x1234); /* standard header. */
*(uint16*)&request[2] = htons(COMMAND); /* COMMAND=0X0002 */
*(uint32*)&request[4] = htonl(NUM_SAMPLES);
send( socketHandle, request, 8, 0 );
recv( socketHandle, response, 36, 0 );
resp.rdt_sequence = ntohl(*(uint32*)&response[0]);
resp.ft_sequence = ntohl(*(uint32*)&response[4]);
resp.status = ntohl(*(uint32*)&response[8]);
for( i_fuerza = 0; i_fuerza < 6; i_fuerza++ )
{
resp.FTData[i_fuerza] = ntohl(*(int32*)&response[12 + i_fuerza * 4]);
ret_fuerza[i]=1.0*resp.FTData[i]/1000000.0; }
}

```

Como se puede ver en el anterior código, en esta ocasión el valor de `COMMAND` (0x0002) indica que se quieren obtener los valores de la fuerza en tiempo real (si fuera 0x0003, tal y como dice el fabricante, no se estaría asegurando la lectura en tiempo real, ya que el valor obtenido vendría a través de un *buffer*). Posteriormente se envía la solicitud y se recibe la respuesta. Finalmente, el valor obtenido se divide entre 10^6 para pasar de *ForceUnits* en Newtons.

De esta forma, dentro del vector `ret_fuerza[6]` se obtienen los valores de fuerza y par en cada iteración.

3.3. Componentes en Orocos

En esta sección se abordará todo lo relacionado con los componentes en Orocos, una de las partes más importantes de todo el proyecto. Además de comentar las partes y estructuras de un componente, se explicará cómo se define un flujo de datos entre componentes (mediante puertos de entrada y salida y la interconexión).

Es muy importante conocer a la perfección cuál es su funcionamiento, puesto que mediante los mismos se deberán implementar los diferentes controles dinámicos, siendo la base de este estudio.

3.3.1. Partes de un componente

Como se ha comentado anteriormente, el proyecto Orocos está implementado en C++, por lo que en toda la programación de los controladores se ha utilizado el estándar de C++. Puesto que una de las características del middleware Orocos es la creación de componentes que funcionen en tiempo real, cada componente debe tener una determinada estructura para que, al compilarlo, se obtenga un módulo que se pueda ejecutar de forma independiente a los demás.

Esta estructura, que se comentará en el siguiente apartado, debe tener en cuenta las distintas partes que tiene un componente (figura 19):

- **PUERTOS DE ENTRADA.** Mediante los puertos de entrada, se obtienen los datos de otros componentes o de sensores externos.
- **PUERTOS DE SALIDA.** Con los puertos de salida se envían los datos a otros componentes, o a actuadores externos.
- **ATRIBUTOS.** Variables propias con un determinado valor.
- **MÉTODOS.** Secuencia de código que se ejecuta cuando se llama a esa función.
- **EVENTOS.** Mediante los eventos, se consigue que se ejecute una secuencia de código cuando se produce un evento, tal como un nuevo dato por un puerto de entrada

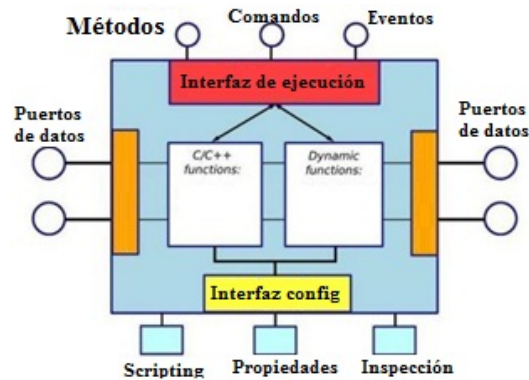


Figura 19: Partes de un componente

3.3.2. Estructura interna de un componente

En el apartado anterior, se han visto cuales son las partes más importantes de un componente de OROCOS, aunque no se ha abordado el tema de la estructura interna. Cabe destacar que todos los módulos creados deben tener una estructura básica (independientemente de los métodos o funciones que se necesiten añadir), la cual puede descomponerse en las siguientes partes y funciones:

- **ZONA DE DECLARACIÓN DE VARIABLES GLOBALES.** En OROCOS es muy importante tener en cuenta todas las variables que se van declarando. Es altamente recomendable cuando se quiere que alguna variable vaya cambiando de valor en cada una de las iteraciones, que la variable se declare como variable global. En caso de que no se haga así (y se declare en una función que se ejecute iterativamente), cuando el componente realice la segunda iteración de forma automática, se producirá un error en tiempo de ejecución de “Violación de Segmento” (y todo lo que ello conlleva) ya que en ese supuesto se intentaría re-declarar la variable.
- **STARTHOOK().** El código que se añada en esta función se ejecutará inmediatamente antes de poner en modo running el componente. Generalmente en esta parte se suele establecer el periodo de ejecución y prioridad del módulo, en caso de no habérselo especificado anteriormente de forma interactiva.
- **CONFIGUREHOOK().** En esta parte de la implementación del componente definen las posibles variables globales que se hayan declarado previamente (tales como ganancias, por ejemplo), inicializándose también todos los dispositivos de entrada/salida, como la tarjeta de los convertidores D/A, la tarjeta de encoders o sensor de fuerza.
- **STOPHOOK().** De forma análoga a la función *StartHook()*, el código que se añada en esta función se ejecutará inmediatamente antes de poner en *stop*

el componente. De forma general, en esta zona de código se suele especificar un periodo de 0 (que indica que el componente se para), además de detener el hardware inicializado previamente. También, en caso de haber inicializado algún descriptor de fichero previamente (para graficar posteriormente la posición de una articulación, por ejemplo) en esta función se cierra ese descriptor.

- UPDATEHOOK(). Al contrario que las funciones anteriores, que sólo se ejecutan cuando se les llama (y sólo una vez), el código perteneciente a la función *UpdateHook()* se ejecuta de forma iterativa (según el periodo de muestreo especificado) cuando el componente pasa al estado running. Se debe tener especial precaución a la hora de definir variables en esta sección ya que, al ejecutarse todo el código de forma periódica, se puede producir una redeclaración de alguna variable, dando un fallo en tiempo de ejecución. En el caso específico de este trabajo, que se presentará posteriormente, en esta función (del módulo que hace de *generadorReferencias*) se calculan, para cada iteración, las referencias de q1, q2 y q3. De la misma manera, el módulo que hace de *controlador*, en esta función y a partir γ , β y z , calcula las acciones de control a aplicar.

```

class Prueba
  : public RTT::TaskContext
{protected:
  /*
  ZONA DE DECLARACIÓN DE VARIABLES GLOBALES
  */
public:
  Prueba(string const& name)
    : TaskContext(name)
  {
    std::cout << "Prueba constructed !" <<std::endl;
  }

  bool configureHook() {
    std::cout << "Prueba configured !" <<std::endl;
    return true;
  }

  bool startHook() {
    std::cout << "Prueba started !" <<std::endl;
    return true;
  }

  void updateHook() {
    std::cout << "Prueba executes updateHook !" <<std::endl;
  }

  void stopHook() {
    std::cout << "Prueba executes stopping !" <<std::endl;
  }
}

```

Figura 20: Estructura interna

3.3.3. Tipos y uso de puertos

En el entorno Orocos, otro aspecto muy importante es el flujo de datos entre los distintos componentes. Como se verá en los apartados posteriores, cuando

se tenga un esquema totalmente modular, será necesario que un cierto módulo, a través de su puerto de salida, le envíe una serie de datos a otro módulo, que lo recogerá por su puerto de entrada.

En primer lugar, puesto que Orocos se implementa en C++, se pueden usar los tipos o primitivas básicas de C++ para definir el tipo de datos. Por ejemplo, para crear un puerto de entrada y otro de salida, y que el flujo de datos sea un sólo elemento (un dato de tipo entero) se realizaría de la siguiente manera.

```
RTT::InputPort<int> puerto_entrada;  
RTT::OutputPort<int> puerto_salida;
```

De esta forma tan sencilla, se ha creado un puerto de entrada, de nombre *puerto_entrada*, y otro puerto de salida, de nombre *puerto_salida*, soportando ambos puertos datos de tipo *entero*.

Puesto que por regla general se va a necesitar que en vez de recibir (o enviar) un único dato (como en el ejemplo anterior), se envíe o reciba una serie de datos, en este caso se ha usado la clase *Vector*, ya definida en C++ para poder almacenar un vector de datos. Concretamente, se haría como se muestra a continuación.

```
RTT::InputPort< std::vector<double> > puerto_entrada;  
RTT::OutputPort< std::vector<double> > puerto_salida;
```

Si en vez de usar la clase *Vector* se quisiera usar cualquier otra estructura, se podría hacer de la misma forma que se crea una estructura en C/C++, siendo ello una gran ventaja.

Una vez se tienen definidos los puertos, el siguiente paso es obtener los datos entrantes (provenientes de otros módulos) y enviarlos. Es muy sencillo, puesto que el entorno Orocos ya tiene los métodos *write* y *read*, que se usan como se puede ver a continuación.

```
int valorLeido;  
puerto_entrada.read(valorLeido);  
puerto_salida.write(valorLeido);
```

De esta forma, se está leyendo un valor por un puerto, y se está enviando por el otro.

Finalmente, tras haber explicado la creación de puertos, tanto de entrada como de salida, es preciso comentar que Orocos cuenta con tres posibles tipos de puertos de entrada, que son los siguientes:

- **PUERTO NORMAL.** Este tipo de puerto tiene la característica que se puede leer de él en el momento que se quiera (por ejemplo, cuando haya algún dato nuevo). También, el hecho de que haya llegado un nuevo dato por ese puerto de entrada, no implica que se ejecute una determinado manejador o algún mecanismo similar.

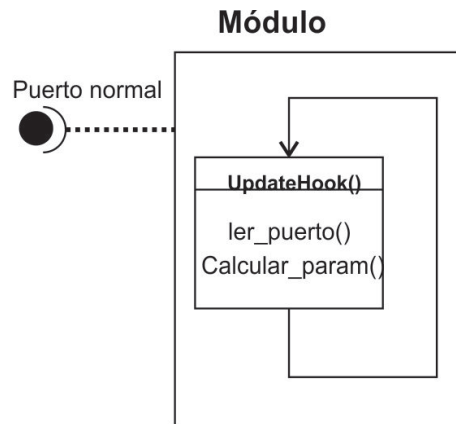


Figura 21: Puerto normal

- **PUERTO DE EVENTO.** La principal característica de este tipo de puertos de entrada es que cuando llega un dato nuevo, pasa a ejecutarse el método, anteriormente comentado, *UpdateHook()*. Mediante este tipo de puertos de entrada, con un único módulo que se ejecute de forma periódica, y que envíe ciertos datos por su puerto de salida, se consigue una ejecución en cascada, puesto que cada vez que llega un nuevo dato va desencadenando una ejecución.

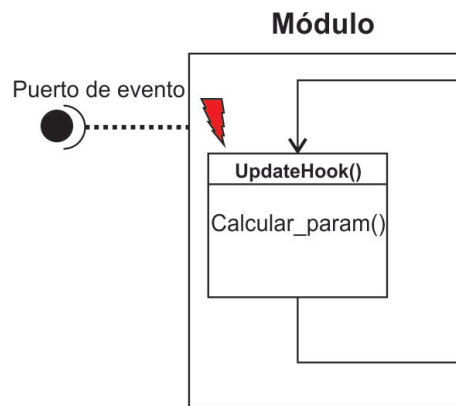


Figura 22: Puerto evento

- **PUERTO DE EVENTO ASOCIADO A FUNCIÓN.** A diferencia del puerto de evento, que se ejecuta toda la función *UpdateHook()* cuando llega un nuevo dato, en esta ocasión se le puede indicar qué función debe ejecutarse (a modo de manejador de instrucciones). En este proyecto, este tipo de puertos de entrada se ha utilizado cuando un módulo recibe datos de varios módulos, y sólo se puede ejecutar el método *UpdateHook()* cuando se

tienen todos los datos. Básicamente lo que se hace es que se tienen varios puertos de evento asociados a una función, y cuando han llegado nuevos datos por todos los puertos de entrada es cuando, finalmente, se ejecuta la función *UpdateHook()*.

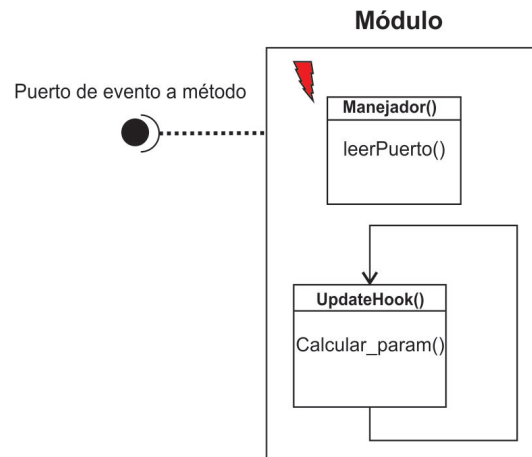


Figura 23: Puerto de evento asociado a función

3.3.4. Interconexión entre componentes

Tras haber explicado los diferentes puertos que existen en Orocos, así como su implementación en el código, a continuación se explicará cómo se conectan los puertos de un componente con otro.

Básicamente, existen una serie de restricciones a la hora de conectar dos componentes entre sí:

- Un puerto de salida sólo puede estar conectado con puertos de entrada, no permitiéndose (lógicamente) conexión de un puerto de salida con otro de salida.
- Puesto que cada puerto soporta un tipo de datos (tipo entero, vector de *doubles*, estructura creada por el programador), únicamente pueden conectarse puertos, cuyo tipo de datos sea exactamente el mismo. Tiene lógica, puesto que no es normal que un puerto de entrada espere un dato de tipo entero, y luego le llegue una estructura desconocida

Además, una característica de Orocos es que permite conexiones como se muestran en la figura 24.

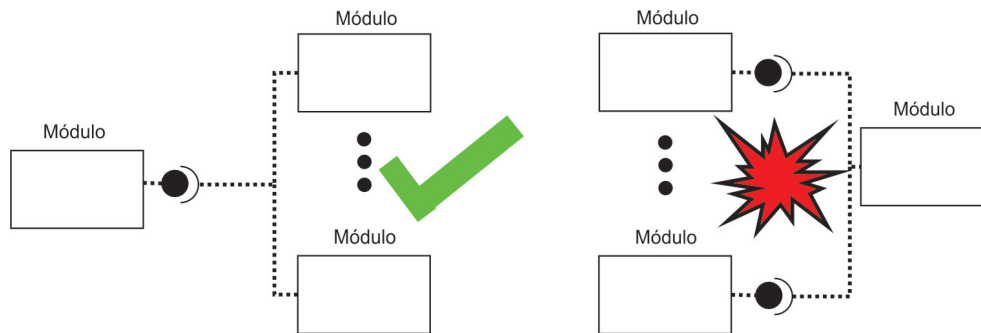


Figura 24: Conexiones de módulos

Como se puede ver en la figura anterior, de un único puerto de salida, se puede conectar a un número indefinido de puertos de entrada. Esto es muy útil (sobre todo cuando se haga la descomposición modular del control), puesto que se puede dar el caso en el que varios módulos necesiten los mismos datos de un mismo componente. En el caso contrario, no es correcto.

Puesto que cuando se crea (y compila) un componente no se sabe en tiempo de compilación qué nombre van a tener los puertos del resto de módulos, es necesario conectarlos de forma interactiva mediante el *deployer*. Por tanto, para tener acceso a los puertos creados, es necesario indicar (o añadir) en el código que en la interfaz se quiere que aparezcan esos puertos (y así conectarlos desde el *deployer*).

```

this->ports()->addPort("nombrePuertoInterfaz1", nombrePuerto1);
this->ports()->addEventPort("nombrePuertoInterfaz2", nombrePuerto2);
this->ports()->addEventPort("nombrePuertoInterfaz3", nombrePuerto3, boost:
:bind(&Clase::manejador, this, _1));

```

En el ejemplo anterior, tras haber creado 3 puertos de entrada, se ha indicado que uno de ellos sea un puerto de entrada *normal*, otro sea un puerto de evento y el último un puerto de evento asociado a un método (en este caso, en método *manejador*).

Por tanto, puesto que ya aparecen los puertos en la interfaz, cuando se esté en el *deployer* y se hayan cargado los módulos necesarios, para conectarlos no hay mas que llamar a la siguiente función.

```

connectTwoPorts (string &oneComp, string &one_port, string &otherComp,
string &other_port)

```

De esta manera se conectan puertos de salida de un componente a puertos de entrada de otro de forma interactiva y justo antes de poner en ejecución el esquema modular.

3.4. Modelos cinemáticos del robot 3PRS

En el ámbito del proyecto de investigación Identificación de Parámetros Físicos en Sistemas Mecánicos Complejos, del Plan Nacional del Ministerio de Ciencia e Innovación se desarrolló un manipulador paralelo. La figura 25 muestra el modelo virtual del robot diseñado. La elección de la arquitectura y movimiento del robot paralelo vino determinada por la necesidad de desarrollar un robot de bajo coste capaz de generar rotación angular en dos ejes (roll y pitch) y elevación (z) como movimiento lineal. Se consideraron dos arquitecturas alternativas: 3-RPS y 3-PRS. Se seleccionó la arquitectura 3-PRS tras comparar las ventajas e inconvenientes de cada una de las alternativas.

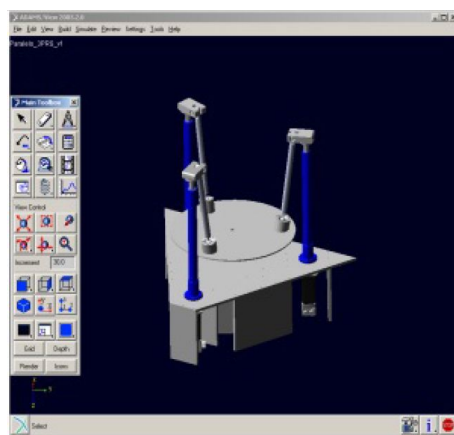


Figura 25: Modelo CAD del Robot paralelo 3PRS

Por ejemplo, una de las ventajas de la arquitectura PRS es que los actuadores se localizan en la base fija mientras que en la arquitectura 3-RPS, los actuadores se mueven junto con las articulaciones de revolución.

El robot paralelo diseñado puede verse como tres patas que conectan la plataforma móvil con la base fija. Cada pata consiste en un motor que mueve un actuador de husillo de bolas y un vástago de conexión.

3.4.1. Cinemática directa

La cinemática directa del manipulador paralelo consiste en, dados los movimientos lineales de los actuadores, encontrar los ángulos de balanceo (β) y alabeo (γ) y la elevación (z). Se puede utilizar la notación de Denavit-Hartenbert para establecer las coordenadas generalizadas del modelo cinemático. La tabla 4 muestra los parámetros D-H del robot considerado.

i	1	2	3	4	5	6	7	8	9
d_i	q_1	0	0	0	0	q_6	0	q_8	0
α_i	0	0	l_a	0	0	0	0	0	0
θ_i	$\frac{\pi}{6}$	q_2	q_3	q_4	q_5	$\frac{5\pi}{2}$	q_7	$-\frac{\pi}{2}$	q_9
α_i	0	$\frac{\pi}{2}$	0	$\frac{\pi}{2}$	$\frac{\pi}{2}$	0	$\frac{\pi}{2}$	0	$\frac{\pi}{2}$

Cuadro 4: Parámetros D-H para el robot de 3DOF

A partir de la tabla 4 se puede ver como a partir de 9 coordenadas generalizadas, se puede definir la cinemática del robot. La localización de los sistemas de coordenadas para modelar la cinemática se muestra en la figura 26.

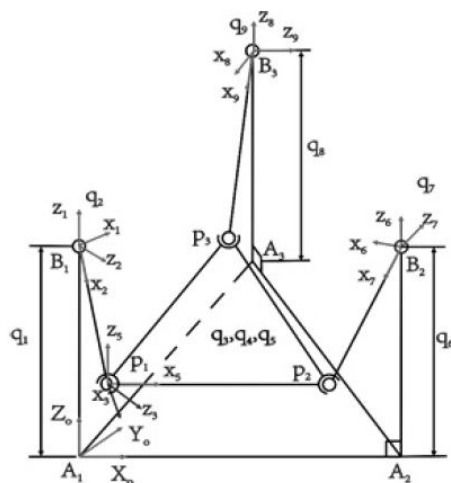


Figura 26: Localización sistemas de coordenadas

Las tres patas del robot paralelo se conectan a la plataforma móvil formando un triángulo equilátero. Por ello se puede ver que la longitud entre p_i y p_j es constante e igual a l_m . De este modo,

$$f_1(q_1, q_2, q_6, q_7) = \|(\vec{r}_{A_1 B_1} + \vec{r}_{B_1 P_1}) - (\vec{r}_{A_1 A_2} + \vec{r}_{A_2 B_2} + \vec{r}_{B_2 P_2})\| - l_{m=0} \quad (2)$$

$$f_2(q_1, q_2, q_8, q_9) = \|(\vec{r}_{A_1 B_1} + \vec{r}_{B_1 P_1}) - (\vec{r}_{A_1 A_3} + \vec{r}_{A_3 B_3} + \vec{r}_{B_3 P_3})\| - l_{m=0} \quad (3)$$

$$f_1(q_6, q_7, q_8, q_9) = \|(\vec{r}_{A_1 A_3} + \vec{r}_{A_3 B_3} + \vec{r}_{B_3 P_3}) - (\vec{r}_{A_1 A_2} + \vec{r}_{A_2 B_2} + \vec{r}_{B_2 P_2})\| - l_{m=0} \quad (4)$$

En la cinemática directa, la posición de los actuadores es conocida, así, el sistema de ecuaciones (2) - (4) se puede ver como un sistema no-lineal con q_2 , q_7 y

q_9 como desconocidas. Para la resolución del sistema no lineal, el problema cinemático directo utiliza el método numérico de Newton-Rhapson ya que tiene una convergencia muy rápida (convergencia cuadrática) cuando la estimación inicial está cerca de la solución deseada (Jalón and Bayo, 1994). El método es iterativo y se puede escribir como:

$$\begin{bmatrix} q_2 \\ q_7 \\ q_9 \end{bmatrix}^{i+1} = \begin{bmatrix} q_2 \\ q_7 \\ q_9 \end{bmatrix}^i - J_i^{-1} \begin{bmatrix} f_1(q_2, q_7) \\ f_2(q_2, q_9) \\ f_3(q_7, q_9) \end{bmatrix}^i \quad (5)$$

En la ecuación anterior, i significa que las variables y funciones se evalúan en la iteración i . La matriz J es la matriz Jacobiana de f_i con respecto a las variables $[q_2, q_7, q_9]$. El proceso iterativo finaliza cuando:

$$\sqrt{(f_1(q_2, q_7)^i)^2 + (f_2(q_2, q_9)^i)^2 + (f_3(q_7, q_9)^i)^2} < \varepsilon \quad (6)$$

El parámetro ε es una cantidad positiva pequeña establecida en la programación del control.

El método de Newton-Rhapson requiere una aproximación inicial tan cercana como sea posible al valor solución. En este caso, esto no supone ningún problema ya que la posición inicial de la articulación que conecta la plataforma con el actuador es aproximadamente $\frac{2\pi}{5}$. La subsecuente aproximación inicial considera los valores de la posición previa del robot del robot.

La localización de la plataforma móvil se define usando el sistema de coordenadas unido a él. Una vez encontradas las coordenadas generalizadas para las patas del robot, se puede encontrar la posición de los puntos p_i . Estos tres puntos comparten el plano de la plataforma. Basado en estos puntos se puede construir la matriz rotacional de la plataforma con respecto a la base. Se define un eje local X_p como un vector unitario \vec{u} con la dirección dada por p_1, p_2 .

El eje Z_p se define mediante un vector \vec{v} y es un eje perpendicular al plano definido por los puntos p_1, p_2 y p_3 .

Finalmente, el eje Y_p se define mediante la dirección de los ejes \vec{w} el cual está determinado mediante el producto vectorial entre los ejes \vec{u} y \vec{v} . La matriz de rotación de la plataforma móvil viene dada por,

$$R_p = \begin{bmatrix} \vec{u}^T & \vec{v}^T & \vec{z}^T \end{bmatrix} \quad (7)$$

A partir de la matriz de rotación se pueden encontrar el resto de coordenadas generalizadas q_3, q_4 y q_5 .

3.4.2. Cinemática inversa

La cinemática inversa consiste en, dados los ángulos de balanceo y alabeo de la plataforma (β, γ) y la elevación (z), encontrar el movimiento lineal de los actuadores. Usando el sistema de ángulos fijo X-Y-Z; la matriz de rotación se puede definir como:

$$R_p = \begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma - s_\alpha s_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma - c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{bmatrix} \quad (8)$$

donde c^* y s^* hacen referencia al $\cos(*)$ y $\sin(*)$ respectivamente. Dados γ y β , el ángulo de cabeceo (α) se puede encontrar como sigue:

$$\alpha = \arctan2(s_\beta s_\gamma, (c_\gamma + c_\beta)) \quad (9)$$

Tras encontrar el ángulo, se pueden encontrar el resto de términos de la matriz de rotación. Las posiciones de los actuadores se pueden encontrar mediante las siguientes expresiones (10) - (12),

$$q_1 = p_x^2 + p_y^2 + p_z^2 + 2h(p_x u_x + p_y u_y + p_z u_z) - 2gp_x - 2ghu_x + g^2 + h^2 \quad (10)$$

$$q_6 = p_x^2 + p_y^2 + p_z^2 + h(p_x u_x + p_y u_y + p_z u_z) - \sqrt{3}h(p_x v_x + p_y v_y + p_z v_z) + g(p_x - \sqrt{3}p_y) + gh(u_x - \sqrt{3}u_y)/2 + gh(v_x - \sqrt{3}v_y)/2 + g^2 + h^2 \quad (11)$$

$$q_8 = p_x^2 + p_y^2 + p_z^2 + h(p_x u_x + p_y u_y + p_z u_z) - \sqrt{3}h(p_x v_x + p_y v_y + p_z v_z) + g(p_x - \sqrt{3}p_y) - gh(u_x - \sqrt{3}u_y)/2 + gh(v_x - \sqrt{3}v_y)/2 + g^2 + h^2 \quad (12)$$

donde: $h = \frac{l_m}{\sqrt{3}}$, $g = \frac{l_b}{\sqrt{3}}$, $p_x = -hu_y$, $p_y = -h(u_x - v_y)$, $p_z = zy$, l_b : longitud entre $A_i A_j$.

3.5. Modelo dinámico del robot 3PRS

Como se conoce, la mecatrónica es una combinación de la ingeniería mecánica, electrónica, informática y de control de procesos (Awtar et. al., 2002). Así, un aspecto muy importante en el desarrollo de dispositivos mecatrónicos es el sistema de control. Otro de los objetivos de este trabajo es el desarrollo de una arquitectura abierta de control que permita el desarrollo y análisis de esquemas de control dinámicos. Esta clase de controladores se basan en la ecuación dinámica del sistema a controlar, por lo que es necesario obtener los distintos términos que componen la ecuación. La ecuación dinámica de los robots manipuladores se puede expresar mediante la siguiente ecuación no lineal:

$$M(\vec{q}, \vec{\Phi}) \cdot \ddot{\vec{q}} + \vec{C}(\vec{q}, \dot{\vec{q}}, \vec{\Phi}) + \vec{G}(\vec{q}, \vec{\Phi}) = \vec{\tau} \quad (13)$$

donde $M(\vec{q}, \vec{\Phi})$, $\vec{C}(\vec{q}, \dot{\vec{q}}, \vec{\Phi})$ y $\vec{G}(\vec{q}, \vec{\Phi})$ son la matriz de masas del sistema, el vector de fuerzas centrífugas y de Coriolis y los términos gravitacionales respectivamente, y como se puede apreciar, dependen de los parámetros dinámicos $\vec{\Phi}$, \vec{q} y $\vec{\tau}$ son los vectores de las coordenadas y pares generalizados. Para poder

obtener los términos de la ecuación dinámica se debe construir el modelo para que éste esté en forma lineal a los parámetros dinámicos (Grotjahn et al., 2004), (Díaz-Rodríguez et al., 2010):

$$K(\vec{q}, \vec{\dot{q}}, \vec{\ddot{q}}) \cdot \vec{\Phi} = \vec{\tau} \quad (14)$$

En la ecuación 14 solo se puede identificar un conjunto de parámetros porque algunos de ellos tienen una contribución pequeña o insignificante en el comportamiento dinámico del sistema. Además, éstos son propensos al ruido en las medidas y a dinámicas no modeladas (Díaz-Rodríguez et al., 2008). Para obtener la identificación del modelo dinámico se ha utilizado una metodología que aparece en (Díaz-Rodríguez et al., 2010).

Los términos de la ecuación 13 se pueden obtener una vez realizada la identificación del proceso. Así, se pueden calcular los términos del vector gravitacional haciendo cero las velocidades y aceleraciones generalizadas de la ecuación 14:

$$K(\vec{q}, \vec{\dot{q}} = 0, \vec{\ddot{q}} = 0) \cdot \vec{\Phi} = \vec{G}(\vec{q}) \quad (15)$$

Los términos centrífugos y de Coriolis dependen de las coordenadas generalizadas y de las velocidades. Haciendo cero las aceleraciones generalizadas de la ecuación 14 otra vez se puede establecer que:

$$K(\vec{q}, \vec{\dot{q}}, \vec{\ddot{q}} = 0) \cdot \vec{\Phi} - \vec{G}(\vec{q}) + M^d A_d^{-1} \vec{b} = \vec{C}(\vec{q}, \vec{\dot{q}}) \quad (16)$$

Por último, la matriz de masa se puede determinar de la forma siguiente:

$$K(\vec{q}, \vec{\dot{q}} = 0, \vec{\ddot{q}} = 0) \cdot \vec{\Phi} - \vec{G}(\vec{q}) = M_i(\vec{q}) \quad (17)$$

Siendo $M_i(\vec{q})$ la columna i -ésima de la matriz de masas, y $\vec{e}_i = [0 \ \dots \ 1 \ \dots \ 0]^T$ un vector columna con un 1 en la posición i -ésima.

Por lo tanto, las ecuaciones diferenciales que describen la ecuación del movimiento (13) se puede construir utilizando las ecuaciones (19)-(17).

3.6. Algoritmos de control

Puesto en la actualidad se está trabajando con la implementación de varias clases de controladores (dinámica inversa, controladores adaptativos, etc.), en esta sección se va a comentar el desarrollo de controladores basados en la pasividad, dinámica inversa y control adaptativo.

3.6.1. Controladores basados en la pasividad

Los controladores basados en la pasividad resuelven el problema de controlar un robot mediante la explotación de la estructura física del sistema robotizado, y en especial de su propiedad de pasividad. La filosofía de diseño de estos controladores es reconfigurar la energía natural del sistema de tal forma que se consiga el objetivo de seguimiento del control (Ortega and Spong, 1989).

En el problema de control punto a punto (regulación del sistema), los controladores basados en pasividad se pueden ver como casos particulares de la siguiente ley de control general:

$$\tau_e = -K_p e - K_d \dot{q} - u \quad (18)$$

donde el error (e) es la diferencia de la posición real (q) y la posición deseada (q_d), y u varía dependiendo de la clase de controlador, de acuerdo con la siguiente tabla.

Controlador	u
PD+G	$-G(q)$
PD+G0	$-G(q_d)$
PID	$K_i \int_0^t e \cdot dt$

Cuadro 5: Controladores basado en pasividad

Control PD+G El controlador PD+G no es más que un PD con compensación de la gravedad. Este controlador está formado por dos partes:

1. Una realimentación lineal del estado.
2. La compensación de fuerzas de la gravedad del robot paralelo.

La segunda ley de control es una variación de la primera donde la compensación de la gravedad se hace en la posición final deseada, por lo que al ser unos términos constantes se pueden calcular de manera offline.

Estos controladores son muy simples, aunque tienen dos inconvenientes principalmente: el primero es la complejidad computacional del término gravitatorio que, dependiendo del robot y su modelo dinámico, ésta puede ser tan alta que sea imposible calcularlo en tiempo real. Por otro lado, el fenómeno de zona muerta o cualquier error en la estimación del término de gravedad puede causar una variación en el punto de equilibrio y por tanto un error de posición estacionario.

En la figura 27 se puede ver el esquema descrito en un modelo de *Matlab-Simulink*.

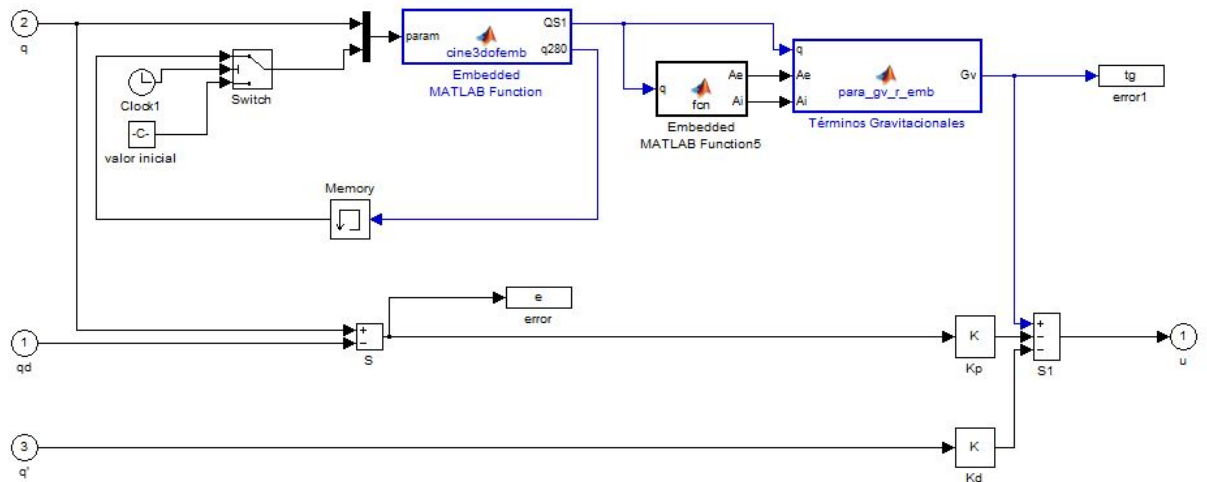


Figura 27: Esquema de control PD+G

Control PID Una posible solución práctica para intentar solucionar los problemas anteriormente comentados, es insertar una acción integral en la ley de control. Estas leyes son básicamente las mismas que en el PD pero la compensación de la gravedad se sustituye por la integral del error.

A continuación se puede ver el esquema propuesto.

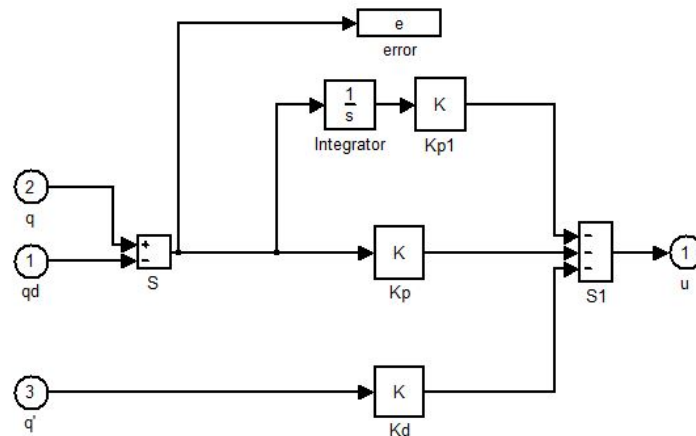


Figura 28: Esquema de control PID

Control de Paden-Panja Finalmente, en este trabajo se ha implementado el control propuesto por (Paden and Panja, 1988), en el que la energía cinética y potencial se modifica de forma adecuada para que el controlador sea global y asintóticamente estable. La expresión del controlador es la siguiente:

$$\tau_c = M(q)\ddot{q}_d + C(q, \dot{q})\dot{q}_d + G(q) - K_p e - K_d \dot{e} \quad (19)$$

En la ecuación 19 se puede apreciar cómo el controlador tiene dos partes:

- Una compensación de los términos dinámicos del robot (inercia, gravedad y Coriolis).
- Un controlador tipo proporcional-derivativo.

Su esquema en *Matlab-Simulink* es el siguiente.

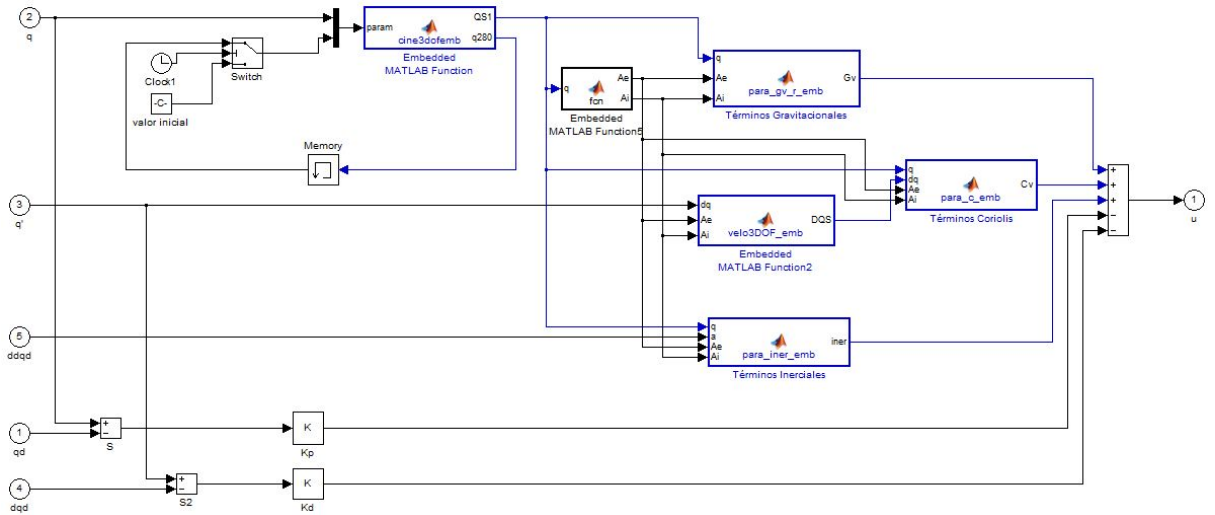


Figura 29: Esquema de control Paden

3.6.2. Controladores por dinámica inversa

Las estrategias de control basadas en la dinámica inversa se pueden incluir en la técnica de control de linealización por realimentación de la entrada. De esta forma, para que un sistema pueda ser linealizable (como es el caso de los sistemas robotizados) se debe cumplir que exista una transformación del espacio de estado y un estado de realimentación estático regular (invertible) que transforme el sistema no lineal en un sistema lineal. Así, para poder establecer el control mediante linealización por realimentación no lineal, se tiene que poder expresar la dinámica del sistema como:

$$\dot{x}^{(n)} = f(x) + b(x)u \quad (20)$$

donde $f(x)$ es una función de estados no lineal y u es la entrada de control. De esta forma se puede cancelar las no linealidades utilizando como entrada de control:

$$u = \frac{1}{b} [v - f] \quad (21)$$

obteniéndose así una relación simple entrada-salida, en la que $x^{(n)} = v$. En el caso del robot paralelo, la ecuación dinámica que se tiene es:

$$\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) \quad (22)$$

y si se despeja la aceleración se obtiene:

$$\ddot{q} = M^{-1}(q)(\tau - C(q, \dot{q})\dot{q} - G(q)) \quad (23)$$

Por tanto, a partir de las ecuación 20 y 23 se puede determinar que:

$$f(x) = M^{-1}(q)(-C(q, \dot{q})\dot{q} - G(q)) \quad (24)$$

$$b(x) = M^{-1}(q) \quad (25)$$

, que si se sustituye en la ecuación 21 se tiene:

$$\tau = \frac{1}{M^{-1}(q)} [v - M^{-1}(q)(-C(q, \dot{q})\dot{q} - G(q))] \quad (26)$$

A partir de la ecuación anterior, simplificándola, se obtiene:

$$\tau = M(q)v + C(q, \dot{q})\dot{q} + G(q) \quad (27)$$

Finalmente, a partir de la ecuación dinámica del robot (22) y de la ecuación 27 se tiene que:

$$M(q)v = M(q)\ddot{q} \quad (28)$$

por lo que si $M(q)$ es definida positiva, se obtiene que $\ddot{q} = v$.

Como se puede apreciar, la ecuación 27 tiene las matrices de la ecuación dinámica del robot (términos inerciales, términos de Coriolis y términos gravitacionales), por lo que realiza una linealización de la dinámica no lineal de éste. Por este motivo a esta técnica de control se le conoce como el *control por dinámica inversa o controlador de par de fuerzas calculado*. Estas estrategias fueron unas de las primeras técnicas de control no lineal que se desarrollaron. Así, por ejemplo, se pueden encontrar las aportaciones de [Paul, 72] o [Markiewicz, 73], [Raibert, Horn, 78] donde es la primera vez que se definen como técnicas por par calculado.

Como se ha visto anteriormente (ecuación 27), se puede obtener una expresión general de la acción de control basada en la dinámica inversa, de forma que dependiendo del objetivo de control que se requiera, se tienen varias elecciones posibles de v que se pueden observar en la siguiente tabla.

Ley de Control	v
Control Punto a Punto	$-K_d\dot{q} - K_p e$
Control por Trayectoria	$\ddot{q}_d - K_d\dot{e} - K_p e$

Cuadro 6: Controladores por Dinámica Inversa

donde:

$e = q - q_d$: error de posición

q_d : posición deseada

$K_p = K_p^T > 0$, $K_d = K_d^T > 0$, $K_i = K_i^T > 0$: ganancias proporcionales y derivativas.

Control punto a punto En el caso del control punto a punto, únicamente nos interesa la posición final a la que debe llegar el robot. Si se utiliza el valor de la acción del controlador lineal v en la expresión general obtenida anteriormente (ecuación 27), para un control punto a punto se tiene:

$$\tau = M(q)(-K_d\dot{q} - K_p e) + C(q, \dot{q})\dot{q} + G(q) \quad (29)$$

A partir de la deducción de la ecuación 28 se obtiene:

$$\ddot{q} + K_d\dot{q} + K_p q = K_p q_d \quad (30)$$

El error de posición tenderá de forma asintótica a cero, siempre y cuando K_p y K_d se escojan de forma que las raíces características de la ecuación anterior tengan la parte real negativa.

En la siguiente figura, está el modelo comentado anteriormente, implementado mediante *Matlab-Simulink*.

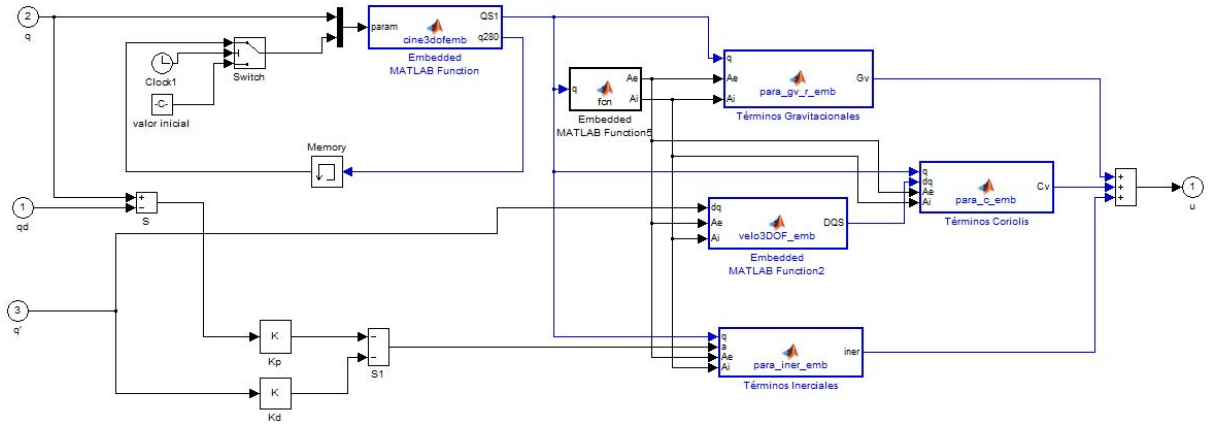


Figura 30: Esquema de control punto a punto

Control por trayectoria De la misma forma que en el control punto a punto, al escoger el valor adecuado de v , según la tabla 6 se tiene el control de trayectoria, expresado mediante la siguiente ecuación:

$$\tau = M(q)(\ddot{q}_d - K_d \dot{e} - K_p e) + C(q, \dot{q})\dot{q} + G(q) \quad (31)$$

Además, utilizando τ de la anterior ecuación como la acción de control del robot en cuestión se tiene:

$$\ddot{e} + K_d \dot{e} + K_p e = 0 \quad (32)$$

donde la convergencia exponencial del error está garantizada.

En la siguiente figura, está el modelo comentado anteriormente, implementado mediante *Matlab-Simulink*.

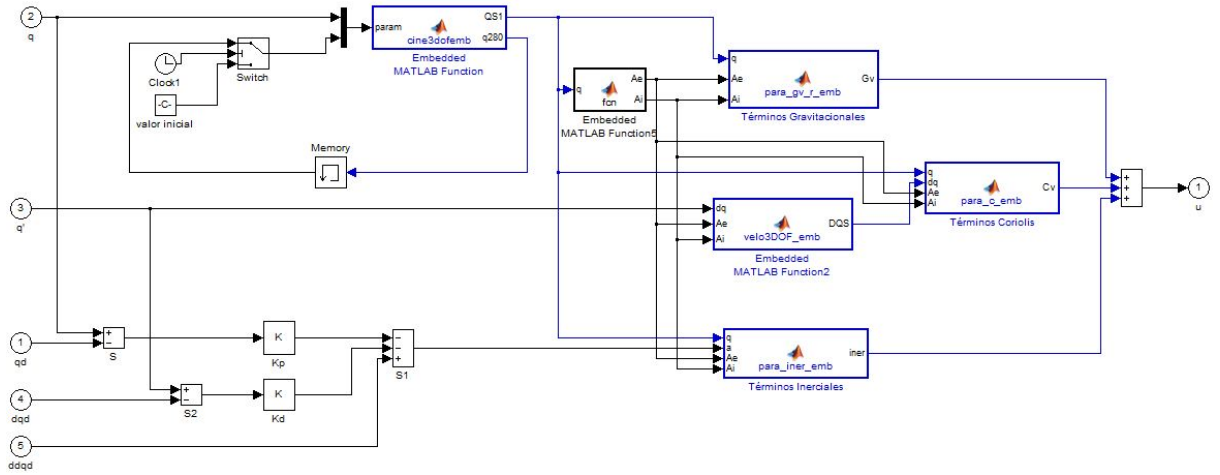


Figura 31: Esquema de control por trayectoria

3.6.3. Controlador adaptativo

Existen muchos sistemas dinámicos a controlar que tienen parámetros desconocidos constantes o que varían lentamente. Una forma correcta de establecer el control de estos sistemas puede ser mediante el control adaptativo. La idea básica del control adaptativo es estimar los parámetros del controlador correspondiente on-line basándose en las señales del sistema medidas, y utilizar los parámetros estimados en el cálculo de la entrada de control.

Para entender el control adaptativo, lo primero que hay que tener en cuenta es que la ecuación dinámica del robot paralelo se puede expresar con la ecuación no lineal siguiente:

$$\tau = M(q)\ddot{q} + C(q, \dot{q}) + G(q) \quad (33)$$

siendo el término gravitacional

$$G(q) = g \begin{bmatrix} G_{11} & G_{12} & G_{13} \\ G_{21} & G_{22} & G_{23} \\ G_{31} & G_{32} & G_{33} \end{bmatrix} \begin{bmatrix} \Omega_1 \\ \Omega_2 \\ \Omega_3 \end{bmatrix} \quad (34)$$

y el término de Coriolis

$$C(q, \dot{q}) = \begin{bmatrix} Fv_1\dot{q}_1 + F_{c_1} \text{sign}(\dot{q}_1) \\ Fv_2\dot{q}_2 + F_{c_2} \text{sign}(\dot{q}_2) \\ Fv_3\dot{q}_3 + F_{c_3} \text{sign}(\dot{q}_3) \end{bmatrix} + \begin{bmatrix} C_{11}(q, \dot{q}) & C_{12}(q, \dot{q}) & C_{13}(q, \dot{q}) \\ C_{21}(q, \dot{q}) & C_{22}(q, \dot{q}) & C_{23}(q, \dot{q}) \\ C_{31}(q, \dot{q}) & C_{32}(q, \dot{q}) & C_{33}(q, \dot{q}) \end{bmatrix} \begin{bmatrix} \Omega_1 \\ \Omega_2 \\ \Omega_3 \end{bmatrix} \quad (35)$$

Finalmente, el término inercial se puede representar de la siguiente manera:

$$M(q) = \begin{bmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \\ \ddot{q}_3 \end{bmatrix} + \begin{bmatrix} M_{11}(q, \ddot{q}) & M_{12}(q, \ddot{q}) & M_{13}(q, \ddot{q}) \\ M_{21}(q, \ddot{q}) & M_{22}(q, \ddot{q}) & M_{23}(q, \ddot{q}) \\ M_{31}(q, \ddot{q}) & M_{32}(q, \ddot{q}) & M_{33}(q, \ddot{q}) \end{bmatrix} \begin{bmatrix} \Omega_1 \\ \Omega_2 \\ \Omega_3 \end{bmatrix} \quad (36)$$

donde:

$$\Omega_1 = my_3 - \sin\left(\frac{2\pi}{3}\right) l_m \sum_{i=1}^5 m_i \quad (37)$$

$$\Omega_2 = \sum_{i=1}^5 m_i \quad (38)$$

$$\Omega_3 = mx_7 + l_r \sum_{i=1}^5 m_i \quad (39)$$

Asumiendo que θ es el vector de los parámetros que se desconocen, la ecuación de movimiento (ecuación 33) se podría describir de la forma siguiente:

$$\tau = M_0(q)\ddot{q} + C_0(q, \dot{q}) + G_0(q) + Y(q, \dot{q}, \ddot{q})\theta \quad (40)$$

donde M_0, C_0 y G_0 se corresponden a los términos de la ecuación dinámica que no están relacionados con los parámetros desconocidos, e $Y(q, \dot{q}, \ddot{q})$ es el vector de regresión, que contendrá los términos de la ecuación dinámica del robot que sí dependen de los parámetros desconocidos.

De esta forma, asumiendo que los parámetros desconocidos son los términos Ω_i , la ecuación 40 se podría describir de la siguiente manera:

$$\tau = \begin{bmatrix} J_1 & 0 & 0 \\ 0 & J_2 & 0 \\ 0 & 0 & J_3 \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \\ \ddot{q}_3 \end{bmatrix} + \begin{bmatrix} Fv_1\dot{q}_1 + F_{c_1} \text{sign}(\dot{q}_1) \\ Fv_2\dot{q}_2 + F_{c_2} \text{sign}(\dot{q}_2) \\ Fv_3\dot{q}_3 + F_{c_3} \text{sign}(\dot{q}_3) \end{bmatrix} + Y_1(q, \dot{q}, \ddot{q})\theta_1 \quad (41)$$

donde:

$$\begin{aligned}
Y_1(q, \dot{q}, \ddot{q}) = & \begin{bmatrix} M_{11}(q, \ddot{q}) & M_{12}(q, \ddot{q}) & M_{13}(q, \ddot{q}) \\ M_{21}(q, \ddot{q}) & M_{22}(q, \ddot{q}) & M_{23}(q, \ddot{q}) \\ M_{31}(q, \ddot{q}) & M_{32}(q, \ddot{q}) & M_{33}(q, \ddot{q}) \end{bmatrix} + \begin{bmatrix} C_{11}(q, \dot{q}) & C_{12}(q, \dot{q}) & C_{13}(q, \dot{q}) \\ C_{21}(q, \dot{q}) & C_{22}(q, \dot{q}) & C_{23}(q, \dot{q}) \\ C_{31}(q, \dot{q}) & C_{32}(q, \dot{q}) & C_{33}(q, \dot{q}) \end{bmatrix} + \\
& + g \begin{bmatrix} G_{11} & G_{12} & G_{13} \\ G_{21} & G_{22} & G_{23} \\ G_{31} & G_{32} & G_{33} \end{bmatrix}
\end{aligned} \tag{42}$$

En el caso en el que se consideren desconocidos los términos Ω_i y las inercias de los actuadores, la ecuación del movimiento se puede reescribir de la forma siguiente:

$$\tau = \begin{bmatrix} Fv_1\dot{q}_1 + F_{c_1} \text{sign}(\dot{q}_1) \\ Fv_2\dot{q}_2 + F_{c_2} \text{sign}(\dot{q}_2) \\ Fv_3\dot{q}_3 + F_{c_3} \text{sign}(\dot{q}_3) \end{bmatrix} + Y_2(q, \dot{q}, \ddot{q})\theta_2 \tag{43}$$

En este caso, el nuevo vector de regresión y de parámetros desconocidos son los siguientes:

$$\begin{aligned}
Y_2(q, \dot{q}, \ddot{q}) = & \begin{bmatrix} M_{11}(q, \ddot{q}) & M_{12}(q, \ddot{q}) & M_{13}(q, \ddot{q}) \\ M_{21}(q, \ddot{q}) & M_{22}(q, \ddot{q}) & M_{23}(q, \ddot{q}) \\ M_{31}(q, \ddot{q}) & M_{32}(q, \ddot{q}) & M_{33}(q, \ddot{q}) \\ \ddot{q}_1 & 0 & 0 \\ 0 & \ddot{q}_2 & 0 \\ 0 & 0 & \ddot{q}_3 \end{bmatrix} + \begin{bmatrix} C_{11}(q, \dot{q}) & C_{12}(q, \dot{q}) & C_{13}(q, \dot{q}) \\ C_{21}(q, \dot{q}) & C_{22}(q, \dot{q}) & C_{23}(q, \dot{q}) \\ C_{31}(q, \dot{q}) & C_{32}(q, \dot{q}) & C_{33}(q, \dot{q}) \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \\
& + g \begin{bmatrix} G_{11}(q) & G_{12}(q) & G_{13}(q) \\ G_{21}(q) & G_{22}(q) & G_{23}(q) \\ G_{31}(q) & G_{32}(q) & G_{33}(q) \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\end{aligned} \tag{44}$$

y

$$\theta_2 = \begin{bmatrix} \Omega_1 \\ \Omega_2 \\ \Omega_3 \\ J_1 \\ J_2 \\ J_3 \end{bmatrix} \tag{45}$$

Por tanto, de forma análoga a lo realizado anteriormente, en caso de considerar que todos los parámetros dinámicos fueran desconocidos, la ecuación dinámica del sistema sería la siguiente:

$$\tau = Y_3(q, \dot{q}, \ddot{q})\theta_3 \quad (46)$$

En este caso, el nuevo vector de regresión y parámetros sería el siguiente:

$$Y_3(q, \dot{q}, \ddot{q}) = \begin{bmatrix} M_{11}(q, \ddot{q}) & M_{12}(q, \ddot{q}) & M_{13}(q, \ddot{q}) \\ M_{21}(q, \ddot{q}) & M_{22}(q, \ddot{q}) & M_{23}(q, \ddot{q}) \\ M_{31}(q, \ddot{q}) & M_{32}(q, \ddot{q}) & M_{33}(q, \ddot{q}) \\ \ddot{q}_1 & 0 & 0 \\ 0 & \ddot{q}_2 & 0 \\ 0 & 0 & \ddot{q}_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} C_{11}(q, \dot{q}) & C_{12}(q, \dot{q}) & C_{13}(q, \dot{q}) \\ C_{21}(q, \dot{q}) & C_{22}(q, \dot{q}) & C_{23}(q, \dot{q}) \\ C_{31}(q, \dot{q}) & C_{32}(q, \dot{q}) & C_{33}(q, \dot{q}) \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \dot{q}_1 & 0 & 0 \\ 0 & \dot{q}_1 & 0 \\ 0 & 0 & \dot{q}_1 \\ \text{sign}(\dot{q}_1) & 0 & 0 \\ 0 & \text{sign}(\dot{q}_2) & 0 \\ 0 & 0 & \text{sign}(\dot{q}_3) \end{bmatrix} +$$

$$+g \begin{bmatrix} G_{11}(q) & G_{12}(q) & G_{13}(q) \\ G_{21}(q) & G_{22}(q) & G_{23}(q) \\ G_{31}(q) & G_{32}(q) & G_{33}(q) \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (47)$$

y

$$\theta_3 = \begin{bmatrix} \Omega_1 \\ \Omega_2 \\ \Omega_3 \\ J_1 \\ J_2 \\ J_3 \\ F_{V_1} \\ F_{V_2} \\ F_{V_3} \\ F_{C_1} \\ F_{C_2} \\ F_{C_3} \end{bmatrix} \quad (48)$$

Finalmente, a partir de la ecuación del movimiento del robot, expresado por la ecuación 33, se puede establecer el control adaptativo del robot paralelo mediante la expresión siguiente:

$$\tau = M_0(q)\ddot{q} + C_0(q, \dot{q}) + G_0(q) + Y(q, \dot{q}, \ddot{q})\hat{\theta} + K_p e + K_d \dot{e} \quad (49)$$

donde:

$$\frac{d}{dt}\{\hat{\theta}(t)\} = -\Gamma_0 Y^T(q, \dot{q}, \ddot{q})s_1 \quad (50)$$

El esquema del control adaptativo comentado anteriormente, implementado mediante *Matlab-Simulink* se puede ver en la figura 32.

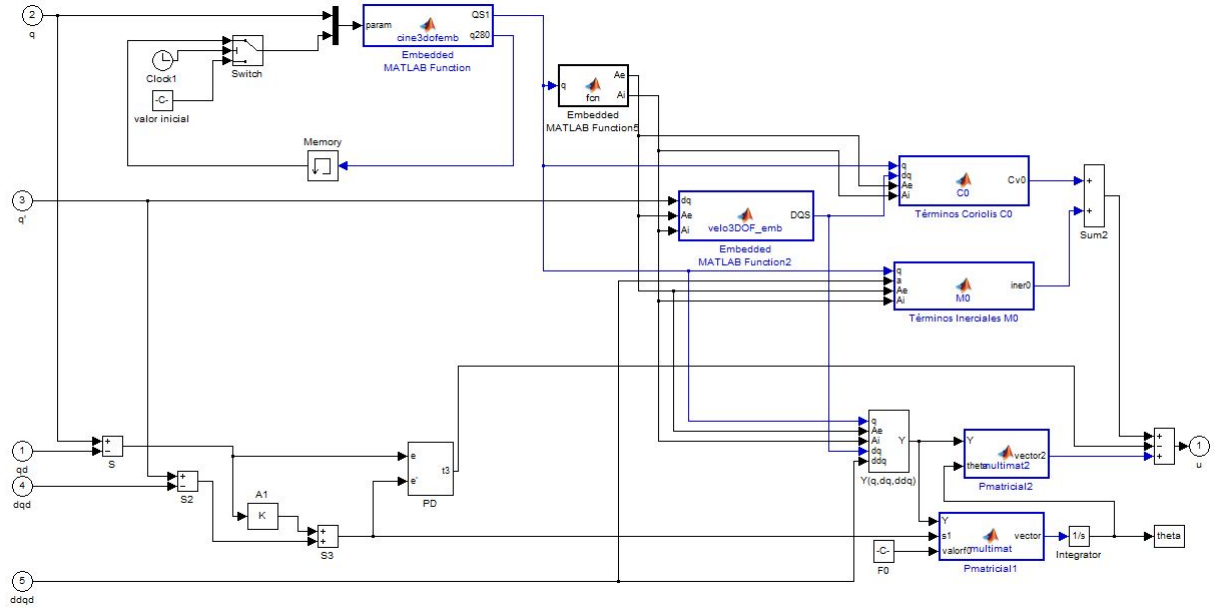


Figura 32: Esquema de control adaptativo

3.7. Configuraciones modulares usando Orocos

En este apartado se comentarán las distintas configuraciones que se han implementado y probado con el objetivo de realizar el control de la forma más óptima posible del robot paralelo, usando el middleware de tiempo real del entorno Orocos.

En primer lugar, se explicarán detalladamente las primeras soluciones propuestas y, a medida que se ha avanzado en este proyecto, se expondrán las soluciones con las que se han obtenido unos mejores resultados.

3.7.1. Solución inicial (un sólo módulo)

Para la resolución inicial del control del robot paralelo comentado en apartados anteriores, se propuso un esquema, el cual se dividiera en tres partes básicas:

- Lectura de datos del sensor
- Cálculo de la acción de control
- Aplicar la acción de control

Por ello, se implementó un único módulo estructurado de la siguiente manera, de acuerdo a lo comentado en el apartado en la estructura interna del componente (3.3.2):

- `CONFIGUREHOOK()`: Esta función sólo se ejecutará una vez (en el momento que se cargue el módulo mediante el *deployer*). Principalmente, en esta sección de código se inicializarán las tarjetas A/D (para la lectura del valor de los sensores del robot) y D/A (para aplicar la acción de control), así como los descriptores de fichero que se usarán para almacenar en texto plano parámetros tales como la posición real del robot y la acción aplicada, y así poder analizarlo mediante *Matlab*.
- `STOPHOOK()`: En esta zona de código (que también se ejecutará una sola vez, concretamente, al final de la ejecución), lo principal será cerrar correctamente las dos tarjetas anteriormente inicializadas y los descriptores de fichero también inicializados, además indicar que el periodo a partir de ahora será 0. En caso de que no se especifique que el periodo es cero (está parado), se producirá una violación de segmento.
- `UPDATEHOOK()`: Puesto que esta función se ejecuta de forma periódica (en función del periodo de muestreo, que en este caso será de 10ms), una forma de poder realizar el control es, ejecutar de forma secuencial las tres partes comentadas anteriormente. Es decir, nada más se entra a la función *UpdateHook()*, se lee el valor de los encoders. A partir del valor obtenido (posición real de las articulaciones del robot) se ejecuta el algoritmo de control (cualquiera de los comentados anteriormente) que calcula la acción de control a aplicar, y después se aplica mediante la tarjeta D/A.

En el siguiente esquema, se puede apreciar la idea principal.

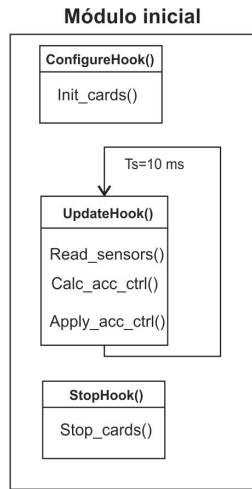


Figura 33: Solución inicial.

Como se puede observar, pese a que este esquema funciona perfectamente y los resultados (error de posición) son buenos, no se está haciendo una estructura modular, tal y como pretende el entorno Orocos, siendo una de sus máximas.

Por ejemplo, mediante el esquema de la figura 33 no se pueden asignar distintas prioridades, en función de los requerimientos del sistema, a cada una de las tres partes que conlleva el control. Esto no es posible, puesto que, únicamente se tiene un sólo módulo que hace las tres funciones de forma secuencial. Por ejemplo, lo ideal sería poder asignarle más prioridad a la lectura de sensores que al resto de partes, cosa que es imposible utilizando este esquema.

Como se ha comentado anteriormente, mediante esta primera propuesta, se consigue realizar el control del robot, aunque presenta grandes desventajas tales como:

- Dificultad a la hora de realizar un seguimiento del flujo de ejecución, ya que al ejecutarse todo el código de forma secuencial es muy complicado conocer por dónde va la ejecución (un único módulo).
- Imposibilidad de ejecución de código de forma paralela o distribuida, por lo que en caso de ser costosa una cierta operación, como el cálculo de los parámetros inerciales, por ejemplo, no se va a poder ejecutar en otro procesador, sino que se va a ejecutar todo línea a línea.
- Poca reusabilidad de código, puesto que para cualquier mínima modificación habría que escribir nuevo código.
- Difícil detección de un posible error de implementación, puesto que al no tener secciones verificadas, no se puede acotar ese error en ninguna parte del código.

- Imposibilidad de integración de módulos en el modelo, puesto que al estar todo encapsulado en un único módulo, no se pueden incluir en el modelo.

Por tanto, por las razones anteriores, aunque es un esquema que *funciona*, está muy lejos de las primitivas básicas que se deben respetar en cualquier desarrollo de software basado en componentes.

3.7.2. Solución modular

Después de implementar un único módulo para hacer todo el proceso del control, se decidió dividir todo el proceso, con el objetivo de implementarlo en varios módulos, y así cada uno se ejecute de manera independiente.

Por ello, basándonos en varias referencias que tratan sobre la implementación de software basado en componentes, se ha adaptado el esquema de la figura 33 a un esquema totalmente modular que se adapte correctamente con el entorno Orocos.

Se determinó que era necesario:

1. Un módulo para la lectura de datos de los encóders del robot, con lo que se obtendrá la posición real del robot.
2. Un módulo de control que calcule la acción de control, a partir de la posición real de los encóders.
3. Un módulo por cada uno de los actuadores (3) que aplique la acción de control.

En el siguiente diagrama, se puede observar el esquema propuesto.

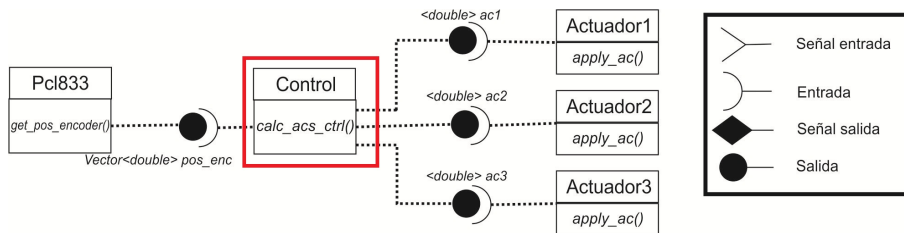


Figura 34: Solución modular.

En primer lugar, el módulo Pcl833 se usa para la lectura de los valores de los encóders del robot paralelo, teniendo como funciones destacadas las siguientes:

- CONFIGUREHOOK(): En esta función se inicializa y configura la tarjeta de adquisición de datos y se definen las diferentes variables que se utilizarán en el proceso de toma de datos.
- UPDATEHOOK(): Esta función, al ejecutarse de forma periódica dependiendo del periodo de muestreo, únicamente se encarga de recoger los valores de los encóders, normalizarlos (recogerlos de los registros y pasarlos a metros), y enviarlos por el puerto de salida hacia el módulo de control.

- `STOPHOOK()`: En esta parte de código, lo único que se realiza es detener la tarjeta inicializada.

Por otro lado, el módulo de control es el encargado de, con los datos previamente obtenidos del módulo `Pci833`, calcular una acción de control para enviársela a cada uno de los actuadores. Cabe destacar que el módulo de control puede ser tanto un PID, un PD+G o, incluso, un control pasivo de Paden y que, cumpliendo únicamente las restricciones de puertos de entrada y salida, el cambio de un control u otro no supone absolutamente ningún cambio en el resto de componentes. Por esto, se puede apreciar claramente una de las principales ventajas de haber cambiado el diseño respecto al apartado anterior.

Todos los módulos de control implementados, se han realizado con unas funciones comunes a todos ellos:

- `CONFIGUREHOOK()`: En esta zona de código se definen las variables y constantes que se usarán en el cálculo de las acciones de control (tales como ganancias o parámetros del robot).
- `UPDATEHOOK()`: Cada vez que se ejecute esta función, se calculan las nuevas acciones de control (para cada iteración), en este caso 3, puesto que se tienen 3 actuadores, y se envía cada una de ellas al módulo actuador correspondiente. Puesto que la entrada al control es la posición actual, en esta parte de código, además, se calcula la posición deseada para la próxima iteración.

Finalmente, el proceso termina en los módulos actuadores, que son los encargados de enviar la acción de control a los actuadores, con la ayuda de la tarjeta D/A (`Pci1720`). Realmente, aunque en la ilustración anterior se aprecian 3 módulos actuadores, realmente son 3 instancias de un único módulo. Este módulo ha sido diseñado de forma que, haciendo tres instancias del mismo, posteriormente en tiempo de ejecución, se pueda seleccionar a qué motor (canal de salida) está asociado, así como el puerto de entrada por el que va a recibir la acción de control. Por ello, aquí se puede apreciar una gran ventaja, puesto que se ahorra una gran cantidad de código, haciendo los módulos reusables (puesto que se pueden instanciar tantas veces como se quiera). Por ejemplo, si se tuviera un robot con 6 motores, no habría más que instanciar 6 veces el módulo actuador, con las ventajas (en cuanto tiempo y fiabilidad) que ello conlleva.

Las funciones más destacadas de este último módulo son las siguientes:

- `CONFIGUREHOOK()`: Mediante esta función se inicializa la tarjeta D/A, así como las variables que luego serán utilizadas.
- `UPDATEHOOK()`: Cuando se llame a esta función (que será de forma iterativa cada vez que le llegue un dato por el puerto de entrada), por el puerto de salida (cada uno para cada instancia) enviará una tensión, acorde con lo recibido por el puerto de entrada.

- `SETCHANNEL(INT NCHANNEL)`: A través de esta función, en tiempo de ejecución (o configuración) y de una forma totalmente interactiva, se consigue asignar a cada una de las instancias del actuador, el motor (puerto de salida) sobre el que va a actuar.

De esta forma, se puede demostrar que con un diseño de módulos acertado, es posible crear una estructura modular que permita la escalabilidad de sensores y actuadores (e incluso controles), así como la reutilización de diversos módulos mediante nuevas instancias de los mismos.

Además, de esta forma es posible asignar prioridades a cada uno de los módulos, dependiendo de nuestras necesidades, hacer un estudio de la temporización de todo el sistema, o conseguir que varios módulos se ejecuten en paralelo (como es el caso de los actuadores, por ejemplo).

3.7.3. Solución modular con supervisor

Como ampliación al diseño modular presentado anteriormente, se ha creado un nuevo módulo llamado *supervisor*. La finalidad de este módulo no es otra más que “supervisar” desde una posición superior o privilegiada, con el objetivo de que todo el sistema funcione correctamente, y que en caso de que se produzca algún error, poder tomar una determinación.

Para la implementación del *supervisor* se ha pensado en un diseño en el que, ante la llegada de un nuevo dato procedente de otros módulos (puesto que los puertos de salida de los otros módulos estarán conectados con él), se ponga en ejecución e identifique de forma automática de qué puerto le ha llegado el dato, así como analizarlo y tratarlo en tiempo real (figura 35).

Cabe destacar la importancia de un módulo que supervise al resto, ya que puede hacer mucho más estable al sistema, en cuanto a fallos se refiere, detectando, entre otros, los siguientes:

- **ERROR EN LA LECTURA DE REFERENCIAS.** En este supuesto, sea el error de la procedencia que sea (mecánico, hardware o software), puede darse el caso de que no se estén recibiendo bien los valores de la posición real del robot. Por ello, el módulo supervisor está programado para determinar el momento en el que un error de posición (diferencia entre el valor real y la referencia) comienza a ser preocupante. Puesto que la lectura de la posición real es un aspecto crítico en el sistema, se ha decidido que, ante esta situación, el módulo supervisor, tras haber hecho todos los cálculos en tiempo real, muestra una advertencia por pantalla e informa al resto de módulos del error. En ese preciso instante, los actuadores reciben mensaje a través del módulo de control y ponen el valor de tensión de la acción de control a 0V, deteniéndose el proceso y quedándose el robot parado.
- **ERROR EN LOS ACTUADORES.** De forma análoga al supuesto anterior, puede darse el caso en el que la tensión de la acción de control esté dando errónea (por ejemplo, porque la tarjeta D/A ha sufrido alguna sobrecarga). En esta ocasión, el módulo supervisor muestra un aviso por pantalla

siendo, además, capaz de configurar una segunda tarjeta D/A conectada al equipo industrial que haga las funciones de la tarjeta errónea, de forma totalmente transparente para el resto de módulos, así como para el usuario y, sobre todo, manteniendo el periodo de muestreo (puesto que los cálculos y algoritmos se realizan en tiempo real). Es decir, teniendo el hardware redundante, si se detecta un posible error, el proceso podría seguir correctamente sin sin ningún problema

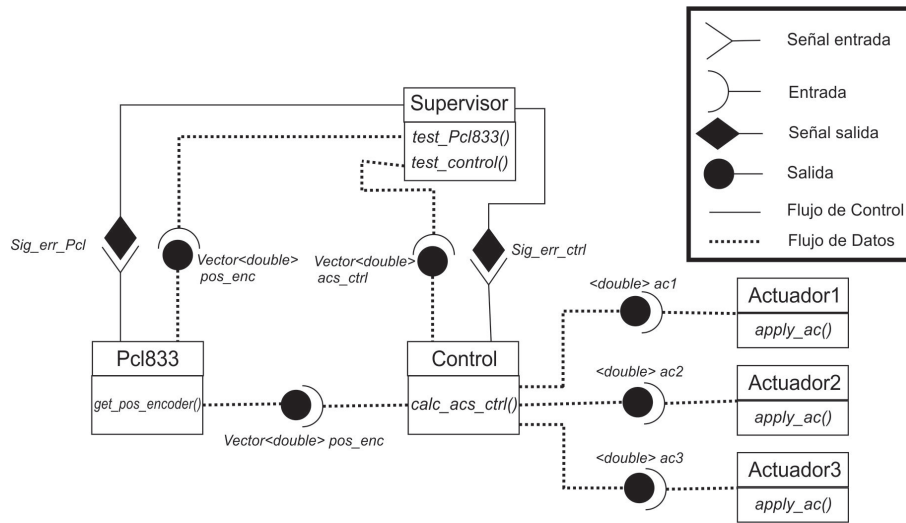


Figura 35: Solución modular con supervisor.

Además de las funcionalidades anteriores, el supervisor se encarga de almacenar el valor del tiempo de ejecución de cada uno de los componentes y el tiempo total de ejecución, así como las posiciones reales del robot y las referencias (con el objetivo de representarlo gráficamente y analizar mejor su comportamiento).

Por ejemplo, se diseñó un control que en las primeras y últimas fases de la trayectoria realizaba un control PID (apartado 3.6.1), mientras que en la parte central de la misma el control era el propuesto por Paden-Panja (apartado 3.6.1). De forma teórica, el coste computacional de un controlador PID debía ser mucho menos que un control de Paden.

Mediante la figura del supervisor, se pudo medir los tiempos de la fase del cálculo de la acción de control, obteniendo la figura 36.

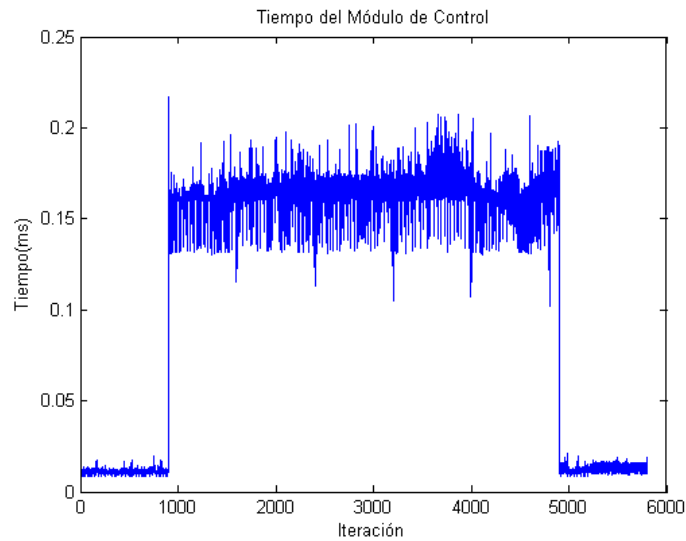


Figura 36: Tiempo de ejecución control

Como se puede observar ver, la suposición teórica se corresponde con la práctica. Concretamente, el control PID se ejecuta en 0.01 ms, mientras que el de Paden se ejecuta con una media de 0.16 ms.

También, haciendo la medición de todo el proceso (desde la toma de datos hasta el envío de la acción de control), tal y como se ve en la figura 37, el tiempo de ejecución está alrededor de los 0.5 ms, siendo un tiempo muy inferior al periodo de muestreo, en este caso de 10 ms.

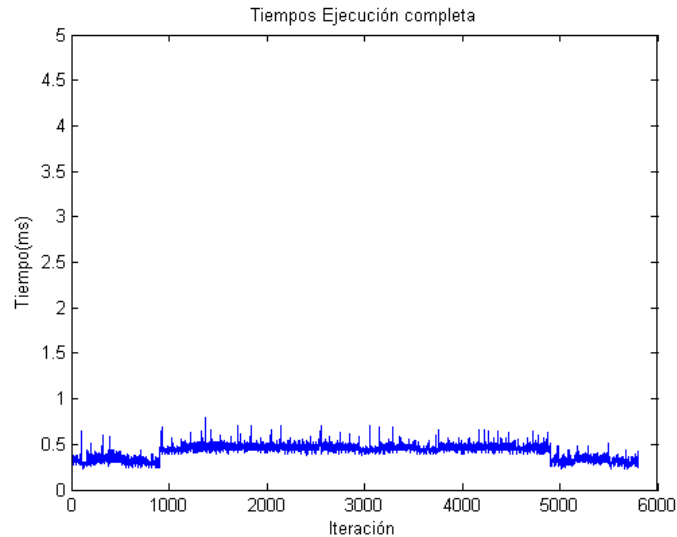


Figura 37: Tiempo de ejecución total

De esta forma, sabiendo el tiempo de cómputo de cada módulo, se pueden contemplar diferentes opciones tales como un control dinámico dependiendo del tiempo restante de CPU, o decrementar el periodo de muestreo para intentar mejorar el control.

Finalmente, cabe señalar que mediante este módulo se consigue apartar a un segundo plano el trío de componentes PCL833-Control-3Actuadores, siendo el supervisor capaz de tomar alguna decisión sobre ellos tanto de forma automática, como también a través de un usuario de forma interactiva (a través de las herramientas de “deployer” que ofrece en middleware Orocos).

3.7.4. Solución modular con generador de trayectorias

En esta ocasión, partiendo de la solución modular con el supervisor, se ha dado un paso más y se ha incluido el módulo *Referencia*, que no hará más que calcular la posición deseada para cada iteración en el algoritmo.

El módulo generador de referencias cobra cierta importancia puesto que el robot seguirá la referencia que indique ese módulo.

Antes de todo, cabe destacar que en este proyecto se han definido todas las trayectorias realizadas dividiéndolas en 3 partes significativas:

- **SPLINE INICIAL.** Es la primera parte del movimiento del robot. Por cuestiones de seguridad, el robot se lleva de la posición inicial ($q_1=q_2=q_3=0$) a la primera posición que se indique en el fichero de referencias mediante una spline cúbica natural. Esta spline se realiza en 8 segundos, por lo que es un movimiento muy suave. Si, por ejemplo, el fichero de referencias indica que la primera posición deseada es $q_1=0.5$, $q_2=0.2$ $q_3=0.4$, si no

se realizara una spline desde la posición inicial del robot a esa posición y se intentara llegar a esa posición (en la siguiente iteración, es decir, en 10 ms) directamente desde la posición inicial, el valor de la acción de control sería desproporcionada y se perdería el control del robot (con el peligro que ello conlleva).

- **MOVIMIENTO CENTRAL.** Una vez el robot ha llegado a la primera posición del fichero de referencias, en cada iteración se va cogiendo una nueva referencia del fichero (que se ha realizado cuidadosamente para coger una cada iteración o 10 ms).
- **SPLINE FINAL.** De la misma forma que se ha hecho una spline desde la posición $q_1=q_2=q_3=0$ a la primera posición que se indique en el fichero de referencias, en esta ocasión se realiza una spline de bajada desde la última posición del fichero de referencias hasta la posición de reposo $q_1=q_2=q_3=0$.

Por ello, es muy importante que la generación de trayectorias sea totalmente independiente del control, como es en este caso. De esta forma, el módulo de control se reduce al cálculo de la acción de control en función de una entrada que recibe de los encóders (que es la posición real del robot) y otra entrada que recibe del módulo *Referencia* que indica la posición deseada (figura 38)

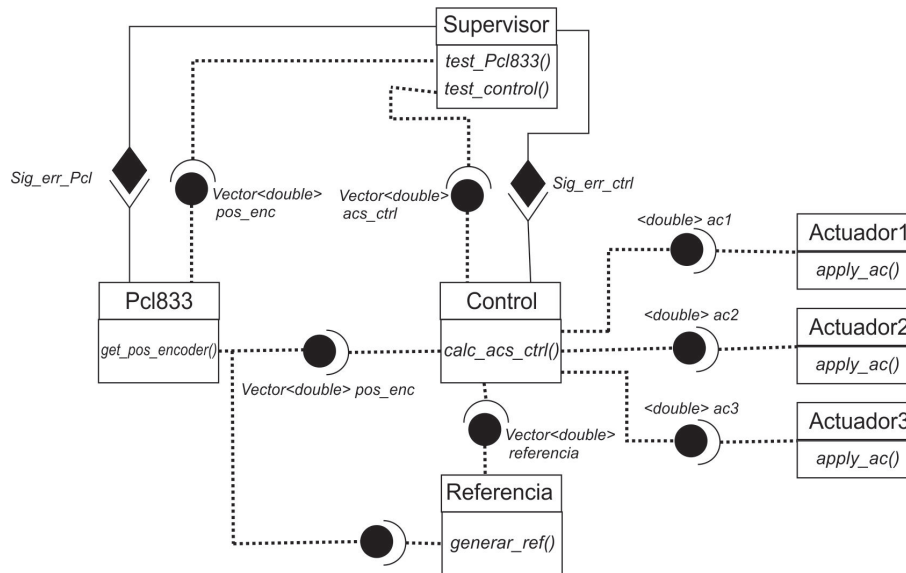


Figura 38: Esquema con generador de referencias

Por otro lado, para conseguir que todo el sistema modular se ejecute de forma correcta, el módulo que recoge la posición real del robot (Pcl833) es el que se ejecuta de forma periódica, concretamente con un periodo de muestreo de 10 ms. Puesto que este módulo se ejecuta de forma periódica, mediante los

distintos tipos de puertos vistos en la sección 3.3.3 se consigue una ejecución en cascada con la ayuda de los puertos de evento y los puertos de evento asociados a función.

Un claro uso del puerto de evento es cuando le llega la acción de control a aplicar a cada módulo encargado de suministrar la tensión a los actuadores. En esta ocasión, en cuanto les llega algo por el puerto de entrada, pasan a ejecutar la función *UpdateHook()*, ejecutándose en paralelo los 3 módulos actuadores.

Otro uso de un puerto de evento asociado a función son los dos puertos de entrada del módulo de control. Puesto que la ejecución es en cascada, si varios módulos se están ejecutando en paralelo, en función del planificador unos pueden terminar su ejecución antes que otros. En el caso del módulo de control, hasta que no tenga tanto la posición real como la posición deseada, no puede pasar a ejecutarse. Habrá ocasiones en las que le llegará antes la posición deseada, y viceversa. Para ello, cada puerto de entrada está asociado a una función que, en pseudocódigo serían las siguientes.

```
Funcion EntradaEncoders()  
{si datosReferencia==recibidos entonces (ejecutar UpdateHook())  
sino (datosEncoders==recibidos)}
```

```
Funcion EntradaReferencia()  
{si datosEncoders==recibidos entonces (ejecutar UpdateHook())  
sino (datosReferencia==recibidos)}
```

De esta forma tan sencilla se consigue que un módulo sólo se ejecute cuando tenga los necesarios para ejecutar su propio algoritmo.

Finalmente, otro aspecto destacable es que, mediante este esquema modular, si se quisiera implementar otro tipo de controlador, sólo sería necesario que tuviera 2 puertos de entrada (posición real y deseada) y 3 puertos de salida (uno para cada actuador), por lo que cambiar de un controlador a otro sería totalmente trivial, además de transparente para el resto de módulos en la estructura.

3.7.5. Inclusión del módulo de fuerza

Uno de los últimos esquemas modulares que se han implementado de manera exitosa ha sido el que se puede ver en la figura 39. El principal cambio que se puede apreciar es la inclusión de un módulo de fuerza (usando el sensor de fuerza comentado en la sección 3.1.4).

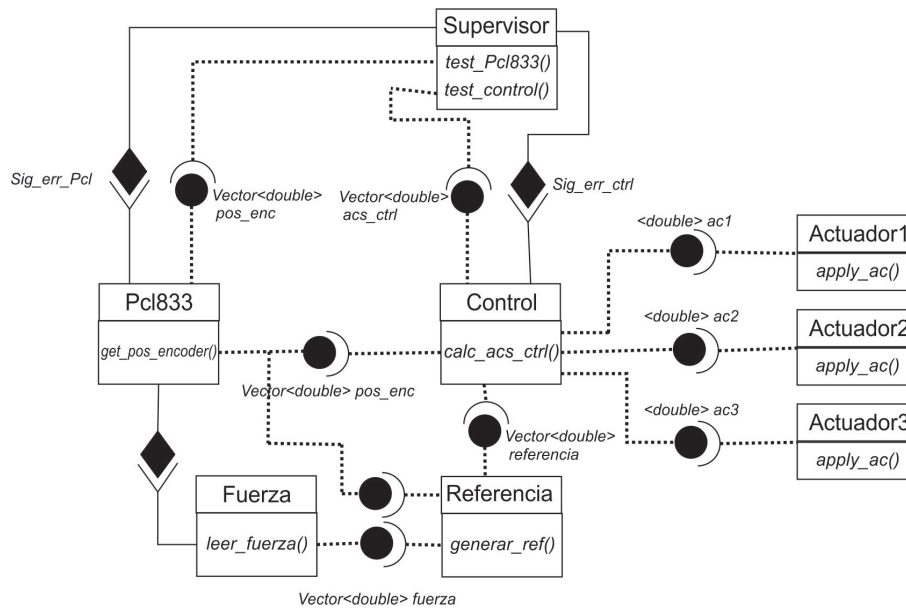


Figura 39: Esquema con el módulo de fuerza

Como se puede ver, gracias a la estructura modular, los cambios en el esquema son mínimos, habiendo reusado prácticamente la totalidad del código ya implementado.

En esta ocasión, al generador de referencias le llega por su puerto de entrada la lectura del sensor de fuerza. De esta forma, en vez de seguir una referencia de posición (como hasta ahora), se ha conseguido que el robot consiga seguir una referencia determinada en función del valor que se obtenga del sensor de fuerza (como se verá en los resultados experimentales).

3.7.6. División en submódulos del módulo de control

Pese a que los esquemas modulares de las anteriores secciones se comportan de manera excelente, se ha decidido dividir el módulo de control en pequeños módulos, con el objetivo de reutilizarlos para otros tipos de controles.

Por ejemplo, partiendo del esquema en *Matlab-Simulink* de un control PD+G como el que se muestra en la siguiente figura, se ha propuesto una división en módulos para implementarlos mediante el entorno Orocós.

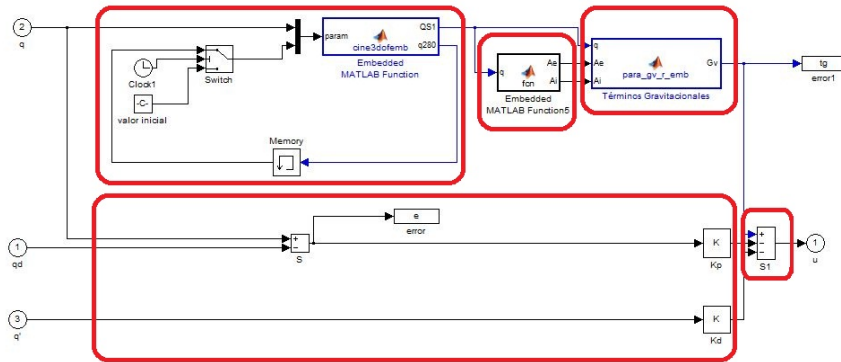


Figura 40: Esquema con el módulo de fuerza

Tras haber dividido el módulo de control de la figura 40 en submódulos, se obtiene el siguiente esquema.

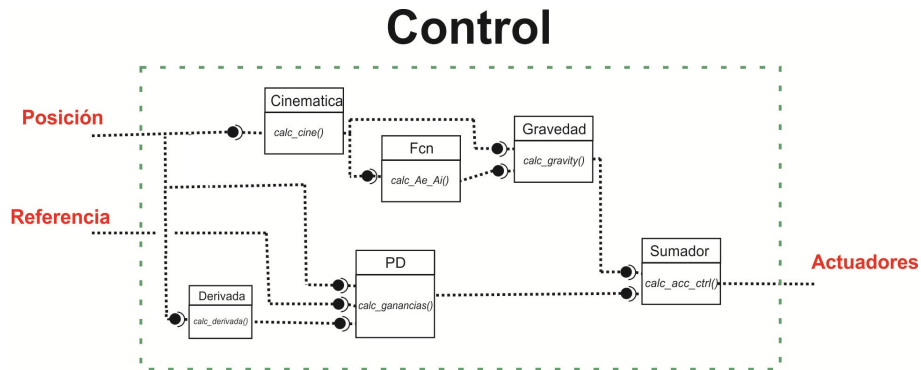


Figura 41: Esquema con el módulo de fuerza

De esta forma y tal y como se ha comentado anteriormente, puesto que el módulo que se encarga de la lectura de sensores (*Pcl833*) es que lleva la marca de tiempo, el resto de módulos se van ejecutando en cascada mediante los puertos de eventos (y asociados a módulos) explicados en apartados anteriores.

Por tanto, sólo hay que preocuparse de que el módulo *Pcl833* se ejecute de forma periódica, puesto que eso desencadenará una ejecución en cascada. Además, como se puede observar, a medida que un módulo va recibiendo por los puertos de entrada los valores necesarios para realizar su función, comienza a ejecutarse, por lo que varios módulos se ejecutan de forma distribuida (como es el caso de módulo *PD* y *Cinématica*), con todas las ventajas que ello conlleva.

Finalmente, puesto que los términos cinemáticos y gravitacionales se usan en los controladores pasivos, como el de Paden, sería muy sencillo de implementar otros controladores reusando los módulos ya implementados.

4. Resultados experimentales

Este apartado tiene como objetivo presentar algunos de los resultados más relevantes que se han obtenido al trabajar en este proyecto con el robot paralelo 3PRS.

En primer lugar se comentarán los aspectos relacionados con el seguimiento de una trayectoria de posición, y posteriormente se abordará el tema del seguimiento de trayectorias de fuerza.

4.1. Seguimiento de trayectoria de posición

Para comprobar los distintos controladores implementados, se llevaron a cabo varias ejecuciones con trayectorias distintas. El objetivo de esta sección es presentar los resultados obtenidos para un controlador PID, uno de Paden-Panja y un adaptativo.

4.1.1. Controlador PID

En primer lugar, se implementó un controlador PID, puesto que es el más básico y sencillo de implementar (apartado 3.6.1). De la misma manera, puesto que no se consideran parámetros dinámicos (sólo puramente cinemáticos) no se esperaba una respuesta demasiado buena.

En las siguientes tres figuras se puede observar, para cada articulación, la posición deseada (en azul) y la posición real del robot.

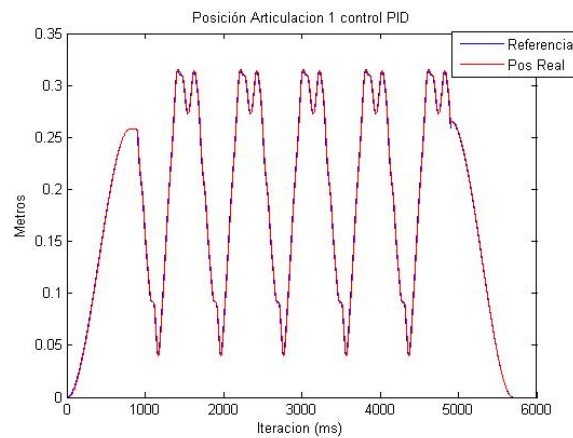


Figura 42: Articulación 1 Control PID

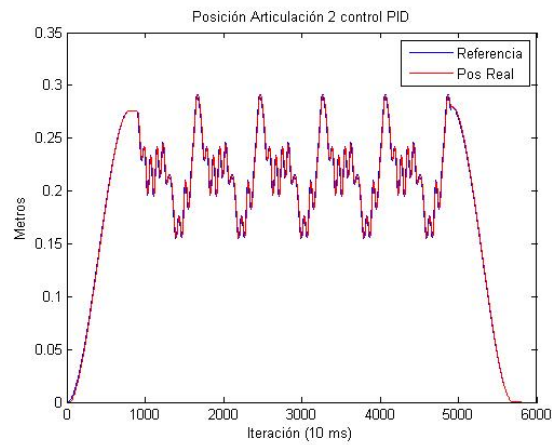


Figura 43: Articulación 2 Control PID

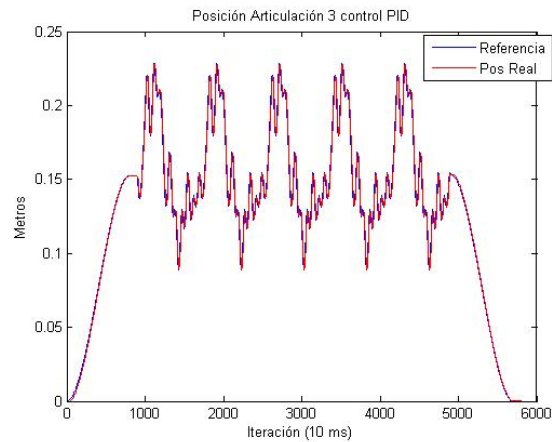


Figura 44: Articulación 3 Control PID

Como se puede apreciar en las gráficas anteriores, el resultado es realmente bueno pese a no haber considerado ningún parámetro de la dinámica del sistema. Cabe destacar que el resultado fue mejor que lo estimado en el esquema simulado (mediante *Matlab-Simulink*) pero, si se amplia la imagen, se puede apreciar el típico desfase de señal que se produce típicamente en un controlador PID.

4.1.2. Controlador Paden-Panja

Las siguientes pruebas realizadas se hicieron con el controlador de Paden-Panja, explicado en el apartado 3.6.1. Puesto que es un controlador que considera

la dinámica, los resultados esperados, según la simulación, deben ser mejores que los experimentados con el PID. A continuación, en las tres gráficas siguientes, se puede ver, para cada articulación, la posición real del robot y la referencia que debe seguir el mismo.

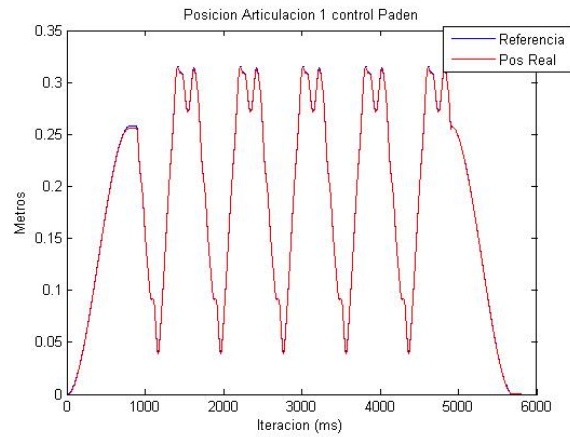


Figura 45: Articulación 1 Control Paden-Panja

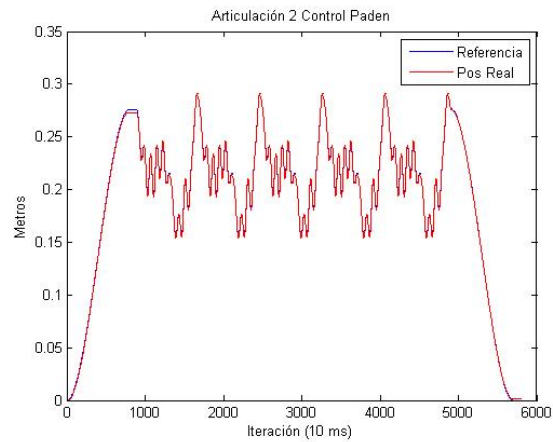


Figura 46: Articulación 2 Control Paden-Panja

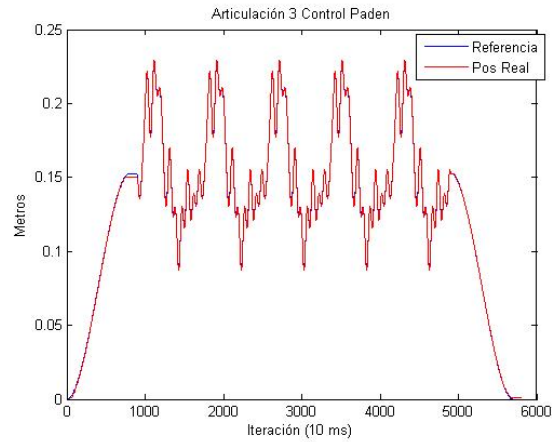


Figura 47: Articulación 3 Control Paden-Panja

Como se puede advertir, las líneas de posición real y deseada están, prácticamente, superpuestas, por lo que se está siguiendo la trayectoria con error mínimo. El resultado es el esperado, puesto que en las simulaciones que se realizaron previamente con el modelo del sistema, ya se observó que el comportamiento era mucho mejor que un PID, realizando una aproximación excelente.

4.1.3. Comparativa PID vs Paden-Panja

Para las pruebas realizadas con los controladores PID y de Paden, se ha usado la misma trayectoria de posición, con el objetivo de realizar una comparativa entre los errores que se producen para cada uno de los controladores.

Pese a que se observa a simple vista que el PID tiene un error mucho mayor al controlador de Paden, en la siguiente gráfica se puede ver el error (para la articulación 1) para la misma trayectoria.

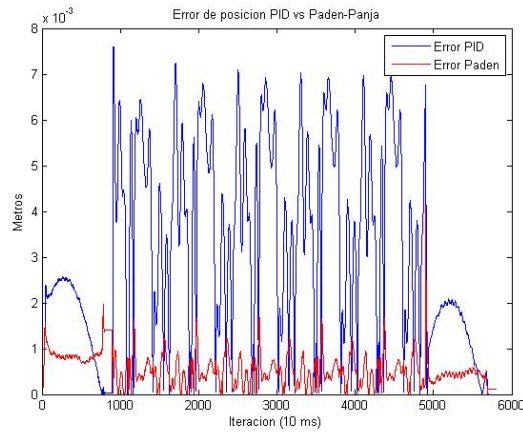


Figura 48: Comparativa control PID vs Paden-Panja

Como se puede ver en la gráfica anterior, mientras que el error para el controlador de Paden se sitúa por debajo de 1 mm, en el caso del PID (color azul) es mucho mayor, llegando a picos de hasta 7 mm.

De esta forma, se puede evidenciar cómo el uso de la dinámica del sistema para un proceso de este tipo, es realmente importante para la implementación de un controlador.

4.1.4. Control Adaptativo

Finalmente, una de las últimas pruebas realizadas fue la comparativa entre el error de posición de un controlador de Paden y un adaptativo. Para poder realizar bien esta prueba, en un determinado instante de la trayectoria se debía cargar el robot con una masa determinada (por ejemplo, 100 Kg). Puesto que era peligroso cargar con una masa de esas características a mitad de ejecución, lo que se hizo fue modificar los parámetros dinámicos del robot con el objetivo de simular que se había puesto sobre la plataforma una masa de 100 Kg.

Como se puede ver en la siguiente gráfica, se tiene por un lado (izquierda) la posición de la primera articulación para un control de Paden y uno adaptativo mientras que en la figura de la derecha se puede ver el error (en valor absoluto) para cada uno de los controladores.

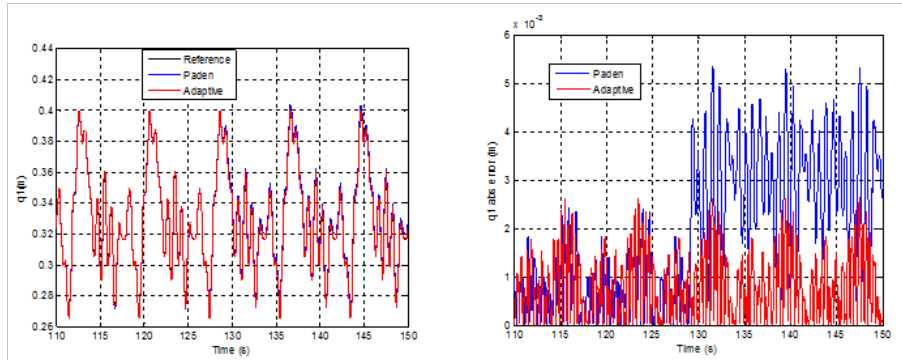


Figura 49: Error Paden y Adaptativo

En el experimento realizado, a los 13 segundos de ejecución se cambiaron los parámetros del robot, simulando una masa de 100 Kg sobre la plataforma del robot. Como se puede ver en la gráfica anterior, mientras que el error en el controlador de Paden aumenta significativamente, el control adaptativo identifica rápidamente los nuevos parámetros, manteniendo el error de posición en un valor bastante pequeño.

En la figura 50 se pueden ver los errores del primer periodo (sin masa adicional sobre la plataforma) y del segundo periodo (con una masa de 100 Kg) para cada uno de los dos controladores. Como se ha visto de forma gráfica, en el segundo periodo hay una gran diferencia entre el error de posición que hay para un controlador de Paden y un adaptativo.

Errores	$\frac{\sum_{j=1}^{n \text{ DOF}} \sum_{i=1}^{n \text{ DOF}} \text{abs}(e_{i,j})}{n \cdot \text{DOF}}$		$\sqrt{\frac{\sum_{j=1}^{n \text{ DOF}} \sum_{i=1}^{n \text{ DOF}} (e_{i,j})^2}{n \cdot \text{DOF}}}$	
	Adaptativo	<u>Paden</u>	Adaptativo	<u>Paden</u>
1º periodo	0.000681	0.000615	0.000871	0.000813
2º periodo	0.000687	0.002809	0.000868	0.002923

Figura 50: Errores cuadráticos adaptativo

4.2. Seguimiento de trayectoria de fuerza

Uno de los últimos ensayos que se han realizado con el robot paralelo 3PRS ha sido el seguimiento de una referencia de fuerza sobre el eje Z. De esta forma, estando el sensor de fuerza como se muestra en la figura 51, el objetivo era seguir una determinada trayectoria (en este caso fue una senoidal) de fuerza.



Figura 51: Robot 3PRS con sensor de fuerza

Puesto que la senoidal era sobre el eje Z, la plataforma del robot siempre debía permanecer paralela al suelo (y para cada una de las articulaciones del robot, la referencia debía ser la misma).

En la siguiente gráfica se puede ver la referencia (en rojo), y la fuerza real que se estaba aplicando con el robot.

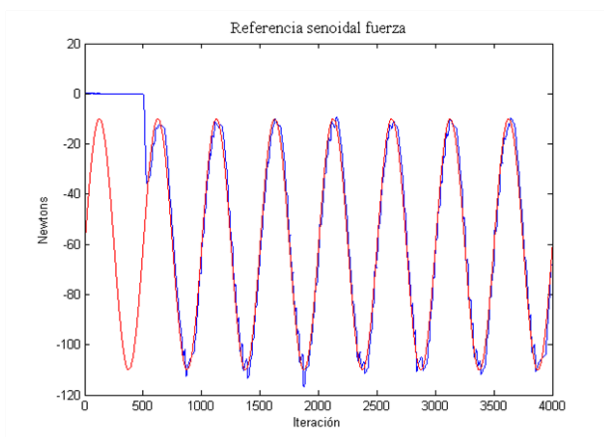


Figura 52: Seguimiento trayectoria de fuerza

Como se puede apreciar, el resultado es realmente bueno, puesto que se está haciendo una aproximación con un error muy pequeño, siendo capaces de seguir (en este caso) una trayectoria senoidal, aunque en pruebas posteriores (con datos proporcionados por el Instituto de Biomecánica de Valencia) se ha conseguido simular un movimiento de *marcha* incluso con la misma velocidad que en la realidad, obteniendo resultados muy satisfactorios.

5. Conclusiones y trabajos futuros

Mediante la realización de este proyecto, se ha conseguido llegar a implementar una estructura modular con el objetivo de realizar el control de un robot PRS de tres grados de libertad.

En un primer momento, antes de implementar nada utilizando Orocos, se estuvo estudiando la manera de integrar el hardware usado en el experimento. Puesto que no se usó un sistema operativo comercial (como Windows), la obtención de los drivers fue una tarea ardua. Para la tarjeta de D/A, mediante la librería *Comedi*, se pudo integrar sin mayor problema pero, por el contrario, para la tarjeta de lectura de encoders A/D no existía ningún controlador, por lo que se tuvo que implementar completamente. Por lo que al sensor de fuerza se refiere, puesto que la entrada al PC industrial era mediante una conexión de red RJ-45, se creó un *socket UDP* en C++, con el que se puede realizar la lectura sin ningún problema.

A continuación, tras haber integrado el hardware y comprendido los aspectos más importantes del desarrollo de software basado en componentes mediante el estudio de diferentes artículos de investigación, se procedió a implementar en un único módulo todo el proceso de control (lectura de sensores - cálculo de acción de control - aplicación de la acción de control). A pesar de que los resultados fueron buenos de inicio, se quiso ir más allá hasta el punto de realizar un esquema totalmente modular (en las últimas fases de este proyecto), dividiendo el módulo de control en pequeños submódulos, de forma que se pudieran reutilizar para la implementación de varios controladores.

Básicamente, los principales objetivos conseguidos y las mayores ventajas encontradas fueron los siguientes:

- Mediante un diseño modular como el propuesto, se permite un rápido seguimiento del flujo de ejecución del programa, facilitando la creación de nuevos componentes con el fin de ir insertándolos en el modelo, y así conseguir nuevas funcionalidades.
- Como se puede intuir del punto anterior, esta estructura modular permite la ejecución de manera distribuida del código de varios módulos (además, de los módulos cuyo tiempo de ejecución es mayor, tales como el cálculo de las componentes inerciales, de Coriolis o gravitacionales), obteniendo un tiempo de ejecución menor que si se realizara en serie.
- La reusabilidad del código es también muy importante, ya que, como se ha visto, un módulo puede ser instanciado un número indeterminado de veces, lo que es muy útil para el programador del control. En este caso, si el robot tuviera dos motores más, con instanciar dos veces más el componente *Actuador* sería suficiente.
- Puesto que todo el proceso se divide en módulos, al haber verificado que uno de ellos funciona correctamente en un esquema, con toda seguridad ese módulo funcionará correctamente en otro esquema. Este aspecto es muy

importante para el programador, puesto que se asegura que un módulo que ya ha sido verificado puede volver a ser usado con el convencimiento de que su funcionamiento será correcto, eliminando el riesgo de cometer un error de implementación.

- También, el hecho de poder configurar y reconfigurar un módulo en tiempo de ejecución es una característica verdaderamente importante ya que, además de poder cambiar los parámetros de control que se deseen (como, por ejemplo, el periodo de muestreo o las ganancias), se permite modificar las prioridades de forma dinámica (más allá de la política de planificación) de cualquier módulo.

Por lo que al coste se refiere, bien es cierto que en una fase inicial el coste es mayor, puesto que hay que adaptarse a la nueva forma de desarrollar el software. Por el contrario, cuando se ha cogido soltura desarrollando módulos y conectándolos entre sí para permitir un flujo de datos, ese coste inicial se reduce de forma muy significativa.

Como uno de los objetivos futuros se ha propuesto mejorar la funcionalidad del supervisor y que realice funciones tales como tener un poder de decisión para poder permutar correctamente de un controlador a otro en función del tiempo de ejecución disponible. También una función muy importante del supervisor es la relacionada con la seguridad, por lo que hay que mejorar este aspecto para un posible fallo en cualquier punto del sistema.

Por lo que respecta al esquema modular, sería interesante en un futuro cercano implementar distintas variantes del control del proceso con la ayuda de los módulos anteriormente creados, así como la nueva creación de módulos con distintas funcionalidades.

También, es uno de los objetivos la implementación de otros controladores no lineales dinámicos, tanto en el espacio cartesiano como en el espacio de cada una de las articulaciones

Debido al buen funcionamiento de todo lo expuesto en este proyecto, se han iniciado otras líneas de investigación, tales como la biomecánica. Concretamente, en la actualidad se está trabajando con un grupo de investigadores del IBV (Instituto de Biomecánica de Valencia) para la calibración dinámica de una plataforma de fuerzas *kinescan* con la ayuda del robot paralelo y el sensor de fuerza comentado en apartados anteriores (figura 53).

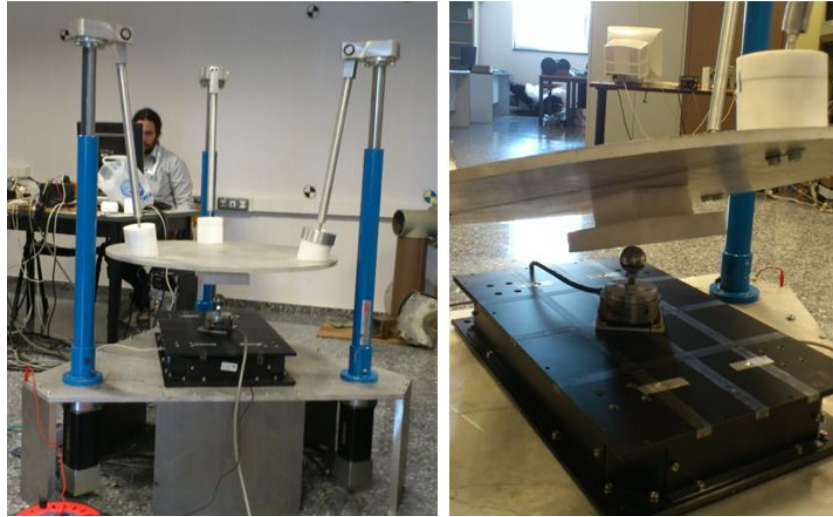


Figura 53: Calibración dinámica de la plataforma

Finalmente, la última línea de investigación (la cual va avanzando rápidamente, aunque aún se está en una fase muy temprana de la investigación) es la referente a la rehabilitación de tobillo y rodilla (también en conjunto con el IBV) con la ayuda del robot paralelo, el sensor de fuerza y las cámaras proporcionadas por el IBV, en la que se están obteniendo resultados realmente interesantes, habiendo realizado pruebas experimentales con personas reales (figura 54).

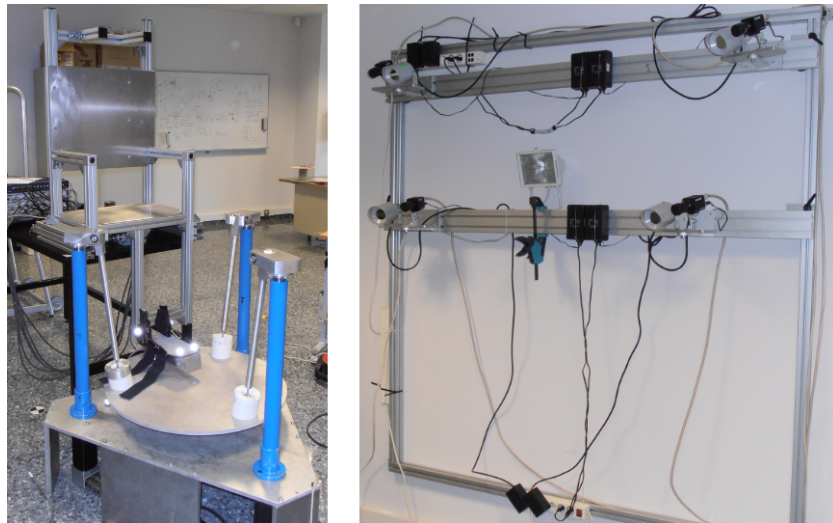


Figura 54: Sistema de rehabilitación

Referencias

- [1] D. Alonso, and B. Álvarez J.A. Pastor, and P. Sánchez, and C. Vicente-Chicote. Generación automática de software para sistemas de tiempo real: Un enfoque basado en componentes, modelos y frameworks. *Revista Iberoamericana de Automática e Informática Industrial*, 9(2):170–181, 2011.
- [2] Desarrollo basado en componentes. <http://www.iuma.ulpgc.es/users/lhdez/inves/tesis/memoria-tesis/node2.html>.
- [3] H. Bruyninckx. Open robot control software: the orocos project. In *In IEEE International Conference on Robotics and Automation (ICRA '01)*, vol. 3, pp. 2523–2528, 2001.
- [4] E. Burdet, B. Sprenger, and A. Cordourey. Experiments in nonlinear adaptive control. In *in Proceedings of the IEEE International Conference on Robotics and Automation*, 1997.
- [5] A. Campbell, G. Coulson, and M. Kounavis. Managing complexity: Middleware explained. *IEEE Computer Society*, 1999.
- [6] Modelos Cinemáticos. <http://www.monografias.com/trabajos16/estacion-robotica/estacion-robotica.shtml>.
- [7] P. Clements and M. Shaw. The golden age of software architecture revisited. *Software, IEEE*, 26(4):70–72, july-aug. 2009.
- [8] A. Codourey. Dynamic modeling of parallel robots for computed-torque control implementation. *The International Journal of Robotics Research*, 17:1325–1336, 1998.
- [9] M. Díaz-Rodríguez, V. Mata, N. Farhat, and S. Provenzano. Identifiability of the dynamic parameters of a class of parallel robots in the presence of measurement noise and modeling discrepancy. *Mechanics Based Design of Structures and Machines*, 36:478–498, 2008.
- [10] M. Díaz-Rodríguez, V. Mata, A. Valera, and A. Page. A methodology for dynamic parameters identification of 3-dof parallel robots in terms of relevant parameters. *Mechanism and Machine Theory*, 45:1337–1356, 2010.
- [11] DeviceNet. <http://es.wikipedia.org/wiki/devicenet>.
- [12] Modelos Dinámicos. <http://proton.ucting.udg.mx/materias/robotica/r166/r99/r99.htm>.
- [13] H. B. Gou, Y. G. Liu, G. R. Liu, and H. R. Li. Cascade control of a hydraulically driven 6-dof parallel robot manipulator based on a sliding mode. *Control Engineering Practice*, 16:105–168, 2009.
- [14] Y. Li and Q. Xu. Design and development of a medical parallel robot for cardiopulmonary resuscitation. *IEEE/ASME Tr*, 12:265–273, 2007.

- [15] Middleware. <http://es.wikipedia.org/wiki/middleware>.
- [16] N. Andreff P. Poignet F. Pierrot O. Company P. Renaud, A. Vivas. Kinematic and dynamic identification of parallel mechanisms. *Control Engineering Practice*, 14:1099–1109, 2006.
- [17] B. Paden and R. Panja. Globally asymptotically stable pd+ controller for robot manipulators. *Int. J. on Control*, 47:1697–1712, 1988.
- [18] Steward. A platform with 6 degree of freedom. In *Proceedings of the Institution of mechanical engineers*, 1965.
- [19] J.F. Tang, L.F. Mu, C.K. Kwong, and X.G. Luo. An optimization model for software component selection under multiple applications development. *European Journal of Operational Research*, 212(2):301 – 311, 2011.
- [20] H. Utz, S. Sablatnög, S. Enderle, and G. Kraetzchmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics an*, 18:493–497, 2002.
- [21] N. Farhat A. Valera V. Mata, F. Benimeli. Dynamic parameter identification in industrial robots considering physical feasibility. *Advanced Robotics*, 19:101–120, 2005.
- [22] T. Wijayasiriwardhane, R. Lai, and K.C. Kang. Effort estimation of component-based software development - a survey. *IET Software*, 5(2):216 – 228, 2011.
- [23] Y.-X. Zhang, S. Cong, W.-W. Shang, Z.-X. Li, and J. S. Long. Modeling, identification and control of a redundant planar 2-dof parallel manipulator. *International Journal of Control, Automation, and Systems*, 5:559–569, 2007.

A. Instalación de Orocos ToolChain

Como se comentó a lo largo de la memoria, la herramienta de Orocos que se ha utilizado ha sido la Orocos ToolChain, que es un paquete en el que vienen incluidos la Real Time ToolKit y la Orocos Component Library. Mediante estos dos paquetes se conseguirán crear numerosos componentes, ayudándonos de los ya creados, y del script “OrocreatePkg” (que nos ayudará para crear la estructura básica del mismo).

Por lo que respecta a la instalación de la Orocos ToolChain, en este proyecto se instaló la versión 2.5.0, siendo la última versión estable. Es importante ir actualizando las versiones a medida que se vayan desarrollando, puesto que de una versión a otra se corrigen un buen número de *bugs* por parte de los desarrolladores.

En un primer paso, hay que ir a la página oficial del Orocos ToolChain (<http://www.orocos.org/toolchain>). Se tiene la opción de instalarlo mediante la red, aunque es recomendable descargarse los fuentes, para luego compilarlos desde el propio sistema.

Una vez se tienen descargados los fuentes de la Orocos ToolChain, se deben extraer en una carpeta del sistema de archivos local.

En este momento, dentro de la carpeta donde se ha extraído, hay que ejecutar las siguientes órdenes:

```
./bootstrap_toolchain
source env.sh
```

Con la primera orden se ejecuta un script, el cual comprueba que estén instalados todos los paquetes necesarios para la instalación de Orocos (como el paquete Ruby, por ejemplo). En caso de no estar instalado, lo indica y termina la instalación. Si en la instalación nos falta algún paquete, mediante el *Gestor de paquetes de Synaptic* que nos ofrece Ubuntu, se puede instalar cualquier paquete sin ninguna dificultad. Una vez termina la ejecución del script, el sistema está preparado para instalarle la ToolChain.

Es importante realizar la segunda orden cada vez que arranque el sistema, ya que incluye todos los PATH necesarios para la ejecución de algún componente de Orocos. En este caso, se optó por incluirla dentro del archivo de configuración “.bashrc”, de forma que cada vez que se inicia Ubuntu, se lanza esa orden de forma completamente transparente al usuario.

Posteriormente, se tiene que ejecutar la orden siguiente:

```
autoprojbuidl-reconfigure
```

Este momento es crítico para el correcto funcionamiento de los componentes, puesto que hay que estar muy atento a la configuración y especificarle que el RTT (“Real Time Target”) es Xenomai (y no gnulinux, como aparece por defecto). Esta parte crítica de la instalación se puede apreciar en la siguiente captura.

```
Archivo  Editar  Ver  Terminal  Ayuda
Advanced users may want to control this behaviour. Additionally, the
installation of some packages require administration rights, which you may
not have. This option is meant to allow you to control autoproj's behaviour
while handling OS dependencies.

* if you say "all", it will install all packages automatically.
  This requires root access thru 'sudo'
* if you say "ruby", only the Ruby packages will be installed.
  Installing these packages does not require root access.
* if you say "os", only the OS-provided packages will be installed.
  Installing these packages requires root access.
* if you say "none", autoproj will not do anything related to the
  OS dependencies.

As any configuration value, the mode can be changed anytime by calling
an autoproj operation with the --reconfigure option (e.g. autoproj update
--reconfigure).

Finally, OS dependencies can be installed by calling "autoproj osdeps"
with the corresponding option (--all, --ruby, --os or --none).

So, what do you want ? (all, ruby, os or none) [all]
autoproj: loading ...
the target operating system for Orocos/RTT (gnulinux or xenomai) [gnulinux] xenomai
```

Figura 55: Instalación Orocos

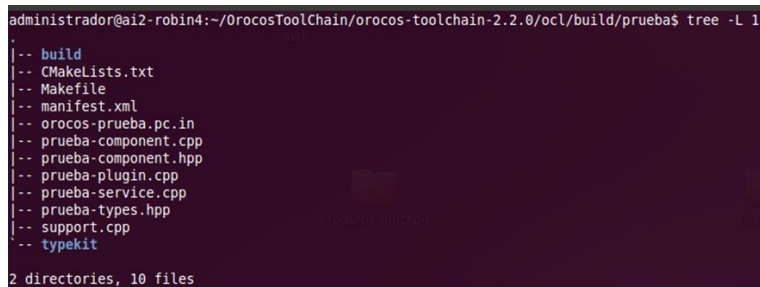
Finalmente, y si no se ha encontrado ningún tipo de problema, ya se tiene instalado en el directorio correspondiente la Orocos ToolChain. Al final de la instalación se puede comprobar que se han instalado los módulos “utilm”, “rtt”, “ocl”, “typelib” y “orogen” de forma satisfactoria.

B. Creación y funcionamiento de componentes

Para la creación de un nuevo componente en Orocos, se puede hacer directamente de forma manual (realmente es un fichero en C++) o mediante una herramienta que viene proporcionada en el OrocosToolChain, llamada *OrocreatePkg*, que no es nada más que un script, el cual genera toda la estructura de ficheros fuente y directorios, listos para compilar y ser ejecutados. La utilización es tan sencilla como teclear lo siguiente:

```
orocreate-pkg nombreDelComponente
```

Una vez se ha creado un componente, se tiene la estructura que muestra la figura siguiente:



```
administrador@a12-robin4:~/OrocosToolChain/orocos-toolchain-2.2.0/ocl/build/prueba$ tree -L 1
.
|-- build
|-- CMakeLists.txt
|-- Makefile
|-- manifest.xml
|-- orocos-prueba.pc.in
|-- prueba-component.cpp
|-- prueba-component.hpp
|-- prueba-plugin.cpp
|-- prueba-service.cpp
|-- prueba-types.hpp
|-- support.cpp
|-- typekit
2 directories, 10 files
```

Figura 56: Estructura ficheros componente

Dentro de esta estructura, hay ficheros que son importantes conocer, de cara a poder modificarlos acorde a nuestras necesidades. Estos ficheros son:

- **NOMBREDELCOMPONENTE-COMPONENT.HPP**: Este fichero es el que contiene el código fuente y estructura mínima que debe tener un componente en Orocos. Será en este fichero en el que se irán añadiendo las funciones que se vayan necesitando, así como implementar los métodos predefinidos.
- **CMAKELISTS.TXT**: Este fichero no se debe modificar, puesto que contiene toda la información necesaria para la creación del Makefile (y así poder compilar el componente de forma automática). Para la generación del Makefile se usarán los programas *ccmake* y *cmake*.

Una vez se tiene la estructura creada, hay que proceder a compilarlo, por lo que se han realizado una serie de pasos.

- En la carpeta que nos ha generado *OrocreatePkg* se crea una carpeta, que se puede llamar “build” y se entra en ella.

```
mkdir build
cd build
```

- Dentro de ese directorio, mediante la herramienta *ccmake* (que permite configurar el fichero “CMakeLists.txt” mediante una interfaz muchísimo más sencilla) y *cmake* (que teniendo un fichero de configuración, generado por *ccmake*, es capaz de generar un Makefile) se consigue crear un Makefile con el que se puede compilar el componente. Tras teclear la siguiente orden, aparecerá la siguiente interfaz de configuración.

```
ccmake ..
```

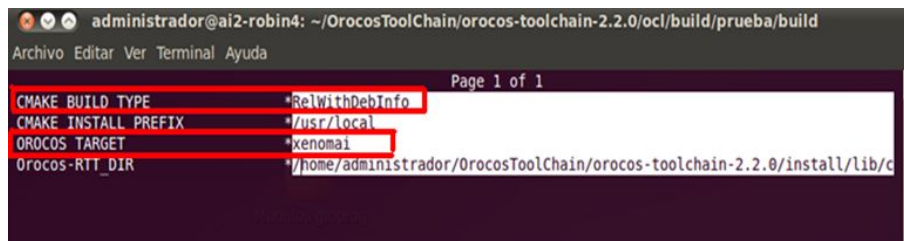


Figura 57: Ccmake

- Es muy importante en la configuración mediante el *ccmake* introducir los parámetros correctos. En este caso, como el *Real Time Target* es Xenomai, habrá que indicarlo de forma oportuna en la configuración, así como el modo de compilación, en vez de ser *Release* (que viene por defecto), en este caso se pondrá *RelWithDebInfo*. Tras ir configurando las diversas opciones, dentro de la interfaz aparece la opción de generar esa configuración, por lo que genera el archivo definitivo.
- Finalmente, cuando se ha creado el fichero final, ayudándonos de la herramienta *cmake* se consigue crear el definitivo fichero Makefile (para poder compilar). Se deberán de teclear las siguientes órdenes, obteniendo un resultado similar al de la siguiente imagen.

```
cmake ..
```

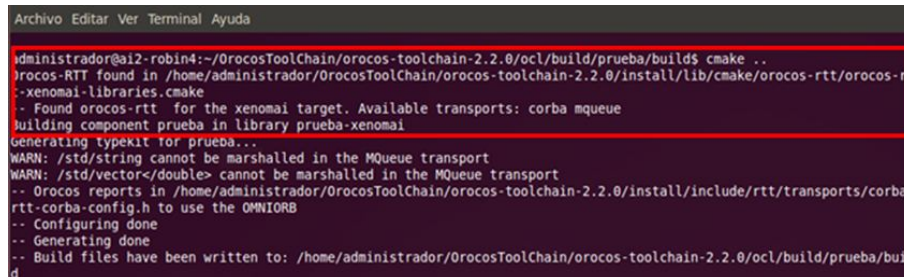


Figura 58: Cmake

- Como paso definitivo, sólo queda lanzar el comando *make* y comprobar cómo se generan los archivos con extensión *.so*. Uno de los cuales será el componente que se ha creado (*libNombreDelComponente-xenomai.so*).

```

administrador@a12-robin4:~/OrocosToolChain/orocos-toolchain-2.2.0/ocl/build/prueba/build$ make
Scanning dependencies of target prueba
[ 5%] Building CXX object CMakeFiles/prueba.dir/prueba-component.cpp.o
Linking CXX shared library libprueba-xenomai.so
[ 5%] Built target prueba
Scanning dependencies of target check-typekit-uptodate
[ 11%] Built target check-typekit-uptodate
Scanning dependencies of target prueba-typekit-xenomai
[ 16%] Building CXX object typekit/CMakeFiles/prueba-typekit-xenomai.dir/Plugin.cpp.o
[ 22%] Building CXX object typekit/CMakeFiles/prueba-typekit-xenomai.dir/type_info/PruebaData_std_string.cpp.o
[ 27%] Building CXX object typekit/CMakeFiles/prueba-typekit-xenomai.dir/type_info/_std_vector_double_.cpp.o
Linking CXX shared library libprueba-typekit-xenomai.so
[ 27%] Built target prueba-typekit-xenomai
Scanning dependencies of target prueba-transport-typelib-xenomai
[ 33%] Building CXX object typekit/transportstypelib/CMakeFiles/prueba-transport-typelib-xenomai.dir/PruebaData_std_string.cpp.o
[ 38%] Building CXX object typekit/transportstypelib/CMakeFiles/prueba-transport-typelib-xenomai.dir/_std_vector_double_.cpp.o
[ 44%] Building CXX object typekit/transportstypelib/CMakeFiles/prueba-transport-typelib-xenomai.dir/TypelibMarshalerBase.cpp.o
[ 50%] Building CXX object typekit/transportstypelib/CMakeFiles/prueba-transport-typelib-xenomai.dir/TransportPlugin.cpp.o
Linking CXX shared library libprueba-transport-typelib-xenomai.so
[ 50%] Built target prueba-transport-typelib-xenomai
[ 55%] Generating pruebaTypesC.cpp, pruebaTypesDynSK.cpp
Scanning dependencies of target prueba-transport-corba-xenomai
[ 61%] Building CXX object typekit/transportscorba/CMakeFiles/prueba-transport-corba-xenomai.dir/Conversions.cpp.o
[ 66%] Building CXX object typekit/transportscorba/CMakeFiles/prueba-transport-corba-xenomai.dir/TransportPlugin.cpp.o
[ 72%] Building CXX object typekit/transportscorba/CMakeFiles/prueba-transport-corba-xenomai.dir/PruebaData_std_string.cpp.o
[ 77%] Building CXX object typekit/transportscorba/CMakeFiles/prueba-transport-corba-xenomai.dir/_std_vector_double_.cpp.o
[ 83%] Building CXX object typekit/transportscorba/CMakeFiles/prueba-transport-corba-xenomai.dir/pruebaTypesC.cpp.o
[ 88%] Building CXX object typekit/transportscorba/CMakeFiles/prueba-transport-corba-xenomai.dir/pruebaTypesDynSK.cpp.o
Linking CXX shared library libprueba-transport-corba-xenomai.so
[ 88%] Built target prueba-transport-corba-xenomai
Scanning dependencies of target prueba-transport-mqueue-xenomai
[ 94%] Building CXX object typekit/transportsmqueue/CMakeFiles/prueba-transport-mqueue-xenomai.dir/TransportPlugin.cpp.o
[100%] Building CXX object typekit/transportsmqueue/CMakeFiles/prueba-transport-mqueue-xenomai.dir/PruebaData.cpp.o
Linking CXX shared library libprueba-transport-mqueue-xenomai.so
[100%] Built target prueba-transport-mqueue-xenomai

```

Figura 59: Cmake

Finalmente, no queda más que iniciar el deployer (tecleando *deployer-xenomai*) y comenzar a cargar y configurar los módulos previamente implementados tal y como se especifica en la API de programación.