
# UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

Coordinación de Manipuladores en Entornos Dinámicos.
Modelo de Programación para Laboratorios
Virtuales y Remotos.

**Dña. Almudena Ruiz Sáez**

**2017**


# UNIVERSIDAD DE MURCIA

## FACULTAD DE INFORMÁTICA

Coordinación de Manipuladores en Entornos Dinámicos.
Modelo de Programación para Laboratorios Virtuales y Remotos.

*Tesis Doctoral:*
Doctorado en Informática

*Autor:*
Almudena Ruiz Sáez

*Directores:*
Humberto Martínez Barberá
Francisco Esquembre Martínez

Murcia, 2017

UNIVERSIDAD DE
MURCIA

D. Humberto Martínez Barberá, Profesor Titular de Universidad del Área de Ingeniería Telemática en el Departamento de Ingeniería de la Información y las Comunicaciones, y D. Francisco Esquembre Martínez, Catedrático de Universidad en el Departamento de Matemáticas,

AUTORIZAN:

La presentación de la Tesis Doctoral titulada "Coordinación de manipuladores en entornos dinámicos. Modelo de programación para laboratorios virtuales y remotos", realizada por Dª. Almudena Ruiz Sáez, bajo nuestra inmediata dirección y supervisión, y que presenta para la obtención del grado de Doctora por la Universidad de Murcia.

En Murcia, a 19 de Mayo de 2017

Fdo.: Humberto Martínez Barberá y Francisco Esquembre Martínez

*A mis padres Pedro y Carmen.*

*A mi hermana Carmen María,*

*y a Julio.*

# *Agradecimientos*

Deseo hacer constar mi más sincero agradecimiento, en primer lugar, a mis directores de tesis, Francisco Esquembre Martínez y Humberto Martínez Barberá. A Paco, por ser mi mentor durante todos estos años, por su apoyo incondicional, por su amistad y por mostrarme que las Matemáticas van más allá de un simple teorema y demostración. A Humberto, por brindarme la oportunidad de formar parte de su grupo de investigación apenas sin conocerme, por la confianza depositada en mí y por transmitirme su pasión por la Robótica.

Así mismo, me gustaría destacar a dos profesores del Departamento de Matemáticas, Víctor Jiménez López y Antonio Linero Bas, porque gracias a ellos empecé a conocer el increíble mundo de las Matemáticas Aplicadas y mi pasión por la programación.

A mi familia, amigos y compañeros que a lo largo de todos estos años han estado a mi lado y que de una forma u otra han contribuido para que este sueño se haga realidad.

A mi abuelo, por demostrarme que en la vida no existen límites ni obstáculos que no se puedan superar, que simplemente hay que confiar en uno mismo y luchar por tus sueños. Por enseñarme que el único secreto para lograr todo esto es el esfuerzo diario, sin olvidar nunca que si tú quieres, tú puedes.

Por supuesto, a mis padres y mi hermana, los pilares de mi vida, porque sin ellos no habría sido capaz de llegar hasta aquí. Por la confianza que siempre han depositado en mí, por hacer mis sueños los suyos propios, por su incondicional apoyo, por no dejarme nunca caer en el camino y siempre estar ahí.

Y, por último a Julio. Gracias por creer en mí, por apoyarme día a día durante todos estos años juntos y por ser el otro pilar fundamental en mi vida. Gracias por compartir conmigo este largo camino, por tu paciencia y comprensión.

Gracias a todos.

# Índice general

# Índice de figuras

# Índice de tablas

# Índice de códigos

# Resumen

La Robótica se ha convertido en una herramienta indispensable en diferentes áreas como son la industria, la ingeniería, la educación o la investigación.

Dentro de cada una de estas áreas, la coordinación de diferentes robots tanto en entornos estáticos como dinámicos es una de las principales cuestiones que se debe analizar.

La coordinación de robots consiste en mucho más que evitar las posibles colisiones que puedan existir entre ellos. Los usuarios desean una sincronización más precisa para que los robots puedan trabajar en equipo y completar tareas que un solo robot no podría llevar a cabo.

Evidentemente, la utilización de múltiples robots que comparten un mismo espacio de trabajo puede aumentar la productividad e incrementar la versatilidad de las aplicaciones a tareas mucho más complejas. Como contrapartida, cuando más de un robot se mueve en un espacio de trabajo común cada uno de ellos se transforma en un obstáculo móvil para los demás.

Una forma idónea de entender y estudiar el comportamiento complejo de estos robots es mediante la creación de laboratorios virtuales y remotos.

Uno de los principales problemas que el usuario se encuentra a la hora de diseñar estas simulaciones robóticas es que cada robot usa un lenguaje propio de programación. Este hecho, junto a las peculiaridades propias de cada robot, complica la implementación de operaciones comunes y reducen la posibilidad de reutilizar código.

En esta tesis se ha modelado, conceptualizado y diseñado una nueva librería, desarrollada en Java, para la coordinación de robots manipuladores tanto en entornos estáticos como dinámicos. Esta librería establece un HAL (hardware abstraction layer) y facilita una API (application programming interface) que incluye todas las operaciones básicas que se requieren a la hora de trabajar con robots manipuladores: posicionamiento, planificación de trayectorias, definición de restricciones, detección de colisiones, visualización 3D, y comunicación con el robot real. Además, ofrece una súper clase *RoboticsLab* que permite coordinar fácilmente los movimientos de todos los elementos de un laboratorio robótico evitando las colisiones que pudiesen existir entre ellos.

Esta API ha sido implementada para dos robots manipuladores particulares: el robot TX60L de Stäubli y el robot Scara de Omron, y ha integrado otros elementos que pueden ser básicos a la hora de configurar un laboratorio robótico, como, por ejemplo, una cinta transportadora. Esta

lista de elementos puede ser fácilmente extendida a otros robots o elementos robóticos ya que estas implementaciones se han llevado a cabo de una manera transparente al usuario y fácilmente reutilizable.

Es una librería fácil de usar, de código abierto y compatible en la mayor parte de los sistemas (todos aquellos que soporten Java). Todo este conjunto de características hacen que los programadores de Java sean capaces de crear laboratorios virtuales y remotos de alto nivel de una forma cómoda, sencilla y reutilizable.

Por otro lado, esta librería ha sido embebida en el software Easy Java Simulations (EJS). Esta implementación se ha llevado a cabo mediante la definición de nuevos elementos del modelo de EJS que incorporan todas las operaciones y propiedades de la nueva API.

La ventaja fundamental de esta implementación es permitir tanto a los programadores como no programadores de Java la posibilidad de crear laboratorios robóticos virtuales y remotos sin necesidad de tener altos conocimientos ni de Java ni de Robótica.

En conclusión, esta nueva librería presentada en esta tesis es una herramienta muy útil en diversos ámbitos: en el campo de la educación, ya que permite aprender los conceptos robóticos de una manera sencilla, en el campo de la investigación, porque permite a los investigadores representar situaciones robóticas complejas para su estudio, y en la ingeniería, como base para la creación de prototipos de plantas industriales que estén compuestas por robots manipuladores.

Finalmente, con el fin de mostrar todas las ventajas de esta nueva librería y las propiedades de su uso, se facilitan ejemplos de distinta complejidad de laboratorios virtuales y remotos, compuestos por robots manipuladores, que han sido diseñados directamente en Java o apoyándose en el entorno EJS. También se incorporan los detalles de las implementaciones de los elementos que han sido integrados en la librería para que puedan servir como guía a la hora de incorporar nuevos elementos.

# Abstract

Robotics has become an indispensable tool in different areas such as industry, engineering, education or research.

In each of these areas, the coordination of different robots both in static and dynamic environments is one of the main questions to be analyzed.

The coordination of robots is much more than just avoiding possible collisions among them. Users desire a perfect synchronization among the robots so that they can work as a team and complete tasks that a single robot would be unable to do by itself.

Evidently, the use of multiple robots that share the same workspace could improve the productivity and increase the versatility of the applications to more difficult tasks. However, when more than one robot is moving in a common workspace, each of them becomes an obstacle for the others.

A good way to understand and study the complex behavior of these robots is to create virtual and remote laboratories.

One of the main problems the user faces when designing these robotic simulations is that each robot has its own programming language. This fact, together with the own peculiarities of each robot, makes difficult the implementation of common operations and reduces the possibility to reuse the code.

In this thesis, a new library has been modeled, conceptualized and designed, developed in Java, for the coordination of manipulator robots in both statics and dynamics environments. This library sets a HAL (hardware abstraction layer) and facilitates an API (application programming interface) that includes all basic operations required when working with manipulator robots: positioning, path planning, definition of restrictions, collision detection, 3D visualization and communication with the real robots. Furthermore, this library offers a super class *RoboticsLab*, that allows to easily coordinate the movements of all the elements of the robotic lab and so to avoid the collisions that might take place.

This API has been implemented for two particular manipulator robots: the TX60L from Stäubli and the Scara from Omron. At the same time, it has integrated other elements that will be essential at the time of configuring the robotic lab; for example, a conveyor belt. This list of elements can be easily extended to other robots or robotic elements, since such implementations have been done

in a transparent way for the user and can be easily reused.

Our library is user friendly, open source, and compatible with the majority of operating systems (all that support Java). All these features allow Java programmers to create virtual and remote labs with ease at a high level, in a simple and reusable way.

Furthermore, this library has been embedded in Easy Java Simulations (EJS) authoring tool. Such implementation has taken place by defining new model elements for EJS that incorporate all the operations and properties of the new API.

The main advantage of such implementation is that it allows Java programmers and non-programmers to create virtual and remote robotic labs without the need of having high skills of either Java or Robotics.

In conclusion, the new library presented in this thesis is a very useful tool in various fields: in education, since it allows the learning of Robotic concepts in a simple way; in the research field, because it allows researchers to represent complex robotic situations for their study; in engineering, practice as basis for the creation of prototypes of industrial plants that are composed of manipulator robots.

Finally, to show the advantages of this library and its properties, examples of varied complexity of virtual and remote labs are provided. These labs are made up by manipulator robots that have been designed directly in Java or in the EJS environment. We also provide instances where elements that have been integrated in this library are implemented, so that they might be used as a guide when new elements need to be incorporated.

# Parte I

# Presentación y Estado del Arte

# Capítulo 1

# Introducción

En este capítulo se presenta la tesis doctoral, se explica su motivación, se lleva a cabo su contextualización dentro del campo, se detallan las aportaciones obtenidas y, finalmente, se facilita su estructura.

## 1.1. Motivación de la tesis

No hay duda de la importancia que, durante los últimos años, ha adquirido la Robótica dentro del mundo de la investigación y la industria. Actualmente, la tecnología robótica se encuentra muy presente en todos los ámbitos de nuestra sociedad, convirtiéndose a día de hoy en una herramienta indispensable para nuestro crecimiento sostenible.

Los sistemas robóticos pueden presentar comportamientos muy complejos a la hora de trabajar con ellos. Una de las maneras más útil y segura para experimentar y estudiar dichos comportamientos sin presentar ningún riesgo, antes de utilizarlos en el lugar de trabajo, es mediante la creación de programas de ordenador que nos permitan simularlos y coordinarlos. Es decir, mediante la creación de *laboratorios robóticos virtuales*. Estos laboratorios permiten a los investigadores y estudiantes analizar y comprender las bases de la Robótica, además de poder representar diferentes situaciones reales de una manera más simple, intuitiva y económica.

Los laboratorios virtuales, sin embargo, no pueden reemplazar el estudio de un sistema robótico real en su totalidad, ya que no pueden modelar todas las características físicas del hardware o todas las situaciones inesperadas que pueden ocurrir en la realidad. Por esta razón, tras una fase inicial de estudio basada en laboratorios virtuales, en ocasiones surge la necesidad de usar un equipo real, ya

sea presente en el mismo laboratorio o, a menudo, localizado en instalaciones remotas. La conexión de un laboratorio virtual previamente diseñado con el equipo real, situado en otra localización, lo convierte en un *laboratorio remoto*, donde los usuarios (investigadores o estudiantes) pueden combinar la interacción entre la simulación del ordenador y el equipo real, observando los resultados del robot real al ejecutar las operaciones descritas.

Basándose en los avances de las nuevas tecnologías, los laboratorios remotos permiten a los usuarios conectarse con un sistema real mediante Internet y poder analizar sus comportamientos desde cualquier lugar. Es decir, los laboratorios remotos pueden definirse como un punto intermedio entre el proceso puramente simulado y su aplicación industrial final, obteniendo grandes beneficios para el usuario. Entre estos beneficios se pueden destacar: minimizar costes, incrementar la seguridad o facilitar el acceso al equipo real [2].

Tanto los laboratorios virtuales como los remotos son una herramienta indispensable para el prototipado de plantas industriales. En una fase inicial de un proyecto, cuando se quiere determinar qué componentes son los más apropiados para llevar a cabo las tareas requeridas o qué problemas pueden aparecer en el desarrollo de las mismas, el uso de un prototipo puede ser la clave para ayudar a analizar la implementación de la planta y facilitar el diseño final de la misma, ahorrando al usuario tanto tiempo como dinero.

Sin embargo, a la hora de trabajar con estos tipos de laboratorios robóticos el usuario se puede encontrar con algunos inconvenientes. En primer lugar, cuando se quiere trabajar con el robot real hay que tener en cuenta que cada robot posee su propio lenguaje de programación. El nivel de complejidad de estos lenguajes de programación varía según el tipo de robot, lo cual hace que resulte difícil crear programas que operen con varios tipos de robots al mismo tiempo [3]. En segundo lugar, pueden existir problemas de implementación, principalmente cuando se implementa la cinemática directa e inversa del robot, ya que la complejidad de éstas depende directamente de la configuración del propio robot. Finalmente, cuando el laboratorio está compuesto por varios robots, una tarea importante es el control de sus movimientos para llevar a cabo tareas en coordinación, además de evitar las posibles colisiones entre ellos. Resulta imprescindible que el movimiento de los robots esté sincronizado de forma que puedan trabajar en equipo y realizar tareas que un sólo robot no podría ejecutar. La coordinación de múltiples robots en un mismo espacio de trabajo, estático o dinámico, permite realizar aplicaciones mucho más complejas aumentando la productividad y versatilidad del laboratorio.

Considerando la importancia de estos laboratorios en la actualidad, la principal motivación con la que se inicia esta tesis es diseñar una herramienta que permita al usuario la creación de laboratorios robóticos virtuales y remotos de alto nivel de una forma cómoda y sencilla tanto en el ámbito de la educación e investigación como en el del prototipado de plantas industriales.

El trabajo presentado en esta tesis se centra en resolver los problemas que aparecen cuando se quiere trabajar con este tipo de laboratorios, analizando todos sus aspectos, especialmente aquellos

que combinan diferentes tipos de robots manipuladores. Para este fin, se desarrollará una capa de abstracción del hadware (HAL) cuyo objetivo es proporcionar un marco para la aplicación y el desarrollo con componentes predefinidos además de una interfaz de programación de aplicaciones (API) común para todos los robots y componentes robóticos integrados en el laboratorio.

En estos aspectos generales son en los que se sitúa esta tesis doctoral. Cabe destacar que esta tesis no sólo facilita una nueva herramienta para la coordinación de robots manipuladores y el diseño de laboratorios robóticos virtuales y remotos, sino que se ha implementado para robots y componentes concretos con los que se han desarrollado diferentes laboratorios de ejemplo, que muestran las principales características de esta nueva herramienta. Además, otra motivación destacable de esta tesis es la integración de esta nueva librería en el software Easy Java Simulations. Como resultado de esto, ambos, programadores y no programadores, serán capaces de usar esta nueva herramienta para desarrollar laboratorios de robótica virtuales y remotos de una manera sencilla y reutilizable, sin requerir altos conocimientos en Robótica.

## 1.2. Objetivos de la tesis

El trabajo presentado en esta tesis está enfocado al modelado, conceptualización, y desarrollo de una librería Java de alto nivel que sirva como base para el estudio y coordinación de sistemas robóticos mediante el diseño de laboratorios virtuales y remotos.

Este objetivo está inspirado en trabajos previamente desarrollados por otros autores. Desde el punto de vista de la Robótica, esta biblioteca está inspirada en la librería *Easy Java Simulations Robotics Library* (EjsRL) [4]. EjsRL es una biblioteca gráfica de alto nivel basada en Java 3D que proporciona un completo marco para modelar, desarrollar y ejecutar aplicaciones avanzadas de Robótica y visión por ordenador (ver Sección 2.3). Esta librería ha sido diseñada específicamente para trabajar con el software de programación EJS [5], de manera que las operaciones de los robots son demasiado dependientes de su arquitectura. Este hecho hace que resulte complicado trabajar con más de un robot en una misma simulación.

Esta tesis se va a centrar en la coordinación de sistemas robóticos tanto en entornos estáticos como dinámicos. Para ello, se va a conceptualizar el trabajo con los robots, prescrito e implementado en una API que recoge e incrementa las capacidades de la biblioteca original facilitando el uso de un número ilimitado de robots, incluso de diferentes tipos. Además, el cálculo de las trayectorias es ahora independiente de la simulación, y se han introducido nuevas características, tales como la definición de restricciones, la detección de colisiones, la coordinación de los diferentes movimientos y la visualización 3D basada en objetos VRML (Virtual Reality Modeling Language).

La visualización 3D de la librería está basada en el proyecto Open Source Physics (OSP) [6, 7], el cual es una sinergia de desarrollos curriculares, física computacional, e investigación en educación de la Física, que utiliza Java como plataforma de software preferida y que proporciona varios entornos

de Java gratuitos de código abierto para el desarrollo de simulaciones de ciencia e ingeniería (ver Sección 2.5). Esta librería está perfectamente coordinada con el entorno de visualización OSP 3D (ver Subsección 4.3.5), el cuál permite visualizar nuestros robots en tres dimensiones, haciendo uso de Java3D (además del uso de otros entornos de la librería OSP, como framework de animación o de cálculo numérico).

Aunque el trabajo presentado en esta tesis se centra en el uso independiente de la nueva librería por los programadores de Java, un segundo e igualmente objetivo importante de esta tesis es la integración de esta nueva librería en el software EJS, también parte del proyecto OSP. EJS es un software gratuito desarrollado en Java cuya finalidad es ayudar a usuarios no programadores a crear simulaciones interactivas en Java o JavaScript, principalmente para el ámbito de la educación e investigación [8] (ver Sección 2.6).

## 1.3. Contribuciones

Partiendo de la motivación inicial con la que se inició esta tesis, a continuación se detallan las diferentes contribuciones que esta tesis aporta en el campo de la Robótica:

- Se ha llevado a cabo la conceptualización de qué es un robot manipulador, cuáles son sus características principales, funcionalidad y problemas que pueden ofrecer cuando se trabaja con ellos. De forma análoga, se ha conceptualizado el resto de componentes robóticos que pueden formar parte del laboratorio junto a los robots, como, por ejemplo, las cintas transportadoras.

- Basándonos en estas conceptualizaciones se ha establecido un HAL común para todos los robots manipuladores y componentes robóticas cuya finalidad es facilitar al usuario una librería robótica para la creación de laboratorios robóticos compuestos por diferentes elementos.

- Este HAL ha sido desarrollado en Java dando lugar a una API universal. Esta API ha sido implementada completamente para algunos robots manipuladores y componentes robóticos concretos, como es el caso de una cinta transportadora.

- La API ofrece una súper clase *RoboticsLab()* que permite coordinar el movimiento de los diferentes elementos del laboratorio robótico, tanto en entornos estáticos como dinámicos, evitando las posibles colisiones entre ellos.

- Diferentes protocolos de comunicación han sido implementados en esta nueva librería para el diseño de laboratorios remotos.

- Esta nueva librería de Java para Robótica ha sido embebida como una librería de modelado en el software EJS dando lugar a nuevos elementos del modelo.

- Finalmente, se aportan diferentes ejemplos de laboratorios virtuales y remotos desarrollados con esta nueva librería tanto en Java como en EJS. Dichos ejemplos muestran que esta librería requiere una curva de aprendizaje muy pequeña por parte del usuario.

Estas contribuciones se han visto realizadas mediante la especificicación y posterior desarrollo de un HAL (con su API universal) que conceptualiza el funcionamiento de un robot, e implementa completamente una librería robótica, incluyendo la definición y el funcionamiento de algunos robots manipuladores en concreto, así como de algunos componentes básicos a la hora de configurar dichos laboratorios.

Esta nueva librería ha sido desarrollada en Java, es libre, fácil de usar y de código abierto, lo que permite su uso gratuito y su extensión a nuevos robots manipuladores o componentes robóticos. Las clases que la componen contienen diferentes métodos, comunes a todos los robots manipuladores y componentes, lo cuál hace que esta librería se pueda extender a otros robots y elementos de una forma muy sencilla.

La principal ventaja de esta nueva librería es que permite la coordinación de diferentes robots tanto en entornos estáticos como dinámicos y se centra en facilitar la creación de laboratorios robóticos tanto virtuales como remotos, y en el desarrollo de prototipos de plantas industriales con diferentes componentes: robots, cintas transportadoras, sensores,... Se trata de una herramienta sencilla, orientada a poder agregar fácilmente uno o varios robots en un mismo laboratorio y operar con los robots con facilidad (tanto virtual como de forma remota). El uso de esta librería no requiere un gran esfuerzo por parte del usuario, sea programador o no, ni un alto conocimiento de Robótica.

En resumen, esta nueva librería de Java desarrollada para robots manipuladores presenta tres características principales: La primera de ellas es la definición de una API de Java, común a una gran variedad de robots manipuladores, que permite coordinar sus movimientos y que contiene todas las operaciones estándares con este tipo de robots. La API estandariza el posicionamiento de un robot, la definición de sus trayectorias, la introducción de restricciones de movimiento - es decir, el movimiento a lo largo de una trayectoria se realiza bajo una serie de comandos establecidos que hacen que el movimiento se interrumpa automáticamente si alguna de estas restricciones son violadas - y la resolución de autocolisiones y detección de colisiones con otros robots o con el propio entorno que lo rodea. Además, esta API también incluye la visualización del robot en un escenario virtual 3D y la comunicación remota con el robot real.

La segunda característica principal de la librería es la implementación de esta API para dos robots manipuladores en concreto: el robot TX60L de 6 GDL de Stäubli [9], y el robot Scara de 4 GDL de Omron [10], así como la implementación de otros elementos básicos, como una cinta transportadora. La implementación de los detalles particulares de estos elementos (como sus propiedades geométricas, sus lenguajes propietarios, o su visualización 3D, por ejemplo) han sido embebidas en la API de una manera transparente al usuario (ver Sección 4.1).

La tercera característica es la integración de esta librería en el software EJS como elementos del modelo. Esta integración permite a los usuarios diseñar laboratorios robóticos sin necesidad de tener altos conocimientos ni de Java ni de Robótica.

En definitiva, el uso de esta API de Java y la librería en conjunto proporciona a los programadores la capacidad de coordinar robots manipuladores y crear laboratorios sofisticados compuestos por estos robots y los componentes integrados, de una forma sencilla y reutilizable.

## 1.4. Estructura de la tesis

Esta tesis doctoral está dividida en cuatro partes que componen un total de siete capítulos y un apéndice. A continuación se detalla brevemente el contenido de cada uno de ellos:

- *Parte I: Presentación y Estado del Arte.* Esta parte está compuesta por dos capítulos y su finalidad es hacer una introducción de la tesis y un análisis del estado del arte.
  - *Capítulo 1: Introducción.* En este capítulo se facilita la contextualización de la tesis, describiendo la motivación de la misma, sus objetivos y aportaciones llevadas a cabo.
  - *Capítulo 2: Estado del arte.* En este capítulo se introduce el problema de la coordinación de robots, se presenta el concepto de laboratorio virtual y remoto y se lleva a cabo una revisión detallada de los más importantes. Además, se introduce el proyecto Open Source Physics y el software Easy Java Simulations, elementos fundamentales en el desarrollo de esta tesis.
- *Parte II: Investigación y Diseño.* Esta parte está estructurada en cuatro capítulos y en ellos se describe tanto la investigación como el diseño de la tesis.
  - *Capítulo 3: Conceptualización de los elementos de un laboratorio virtual y remoto.* En este capítulo se analiza cómo se puede llevar a cabo la coordinación de diferentes robots, se conceptualizan los diferentes elementos que integran un laboratorio robótico, robots manipuladores y componentes robóticos, y se plantean los diferentes problemas que pueden surgir a la hora de trabajar con ellos.
  - *Capítulo 4: Diseño de una nueva API para laboratorios virtuales y remotos.* En este capítulo se lleva a cabo la definición de un nuevo HAL y diseño de una API en Java para la creación de laboratorios virtuales y remotos que facilite la coordinación de robots en entornos estáticos y dinámicos.
  - *Capítulo 5: Integración de la API en el software EJS.* En este capítulo se lleva a cabo la integración de la nueva API de Java, presentada en el capítulo anterior, en el software EJS.

- *Capítulo 6: Creación de laboratorios virtuales y remotos de Robótica.* En este capítulo se presentan diferentes ejemplos de laboratorios virtuales y remotos desarrollados haciendo uso de esta nueva librería de Java.

- *Parte III: Conclusiones.* En esta parte, constituida por un único capítulo, se describen y analizan los resultados obtenidos así como los posibles trabajos futuros.

  - *Capítulo 7: Conclusiones y Trabajos futuros.* En este capítulo se muestran las conclusiones de esta tesis tras analizar las aportaciones obtenidas, se plantean posibles trabajos futuros y se detallan las publicaciones obtenidas.

- Parte IV: Apéndices. Esta última parte está compuesta por una única sección donde se aportan los detalles complementarios del desarrollo de esta tesis.

  - *Apéndice A: Implementación de los elementos de la nueva librería.* En esta sección se facilitan los detalles de las implementaciones llevadas a cabo durante el desarrollo de la tesis.

    El software desarrollado en esta tesis y los ejemplos están disponibles en la siguiente dirección web: `http://www.um.es/fem/publications/2017/Robotics`.

# Capítulo 2

# Estado del arte

En este capítulo se analiza el problema de la coordinación de robots manipuladores, se introduce el concepto de laboratorio virtual y remoto y se hace un análisis de las principales herramientas que existen hoy en día. Además, se detalla el proyecto Open Source Physics y el software de simulación EJS.

## 2.1. Introducción a la coordinación de robots manipuladores

En los últimos años, los robots se han convertido en una pieza indispensable en todos los procesos industriales. La incorporación del robot dentro de estos procesos ha supuesto uno de los avances más importantes de la edad moderna. Se ha pasado de aquellos primeros modelos, simples y limitados, a sofisticados sistemas capaces de sustituir al hombre en numerosos tipos de tareas repetitivas o peligrosas.

Los robots aportan diversos beneficios tales como reducción de costos, incremento de la productividad, mejora de la calidad final del producto y reducción de problemas en ambientes peligrosos al ser humano como, por ejemplo, ambientes radioactivos.

A pesar de todo, la operación con un único robot en un entorno de trabajo limita el tipo de tareas que pueden ser ejecutadas, de forma que la utilización de varios robots operando de forma coordinada se hace indispensable para llevar a cabo trabajos más complejos como transportar una carga de gran tamaño. Por otra parte, el uso de más de un robot también puede mejorar la eficiencia y versatilidad de las tareas. Así, la tarea puede ser descompuesta en varias subtareas más simples, que pueden ser realizadas en paralelo por cada uno de los robots [11].

Sin embargo, cuando se utilizan varios robots en un mismo entorno de trabajo resulta imprescindible coordinar sus movimientos, ya que cada uno de sus movimientos se convierte en un obstáculo móvil para los demás.

La coordinación de múltiples robots es uno de las principales problemas que surgen dentro del campo de la Robótica. En la literatura se pueden encontrar diversos trabajos enfocados a la resolución de dicho problema. Algunos de ellos se detallan a continuación:

- En [12] se describe el desarrollo de un nuevo controlador de robots, el IRC5 de ABB en el cual se ha integrado la función *MultiMove* que permite trabajar en perfecta coordinación con hasta cuatro robots. *MultiMove* es totalmente flexible por su capacidad para conmutar el funcionamiento de los robots. El principio en que se basa este controlador es una ampliación del principio de coordinación de un robot con un posicionador de trabajo, una tecnología más antigua, capaz sólo de coordinar dos robots u otros dispositivos. Con MultiMove, el dispositivo de trabajo, que puede ser un robot o un posicionador de trabajo, controla el objeto sobre el que se opera. Los otros dispositivos se mueven coordinadamente respecto del objeto de trabajo.

- En [13] se analiza y se propone una solución al problema de coordinar múltiples robots que desarrollan tareas programadas de manera independiente en el mismo espacio de trabajo. Este método eficiente restablece en tiempo real la pérdida de la coordinación debido a la incertidumbre temporal en la ejecución de la tareas. La coordinación se restablece a partir de la replanificación de los perfiles de velocidad sin alterar el camino geométrico, introduciendo los retardos necesarios en una forma controlada y sin establecer prioridades jerárquicas entre ellos.

- Una herramienta para la coordinación de la manipulación de múltiples robots es propuesta en [14]. Es un método sencillo y eficaz para resolver el conflicto de la coordinación de varios robots industriales que operan en un mismo espacio común. Para ello, calcula las prioridades dinámicas para modificar la ruta y resolver los posibles conflictos. Se trata de un método desacoplado de dos fases. En la primera fase se utiliza un algoritmo $A^*$ para hallar la ruta sin colisiones con respecto a los obstáculos fijos para cada robot. En la segunda fase, se resuelve la coordinación entre los robots para evitar conflictos entre ellos.

- En [15] se desarrolla un controlador basado en una red neuronal en línea para la cooperación de varios robots manipuladores en la manipulación de un objeto rígido común. La integración del modelo dinámico del sistema manipulador y del objeto se deriva en términos de la posición y orientación del objeto, así como del resto de estados del modelo. Basándose en este modelo, se propone un controlador que consigue el seguimiento de la trayectoria requerida del objeto, así como el seguimiento de las fuerzas internas que surgen en el sistema. El algoritmo adapta-

tivo se deriva del análisis de estabilidad de Lyapunov, de modo que tanto la convergencia de errores como la estabilidad de seguimiento están garantizadas en el sistema de bucle cerrado.

- En [16] se presenta un método para coordinar el movimiento de varios manipuladores que comparten un espacio de trabajo común. La herramienta emplea controladores independientes de movimiento para cada manipulador unido por medio de restricciones en los espacios comunes - cada robot se representa como una restricción en el otro espacio de configuración.

En conclusión, tras este análisis se puede observar que el problema de la coordinación de múltiples robots manipuladores es una cuestión muy relevante en cualquier aplicación industrial. En la Sección 3.5 se analizarán y clasificarán los métodos generales que existen para la resolución de este problema y se indicará cómo se ha resuelto en la librería presentada en esta tesis.

## 2.2. Introducción al concepto de laboratorio virtual y remoto

En disciplinas técnicas y científicas, la teoría no proporciona el conocimiento suficiente al usuario y la realización de experimentos con equipamiento real es fundamental para consolidar los conceptos teóricos adquiridos. Esto es especialmente importante en ingeniería, y en particular en robótica, que integra el conocimiento de diversas materias de ingeniería (mecánica, electrónica, ...) y matemáticas [17].

Afortunadamente, las nuevas tecnologías de la información y la comunicación (TICs), basadas en Internet, han tenido un enorme impacto en ingeniería, pudiendo ser utilizadas para mejorar la accesibilidad a los experimentos. Internet proporciona nuevas herramientas para todas sus disciplinas, al mismo tiempo que también facilita el desarrollo de estrategias adicionales en el ámbito de la educación, incluyendo formas interactivas para poder ilustrar, simular, demostrar, experimentar, operar, comunicar, ... Pero uno de los cambios fundamentales que ha tenido lugar en estas áreas es la manera en que se pueden extender los laboratorios tradicionales con Internet [18].

Los laboratorios tradicionales han sido durante mucho tiempo el único lugar de experimentación. Las principales ventajas de estos laboratorios son su alta interactividad con el equipo real y la motivación que supone al usuario observar el experimento y poder analizar los datos reales. Sin embargo, dichos laboratorios tienen asociado un alto coste en equipamiento, mantenimiento y consumo de energía. Además, requieren supervisión, presentan restricciones de espacio y personal, y obligan al usuario a estar físicamente en el laboratorio.

La inclusión de las TICs ha cambiado radicalmente el concepto de espacio físico. Esto ha hecho patente una serie de limitaciones en estos laboratorios ya que a pesar de su enorme importancia no pueden ofrecer la versatilidad idónea [19].

En [1] se proponen dos criterios para caracterizar las diferentes modalidades de los nuevos entornos:

I) Según la manera en la que se accede a los recursos para el propósito de la experimentación, los entornos pueden ser *locales* o *remotos*.

II) Según la naturaleza física del sistema, los laboratorios pueden ser *reales* o *simulados*.

Combinando estos criterios se presentan cuatro tipos de entornos de experimentación, resumidos en la Figura 2.1:

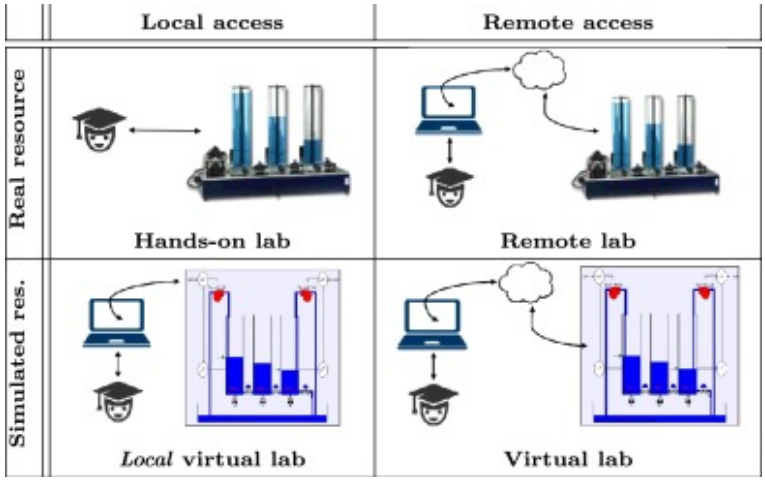



Figura 2.1: Diferentes modalidades de entornos de simulación (fuente [1]).

- *Acceso local - recursos reales*. Esta combinación presenta el laboratorio tradicional, donde el usuario controla directamente la planta real.
- *Acceso local - recursos virtuales*. En esta opción se presenta un laboratorio virtual local, es decir, un laboratorio presencial con el entorno de la planta simulado.
- *Acceso remoto - recursos reales*. Se accede a los equipos de la planta real a través de Internet. El usuario opera y controla la planta remotamente a través de una interfaz de experimentación. Este enfoque se conoce como laboratorio remoto.
- *Acceso remoto - recursos virtuales*. Esta forma de experimentación es similar a la anterior, pero reemplaza al sistema físico por un modelo simulado. Esto es lo que usualmente se conoce como laboratorio virtual.

La importancia y el uso de los laboratorios virtuales y remotos ha ido creciendo a lo largo de estos últimos años ([20]) a medida que la tecnología ha avanzado y se han resuelto algunas de sus principales preocupaciones. Dichos laboratorios permiten a los ingenieros adquirir conocimientos prácticos, bien sea operando con entornos simulados u operando remotamente con el equipo real [21]. Además, proporcionan otras ventajas adicionales como son ([22]):

i) Disponibilidad: se pueden utilizar desde cualquier lugar en cualquier momento.

ii) Observabilidad: las sesiones del laboratorio pueden ser observadas por muchas personas al mismo tiempo.

iii) Accesibilidad: los laboratorios están accesibles para gente con minusvalías.

iv) Seguridad: son la mejor alternativa a los laboratorios prácticos cuando se trata de experimentos peligrosos.

Existe un gran número de estudios de psicología que demuestran que las personas adquieren mejor el conocimiento haciendo cosas y reflexionando sobre las consecuencias de sus acciones que mirando o escuchando a alguien que les cuenta lo que deben aprender [23]. Además, la experimentación obliga al usuario a implicarse y permite probar el conocimiento conceptual, trabajar en colaboración, interactuar con el equipo, aprender por ensayo-error, y realizar análisis sobre datos experimentales [24].

Un *laboratorio virtual* es un sistema computacional que pretende aproximar el ambiente de un laboratorio tradicional. Dicho laboratorio se puede implementar como programa de escritorio (que se ejecuta de forma nativa en el sistema operativo del usuario) o aplicación web [25]. Aunque en este caso no se interacciona con plantas reales, la experimentación con modelos simulados es comparable siempre que se usen modelos matemáticos realistas que presenten los detalles importantes del sistema a analizar y se visualicen sus datos para permitir entender su comportamiento [17].

Las principales ventajas de los laboratorios virtuales son:

- Debido a que están basados en modelos matemáticos, su configuración y puesta a punto es más sencilla que la de un laboratorio real.
- Presentan un grado de robustez y seguridad más elevado.
- No imponen restricciones de tiempo ni espacio.
- Reducen el coste de montaje y mantenimiento de los laboratorios tradicionales.

En contra, tenemos los siguientes inconvenientes:

- Están limitados por el modelo y pueden provocar una pérdida parcial de la visión del sistema real con el que se está experimentando.
- No pueden sustituir completamente a la experiencia práctica del laboratorio real.

Los laboratorios remotos se pueden considerar como una extensión de los laboratorios virtuales. Dado que realizan los experimentos con equipos reales, generalmente son más complejos y caros

que los laboratorios virtuales [20]. Para abordar esta complejidad, siguen una arquitectura servidor-cliente. El lado cliente suele ser una aplicación web que interactúa con el servidor para controlar la configuración real y visualizar la información del laboratorio. El servidor controla el experimento mediante el uso de tarjetas de adquisión de datos, instrumentos de medida, conexiones con interfaces diversas, comunicación de datos, ...

Los laboratorios remotos presentan más ventajas que los virtuales [19, 25]:

- Presentan mayor interactividad con el equipo real.
- Permiten aprovechar los laboratorios tradicionales existentes, al integrar el sistema computacional con el equipo real. Reducen los costes de nuevas implementaciones.
- No se pierde la perspectiva, la respuesta del sistema es la de un sistema real y solo se usa simulación para comparar resultados.
- No imponen restricciones de tiempo ni espacio.
- Aprovechan los últimos avances tecnológicos.

No todo son ventajas, también presentan inconvenientes:

- La experimentación en tiempo real exige periodos de muestreo relativamente pequeños, requiriendo recursos y hardware que, por lo general, resultan costosos.
- Es necesario implementar protocolos de comunicación y esquemas de seguridad, siempre sujetos a vulnerabilidades.
- Es necesario utilizar entradas y salidas digitales o analógicas para poder manipular e interactuar con el hardware, el cual debe ser robusto para que no falle en ningún momento.

En conclusión, podemos asegurar que los laboratorios virtuales y remotos son una herramienta indispensable a día de hoy tanto en investigación como en docencia. En [26, 27, 28] se presentan diferentes trabajos que evalúan el impacto que tienen los laboratorios virtuales y remotos de robótica en el ámbito de la docencia, obteniendo como conclusión que los alumnos valoran positivamente dichas herramientas, ya que les ofrecen grandes ventajas frente a otras alternativas o aplicaciones.

Por otro lado, dichas herramientas no son sólo aplicadas a la investigación y la docencia, también son una pieza fundamental en el prototipado de plantas industriales. La industria está en constante desarrollo de sus plantas productivas a través de sistemas automatizados. Cabe resaltar la necesidad que existe de generar diseños de estos sistemas de automatización industrial, presentar sus modelos matemáticos para su control y, a través del enfoque actual, producir prototipos capaces de satisfacer las necesidades productivas, de desarrollo e investigación dentro de la industria. Esto

permite a los diseñadores estudiar el modelo y someterlo a diferentes aplicativos y condiciones, una vez establecidas sus condiciones de funcionamiento mediante la modelización cinemática y dinámica. Con dichos modelos se puede buscar una aplicación determinada y establecer las condiciones funcionales del sistema que permitirán la generación de prototipos en la industria [29]. La posibilidad de obtener prototipos sin que para ello sea necesario diseñar y fabricar los útiles supondría, además de un evidente ahorro de tiempo, un ahorro de costes importante [30].

Algunos ejemplos de uso serían los siguientes: En [31] se facilita el prototipado rápido de un robot Stanford con 6 GDL cuyo objetivo principal es presentar el diseño funcional que cumpla con los 6 GDL y la configuración específica de este manipulador. En [32] se analizan los problemas que el robot industrial presenta al aplicar la técnica del prototipado rápido, se discuten las tecnologías clave y se exponen las tendencias de prototipos rápidos basados en robots industriales. Finalmente, en [33] se exponen soluciones sobre el desarrollo de una plataforma robótica con un prototipado rápido. Construir una plataforma robótica para investigaciones científicas significa encontrar un sistema eléctrico y mecánico robusto que esté conectado a un dispositivo computacional y que sea fácilmente programable. Se presenta un nueva plataforma para un robot móvil (Neobotix MP500) programado con LabVIEW.

## 2.3. Revisión de los principales laboratorios virtuales y remotos

Durante los últimos años, una larga variedad de laboratorios virtuales y remotos ha ido surgiendo como base para el estudio y análisis de los sistemas robóticos. Estos laboratorios han sido desarrollados haciendo uso de diferentes plataformas y herramientas, por lo que el usuario puede encontrar entornos de simulación de código abierto y otros de uso comercial. La compejidad de estas herramientas depende principalmente del fin para el que hayan sido desarrolladas, desde aplicaciones sencillas para el ámbito de la enseñanza hasta entornos complejos para el estudio de prototipos de aplicaciones industriales.

En la literatura podemos encontrar trabajos cuyo fin ha sido analizar y comparar con detalle los principales entornos de simulación y herramientas tanto de uso comercial como gratuito que han ido surgiendo desde finales del siglo XX hasta el día de hoy. Entre todos estos estudios se encuentran [34] y [35], donde se facilita una gran variedad de estos entornos, entre los que podemos destacar, por ejemplo, los siguientes:

- Entornos de uso comercial:
  - *Webots*: Entorno de desarrollo utilizado para modelar, programar y simular robots móviles. Con Webots, el usuario puede diseñar configuraciones robóticas complejas, con uno o varios robots, similares o diferentes, en un entorno compartido. Las propiedades de cada objeto, como forma, color, textura, masa, fricción, etc., son elegidas por el usuario.

Los controladores de los robots pueden ser programados con la interfaz incorporada (ver Figura 2.2) o con entornos de desarrollo externos [36].

Webots es utilizado por más de 1281 universidades y centros de investigación de todo el mundo. Debido a que Webots es un entorno de uso comercial, se puede encontrar una versón gratuita (con ciertas limitaciones), Webots EDU, para el ámbito de la educación y, Webots PRO, en el caso de que ya haya adquirido una licencia Webots.

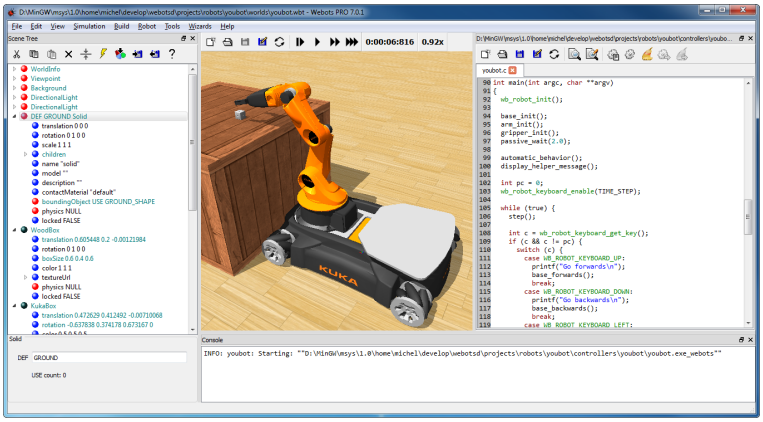


Figura 2.2: Interfaz de usuario de Webots

A continuación se detallan las características esenciales que hacen que esta herramienta de simulación sea tan potente y fácil de usar [37]:

- Modela y simula cualquier robot móvil, incluyendo ruedas, patas y robots voladores.
- Dispone de una biblioteca completa de sensores y actuadores que el usuario puede utilizar para configurar los robots.
- Le permite programar los robots en C, C++, Java y Python, o desde otros software externos conectados a través de TCP/IP.
- Transfiere controladores a robots móviles reales, incluyendo: Aibo, Lego Mindstorms, Khepera, Koala y Hemisson.
- Utiliza la biblioteca ODE (Open Dynamics Engine) para simulación precisa de la física.
- Permite crear películas de simulación AVI o MPEG para la web y presentaciones públicas.
- Incluye muchos ejemplos con el código fuente del controlador y modelos de robots comercialmente disponibles.
- Permite simular sistemas multi-agente, con sistemas globales y medios de comunicación locales.

- *Microsoft Robotics Developer Studio (MRDS)*: Es un entorno basado en Windows para

desarrolladores aficionados, profesionales, académicos y comerciales que les permite crear aplicaciones de robótica dirigidas a una amplia gama de escenarios [38].

El modelo de programación MRDS se puede aplicar a una gran variedad de plataformas robóticas, lo que permite a los usuarios transferir sus habilidades a través de múltiples plataformas. La funcionalidad de MRDS puede extenderse fácilmente con bibliotecas o servicios adicionales incorporados por los usuarios y proveedores de hardware o software.

MRSD soporta escenarios conectados de forma remota desde un PC con Windows al robot a través de un puerto serie, Bluetooth, WiFi o módem RF, o bien ejecutarse directamente en el robot de forma autónoma si éste posee un PC que ejecuta uno de los sistemas operativos Microsoft Windows.

Las aplicaciones de robótica diseñadas en MRDS pueden desarrollarse utilizando diferentes lenguajes de programación: C# en Microsoft Visual Studio y Microsoft Visual Studio Express, o el Visual Programming Language (VPL) que se incluye en el paquete MRDS.

MRDS incluye cuatro componentes principales:

- CCR: La concurrencia y coordinación en tiempo de ejecución (CCR) simplifica la escritura de los programas para manejar la entrada asincrona desde múltiples sensores robóticos y salida a motores y actuadores.
- DSS (Servicios de software descentralizados): La arquitectura orientada a servicios (SOA) de DSS facilita el acceso y la respuesta al estado de un robot mediante un navegador Web o una aplicación basada en Windows.
- VPL (Lenguaje de programación visual): Este lenguaje de programación visual permite a los desarrolladores crear y depurar aplicaciones robóticas fácilmente. Simplemente arrastrando y soltando los componentes en un lienzo y conectándolos juntos tal y como se puede observar en la Figura 2.3.
- VSE (Entorno de simulación visual): El potente entorno de simulación visual (VSE) proporciona un entorno de simulación de alta fidelidad basado en la tecnología NVIDIA, PhysX, que permite ejecutar simulaciones 3D de alta calidad.

La última versión Microsoft Robotics Developer Studio 4 tiene la funcionalidad de la versión anterior con la adición del soporte para el sensor Kinect y una plataforma de referencia definida, también conocida como MARK (Mobile Autonomous Robot usando Kinect).

Finalmente, para ayudar a los desarrolladores a comenzar, Robotics Developer Studio contiene una amplia documentación y un gran conjunto de ejemplos y tutoriales que ilustran cómo escribir aplicaciones que van desde simples ejemplos como "Hello Robot"hasta complejas aplicaciones que se ejecutan simultáneamente con varios robots (disponible en

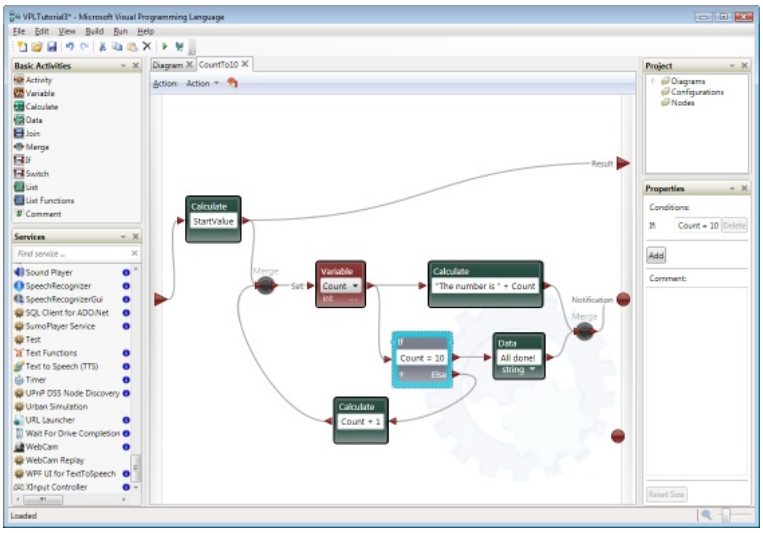



Figura 2.3: Lenguaje de programación visual incluido en MRDS

[39]).

- *Robotics System Toolbox / Matlab-Simulink*:

  Matlab es un lenguaje de alto nivel y un entorno interactivo para la computación numérica, desarrollo de algoritmos, visualización y análisis de datos. Matlab es el software más sencillo y productivo para ingenieros y científicos. Dicho entorno es, asimismo, una parte fundamental del diseño basado en modelos, que se utiliza para simulación multidominio, simulación física y de eventos discretos y, verificación y generación de código. Se utiliza para aprendizaje automático, procesamiento de señales, procesamiento de imágenes, visión artificial, comunicaciones, finanzas computacionales, diseño de control, robótica y muchos otros campos [40].

  Sus características principales son:

  - Lenguaje de alto nivel (basado en matrices) para cálculos de ingeniería y científicos.
  - Entorno de escritorio afinado para la exploración, el diseño y la solución de problemas de manera iterativa.
  - Gráficos para visualizar datos y herramientas para crear gráficos personalizados.
  - Apps para ajustar curvas, clasificar datos, analizar señales y muchas otras tareas relacionadas con dominios concretos.
  - Toolboxes complementarias para una amplia variedad de aplicaciones de ingeniería y científicas.
  - Herramientas para crear aplicaciones con interfaces de usuario personalizadas.
  - Interfaces para C/C++, Java, .NET, Python, SQL, Hadoop y Microsoft Excel.

- Opciones de despliegue sin coste en concepto de derechos de licencia para compartir programas de Matlab con los usuarios finales.

Simulink es un entorno de diagramas de bloque para la simulación multidominio y el diseño basado en modelos. Admite el diseño y la simulación a nivel de sistema, la generación automática de código y la prueba y verificación continuas de los sistemas embebidos [41]. Las características fundamentales de este entorno son:

- Editor gráfico para crear y gestionar diagramas de bloques jerárquicos.
- Bibliotecas de bloques predefinidos para modelar sistemas continuos y discretos.
- Motor de simulación con solvers de ecuaciones diferenciales ordinarias de paso fijo y paso variable.
- Scopes y data displays para ver los resultados de la simulación.
- Herramientas de gestión de proyectos y datos para administrar los archivos y los datos del modelo.
- Herramientas de análisis de modelos para perfeccionar la arquitectura del modelo y aumentar la velocidad de simulación.
- Bloque Matlab Function para importar algoritmos de Matlab en los modelos y poder exportar los resultados de la simulación para llevar a cabo más análisis.
- Legacy Code Tool para importar código C y C++ a los modelos.

En la Figura 2.4 se presenta un diagrama realizado en Simulink de un sistema de control de movimiento para el manipulador PUMA 560 basado en la técnica del jacobiano inverso.

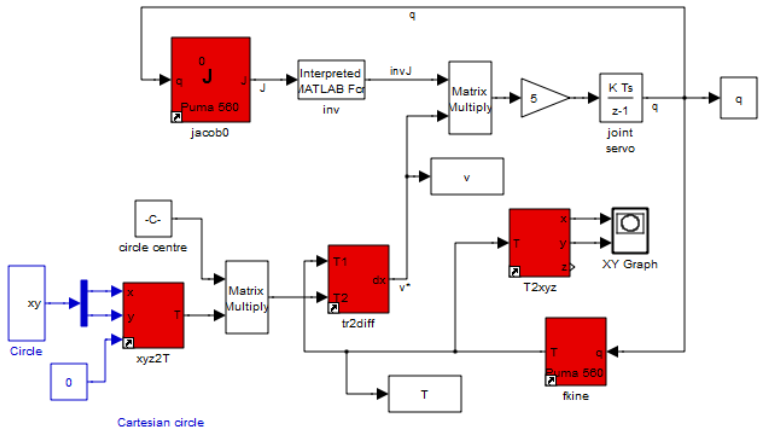



Figura 2.4: Diagrama de bloques en Simulink para el control de movimiento del robot manipulador PUMA 560

MATLAB y las toolboxes complementarias están integrados entre sí y diseñadas para trabajar conjuntamente. Estas herramientas le ofrecen a Matlab funcionalidades específicas para aplicaciones científicas y de ingeniería, que se han desarrollado de manera

profesional, probado rigurosamente, mejorado con la experiencia en cada campo y completamente documentadas.

En definitiva, Matlab es una de las plataformas más usadas para el modelado y simulación de diferentes tipos de sistemas y especialmente para la simulación de sistemas robóticos. Como extensión de este entorno, Simulink añade propiedades para facilitar la simulación de sistemas dinámicos, por ejemplo, modelos gráficos y la posibilidad de simular en tiempo real. En [42] se presentan y se comparan diferentes herramientas para la simulación de robots manipuladores en el entorno de Matlab/Simulink. De entre todas estas podemos destacar *Robotics Toolbox*, librería para el entorno de Matlab-Simulink, que provee las cinemáticas, dinámicas y generación de trayectorias. Dicha herramienta es útil para la simulación así como para el análisis de los resultados de experimentos con los robots reales, y puede ser una herramienta muy adecuada en el ámbito de la educación [43].

En 2015, Matlab presenta *Robotics System Toolbox* para una completa integración con Robot Operating System, ROS, (entorno descrito en esta misma sección). Robotics System Toolbox proporciona una integración completa entre MATLAB-Simulink, y ROS. Los algoritmos que se ofrecen dentro de esta herramienta incluyen representación cartográfica, planificación de rutas y seguimiento de rutas para robots con tracción diferencial [44].

Robotics System Toolbox permite también trabajar en un entorno de diseño único e integrado con el objetivo de diseñar, probar y desplegar algoritmos en robots habilitados para ROS y simuladores robóticos como Gazebo y V-REP (ver Figura 2.5).

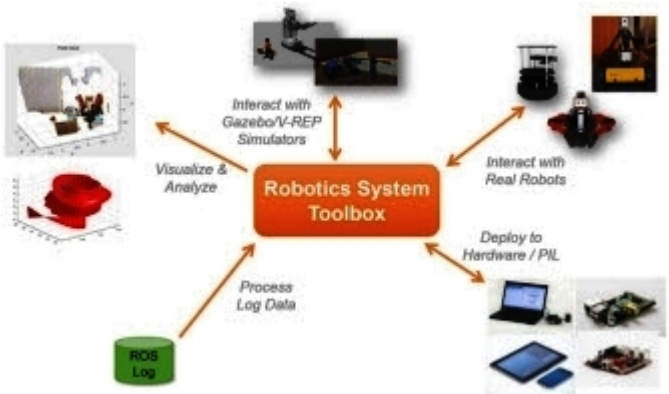



Figura 2.5: Robotics System Toolbox conecta MATLAB y Simulink con simuladores y robots habilitados para ROS

- Entornos de código abierto:
  - *USARSim (Unified System for Automation and Robotics Simulation)* es un simulador multi-robot de propósito general desarrollado inicialmente para la simulación de vehícu-

los que realizaban operaciones de búsqueda y rescate en un escenario urbano. USARSim está basado en la plataforma Unreal Engine 2.0 y permite el modelado realista de robots, sensores y actuadores, así como de entornos complejos, no estructurados y dinámicos [45].

USARSim ha sido seleccionado por la federación Robocup como infraestructura de software en dos competiciones robóticas y por el IEEE para apoyar la educación robótica [46].

Esta herramienta ofrece ciertas características que lo diferencian de otros simuladores existentes[47]:

- Es un simulador gratuito de código abierto. Para empezar a utilizar este entorno se necesita adquirir una licencia del motor de juego, y luego descargar desde el sitio web de sourceforge el módulo USARSim que se instalará encima de este motor de juego [48].
- USARSim es independiente de la plataforma y se ejecuta en Windows, Linux y Mac OSX.
- Es un simulador configurable y extensible. Los usuarios pueden agregar nuevos sensores o modelar nuevos robots.
- Se basa en la técnica de un motor de juego comercial. Cada mejora impulsada por la cada vez mejor industria del juego se traduce en ventajas de USARSim.
- USARSim se puede interconectar con Player, un middleware popular usado para controlar diferentes robots [49]. De esta forma, el código desarrollado dentro de USARSim puede ser trasladado de forma transparente a plataformas reales sin ningún cambio (y viceversa).

En la Figura 2.6 se facilita la arquitectura de USARSim. Esta arquitectura se basa en el modelo Cliente/Servidor, los controladores actuan como la parte cliente y GameBots y el motor Unreal son la parte servidor. GameBots se comunica entre el motor Unreal y los controladores mediante una interfaz TCP/IP. Para acceder a las propiedades del motor del juego, USARSim hace uso de GameBots, que proporciona una interfaz de red para el juego. Mediante el uso de un motor de juego comercial, la simulación es capaz de utilizar la simulación física del motor y sus capacidades gráficas.

En conclusión, según se cita en [46], se pueden dar cinco razones para resumir el notable éxito de este simulador. En primer lugar, su modelo distribuido permite a los desarrolladores contribuir con sus nuevas extensiones. En segundo lugar, aprovechando las ventajas de un motor de juego comercial, los desarrolladores robóticos pueden concentrarse en temas robóticos relevantes, mientras que delegan la visualización y otros aspectos al motor. En tercer lugar, la disponibilidad de interfaces compatibles con controladores ampliamente utilizados permite la migración de código de simulaciones a robots reales y viceversa sin costo alguno. La cuarta razón ha sido el acoplamiento de USARSim,

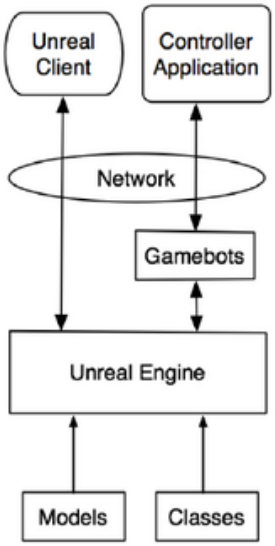



Figura 2.6: Arquitectura de USARSim

con Robocup, obligando así a la comunidad científica a fijar metas anuales y establecer un plan de investigación continuamente actualizado en sincronía con las necesidades robóticas del momento. Por último, los esfuerzos de validación convencieron a la gente más escéptica de que USARSim es una herramienta útil para desarrollar código que eventualmente se ejecutará en sistemas de robots reales.

- *ROS (Robot Operating System)* es un framework flexible que dispone una colección de librerías y herramientas cuyo objetivo es simplificar la tarea de crear aplicaciones robóticas complejas y robustas a través de una amplia variedad de plataformas robóticas [50, 51]. ROS provee hardware de abstracción, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes y más.

  ROS se contruyó desde cero bajo la licencia de código abierto y ha ido creciendo exponencialmente para promover y facilitar el desarrollo de software colaborativo para Robótica [52]. El código compartido ayuda a la comunidad robótica a progresar más rápido al permitir que los investigadores conozcan y extiendan el trabajo de otros grupos de investigación [53].

  En el nivel más bajo, ROS ofrece una interfaz de paso de mensajes que proporciona la comunicación entre los procesos y que comúnmente se conoce como un middleware. El middleware ROS ofrece lo siguiente:

  - Publicación y suscripción de paso de mensajes anónimos.
  - Grabación y reproducción de mensajes.
  - Petición y respuesta de llamadas a procedimientos remotos.

- Sistemas de parámetros distribuidos.

Además de los componentes básicos del middleware, ROS proporciona bibliotecas robóticas comunes y herramientas que harán que un robot esté en funcionamiento rápidamente. A continuación se detallan algunas de las capacidades que presentan estos robots:

- Definición estándar de mensajes de robots.
- Biblioteca geométrica.
- Descripción y modelado del robot mediante un lenguaje descriptivo.
- Llamadas a procedimientos remotos apropiadas.
- Diagnóstico del estado del robot.
- Resolución de estimación, localización y navegación.

En [54] se puede encontrar el listado de todos los robots que están soportados por ROS: robots manipuladores, robots móviles, vehículos autónomos, humanoides, drones, vehículos submarinos autónomos,... Además, en esta wiki se encuentran las instrucciones a seguir para añadir un nuevo robot a esta lista.

De manera análoga, en [55] se facilitan todos los sensores de robótica que son compatibles con ROS: telemétros 1D, telémetros 2D, sensores 3D, sensores de audio y reconocimiento de voz, cámaras, sensores de fuerza, GPS,...

ROS permite una integración perfecta con otras librerías y otros proyectos populares de código abierto, junto con un sistema de paso de mensajes que le permite cambiar fácilmente diferentes fuentes de datos, de los sensores en vivo en registro de datos.

Algunas de las librerías con las que se puede integrar ROS son: OpenCV, biblioteca de visión por ordenador, MoveIt, biblioteca de planificación de movimientos que ofrece implementaciones eficientes, Point Cloud Library, biblioteca para la manipulación y procesamiento de datos tridimensionales e imágenes de profundidad [56]. En la Figura 2.7 se muestra la integración de ROS con las librerías OpenCV y Point Cloud para el estudio de visión por computador.

Las desventajas que presenta este framework es que no es soportado en todos los sistemas operativos y requiere de una instalación compleja por lo que el usuario necesita una gran curva de aprendizaje. En [57] se facilitan tutoriales de diferentes niveles para la instalación y familiarización de este entorno.

- *V-REP (Virtual Robot Experimentation Platform)* es un simulador de robot, que contiene un entorno integrado para desarrollar aplicaciones robóticas y está basado en una arquitectura de control distribuida: cada objeto/modelo puede ser controlado individualmente mediante un script embebido, un plugin, un enlace a ROS, una aplicación cliente remota, o una solución propia. Esto hace que V-REP sea muy versátil e ideal para aplicaciones multi-robot.

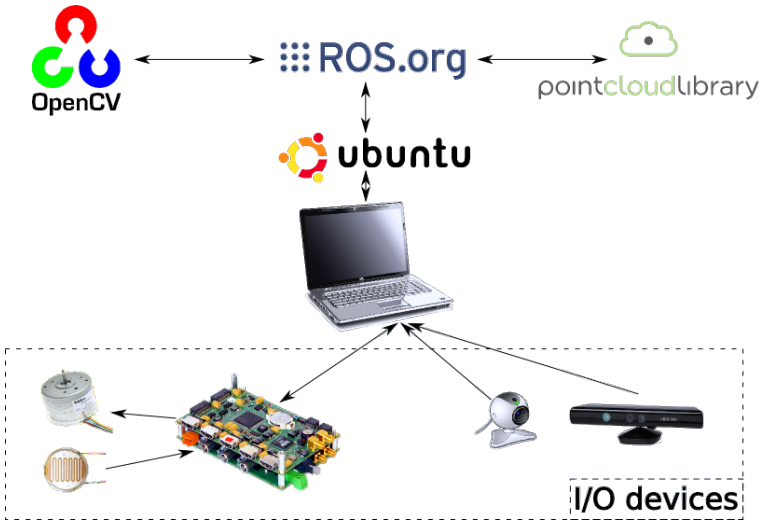



Figura 2.7: Integración de ROS con otras librerías

V-REP se emplea principalmente para el desarrollo rápido de algoritmos, simulación de sistemas de automatización, en el prototipado y verificación de sistemas, en educación, monitorización remota, control de hardware, vigilancia de seguridad,... [58, 59]

Uno de los inconvenientes de este simulador es que no es completamente gratuito. Existen cuatro versiones: V-REP PRO EDU, versión gratuita para el ámbito de la educación, V-REP PRO EVAL, versión comercial gratuita para evaluar el sistema, V-REP PLAYER, edición gratuita con capacidades limitadas para ejecutar las simulaciones y, V-REP source (no gratuita). Todas estas versiones están disponible en su web dada en [60].

V-REP es uno de los simuladores robóticos con más funciones, recursos o APIs. A continuación se resumen sus características principales:

- Es un entorno multiplataforma: Windows, MacOS y Linux.
- Integra seis enfoques diferentes de programación: scripts embebidos, plugins, add-ons, nodos ROS, APIs de clientes remotos, o soluciones personalizadas. Cada uno de estos tiene sus ventajas e inconvenientes particulares sobre el resto, pero los seis son mutuamente compatibles.
- Incluye 7 lenguajes de programación: C/C++, Python, Java, Lua, Matlab, Octave o Urbi.
- Ofrece una API remota con más de 100 funciones específicas y una genérica que puede ser llamada desde una aplicación C/C++, un script Python, una aplicación Java, un programa Matlab/Octave, un script Urbi o un script Lua.
- El módulo de dinámica de V-REP soporta 4 motores físicos diferentes (Open Dynamics

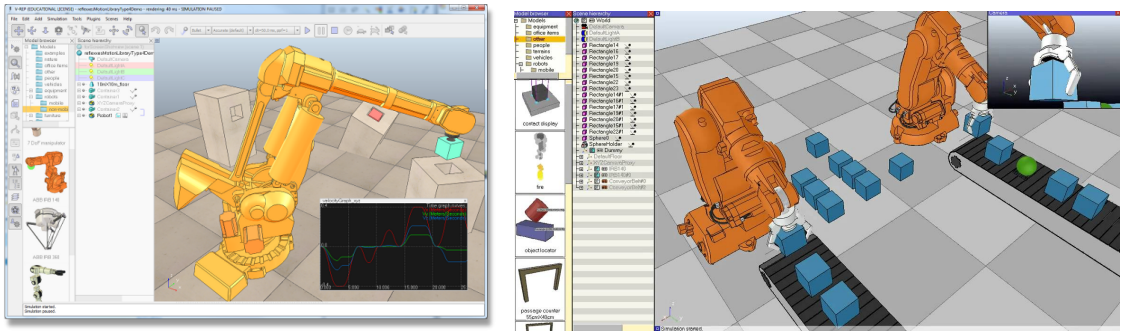
Engine, la librería física Bullet, Vortex Dynamics y Newton Dynamics) para cálculos dinámicos rápidos y personalizables, para simular la física del mundo real y las interacciones de objetos.

- Cálculo completo de la cinemática directa e inversa.
- Detección de interferencias u obstáculos.
- Cálculo de la distancia mínima entre objetos.
- Sensores de proximidad real.
- Sensores de visión con procesamiento de imagen integrado (totalmente extensible).
- Planificación de trayectorias.
- Registro y visualización de datos (gráficos de tiempo, gráfico X/Y o curvas 3D).
- Interfaces de usuario integradas y personalizadas, incluyendo un editor.
- Modos de edición de formas integrados: se pueden integrar nuevas formas diseñadas en una aplicación de CAD (por ejemplo, AutoCAD, 3D Studio Max, etc.) o crear directamente formas primitivas y personalizarlas con uno de los 3 modos de edición soportados en V-REP.
- Resulta fácil importar/exportar datos.
- Integración de la interfaz RRS (Realistic Robot Simulation) y la librería Reflexxes Motion .
- V-REP soporta partículas personalizables que pueden ser usadas para simular partículas de aire o agua.
- Navegador de modelos con función de arrastrar y soltar (drag and drop).
- Permite total interacción también durante la simulación: los modelos pueden ser desplazados, rotados, copiados, escalados, ... sin tener que ajustar el código.

En [61] se encuentra un manual de usuario con el detalle de todas las características de este simulador, descripción de sus interfaces, entornos, módulos, simulaciones, y otros tutoriales básicos para que el usuario conozca completamente este entorno y pueda trabajar en él sin ningún problema.

Finalmente, en la Figura 2.8 se facilitan diferentes ejemplos de simulaciones desarrolladas en V-REP. Una simulación para la planificación de trayectorias de un robot industrial es presentada en la Figura 2.8(a). En la Figura 2.8(b) se muestra un laboratorio robótico compuesto por dos cintas transportadoras y dos robots manipuladores que seleccionan y cogen los objetos que pasan por las cintas.

Además de estos entornos, en esta tesis se ha llevado a cabo un análisis exhaustivo de otras herramientas existentes en la literatura. Este análisis ha estado enfocado sobre todo en entornos Java



(a) Planificación de trayectorias (b) Coordinación y manipulación de objetos

Figura 2.8: Ejemplos de simulaciones de robots manipuladores en el entorno V-REP

ya que la librería presentada en esta tesis ha sido desarrollada en este lenguaje de programación. Posteriormente, en la Sección 4.1 se justificará en detalle los motivos por los que se decidió desarrollar esta nueva librería en Java y no en otro lenguaje. Por lo tanto, el objetivo de centrar esta revisión en entornos Java es para poder estudiar sus ventajas e inconvenientes y poder así compararlas con las de esta nueva librería.

A continuación se describen dichas herramientas:

- *JRoboOp* es una colección de clases de Java para la simulación robótica desarrollado por Carmine Lia. Estas clases abordan la implementación de la cinemática y la dinámica de un robot, basándose en el algoritmo de Denavit-Hartenberg, la detección de colisiones, la planificación de trayectorias, restricciones, incluyen controladores para el espacio articular y cartesiano del robot, soportan Java 3D para una fácil visualización 3D, resolución de ecuaciones diferenciales, funciones de matrices basadas en el paquete *JAMA*, funciones gráficas basadas en el paquete *Ptolemy plot*, configuración avanzada en XML con *XMLBeans* y ejecución de código más rápido con Javolution.

  JRoboOp está disponible en su wiki dada en [62]. Además, en esta web se puede encontrar el detalle de todo lo que esta herramienta puede hacer, la descripción de las clases de Java que lo integran, así como diferentes ejemplos de uso.

  Sin embargo, esta herramienta está enfocada exclusivamente en el desarrollo de simulaciones virtuales ya que carece de la posibilidad de conexión con el hadware real, parte esencial para el diseño de un laboratorio remoto. En la Figura 2.9 se representa el modelo 3D de un robot PUMA 560 para el control de sus 6 articulaciones haciendo uso del entorno JRoboOp.

- *RosJava* es la primera implementación pura de Java del entorno ROS descrito anteriormente. Desarrollado por Google en cooperación con Willow Garage, RosJava proporciona una biblioteca cliente para comunicaciones ROS en Java y una creciente lista de herramientas

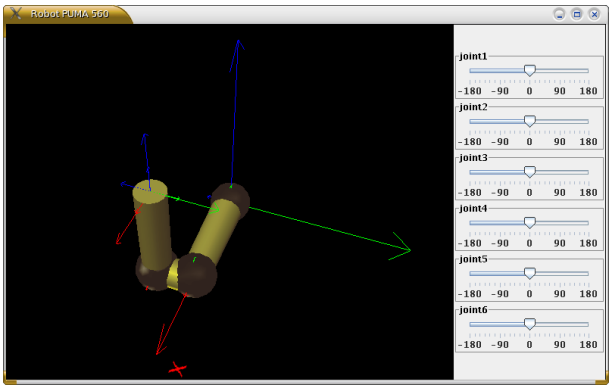



Figura 2.9: Representación del modelo de un robot PUMA 560 en JRoboOp

principales y controladores. RosJava permite la integración de robots compatibles con Android y ROS. Además, dicha implementación facilita a los usuarios que no utilizan ROS el que sólo usen los binarios de RosJava para vincular sus propios proyectos a cualquier robot ROS.

En su wiki dada en [63] se puede encontrar documentación acerca del ecosistema RosJava e instrucciones a seguir para su instalación. El código fuente oficial de los proyectos RosJava se puede encontrar en el directorio github de la organización rosjava [64].

Finalmente, tras la descripción del entorno ROS y RosJava, en la Figura 2.10 se puede observar una simulación de un robot ABB IRB 2400 diseñada en este entorno para controlar los valores de sus articulaciones.

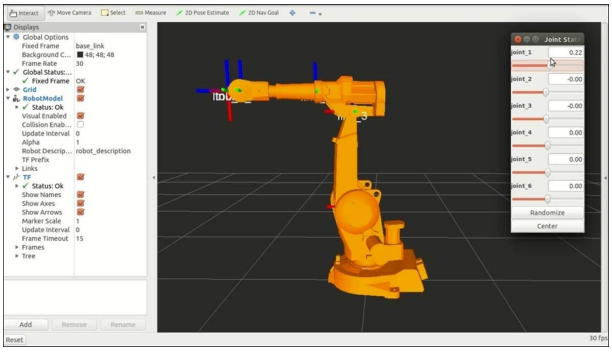



Figura 2.10: Control de las articulaciones de un robot ABB IRB 2400 en el entorno ROS

- *dLife* es una librería desarrollada en Java enfocada en la enseñanza e investigación en Robótica, inteligencia artificial y visión por ordenador. La librería incluye paquetes para redes neuronales, algoritmos de genética, robótica y visión artificial. dLife es una de las librerías que

están diseñadas para soportar la programación de una gama de plataformas de hardware robóticos que utilizan la misma API y mismo lenguaje y, por lo tanto, resulta un entorno particularmente adecuado para trabajar en desarrollos robóticos.

DLife actualmente soporta los siguientes robots: Pioneer, Hemisson, Khepera III, Khepera II, Scribbler+Fluke, ARDrone, Sony Aibo y Finch. Para muchos de estos robots los controladores dLife se pueden ejecutar tanto en simulación como en un robot físico. Además, dLife tiene integradas clases e interfaces que soportan la interacción con el sistema de simulación Player/Stage.

Para cada plataforma, dLife soporta los sensores y efectores incluidos con la unidad base, así como otros complementos disponibles [65]. La documentación completa de las funciones soportadas para cada robot se puede encontrar en formato Javadoc estándar en el sitio web dLife en `www.dickinson.edu/~braught/dlife` [66].

Las interfaces gráficas como las de la Figura 2.11 permiten al usuario aprender sobre el robot, moviéndolo manualmente (por ejemplo, encontrar los valores de panorámica / inclinación / zoom para una cámara) y observando los valores de sus sensores (por ejemplo, sensores de proximidad de infrarrojos a una distancia específica de un obstáculo). Entonces, debido a que existe una correspondencia directa entre la API de dLife y las visualizaciones, las acciones realizadas manualmente en la interfaz se convierten fácilmente en acciones a ejecutar en el programa.

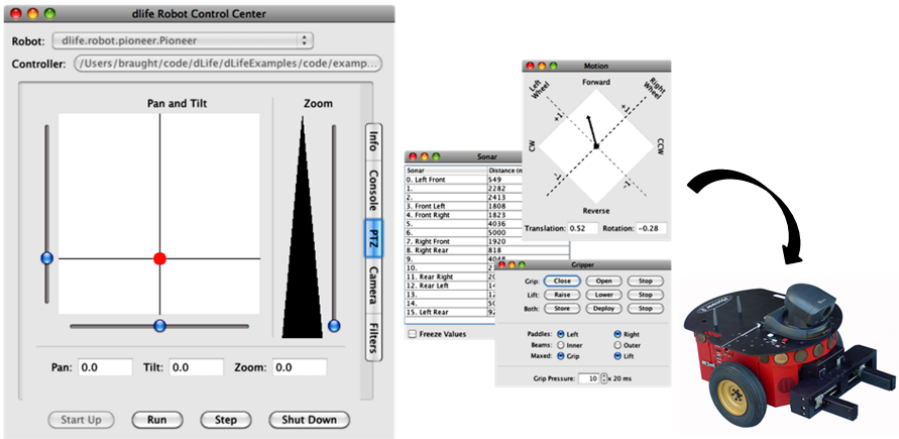



Figura 2.11: Interfaces para el control de sensores y actuadores de un robot Pioner con dLife

- *EjsRL (Easy Java Simulations Robotics Library)* es una biblioteca gráfica de alto nivel basada en Java 3D, diseñada específicamente para el software de programación EJS, que proporciona un completo marco para modelar, desarrollar y ejecutar aplicaciones avanzadas de Robótica y visión por ordenador (R&CV).

La biblioteca se compone de cuatro bloques principales [4]:

- Robótica: Incluye todos los algoritmos de robótica necesarios para el modelado y simulación de robots manipuladores, como son la cinemática y dinámica (directa e inversa), y planificación de trayectorias. También proporciona otras funciones avanzadas, como la programación virtual de robots con lenguaje Java, o la operación remota de los equipos reales.
- Cálculo Matricial: Permite realizar los cálculos matemáticos necesarios para resolver todos los algoritmos R&CV. Cubren las operaciones fundamentales del álgebra lineal numérica, el matricial y vectorial (suma, resta, multiplicación, determinantes, inversa, norma,...), además de toda la matemática esencial para los algoritmos de robótica, tales como rotaciones de ejes, transformaciones de Euler, representación Roll-Pitch-Yaw, representaciones matriciales, etc.
- Visión artificial: Proporcionan una amplia gama de funciones para el procesamiento de imágenes, y para el análisis de datos geométricos y reconocimiento de objetos. Combinando estas operaciones con las cámaras virtuales que ofrece el entorno gráfico 3D de Ejs, y con las funciones de robótica, se pueden crear aplicaciones de control visual o de manipulación de objetos.
- Comunicación remota: Las funciones contenidas en este módulo permiten establecer conexiones a través de internet mediante el protocolo HTTPS. Contiene una serie de interfaces para poder establecer comunicaciones mediante el protocolo HTTPS para operar con sistemas remotos de una manera muy sencilla.

Además de los paquetes antes mencionados, la biblioteca incorpora funciones de importación/exportación para diferentes formatos de archivo con el fin de permitir a los usuarios guardar y restaurar sus diseños.

El diseño de este software se basa en una coordinación jerárquica entre EJS (ver Sección 2.6) y EjsRL, los dos componentes de la plataforma, tal y como se muestra en la Figura 2.12. Por un lado, EJS proporciona la interfaz gráfica 2D/3D y herramientas para la construcción del modelo dinámico. Por otro lado, EjsRL proporciona todos los algoritmos de R&CV que son necesarios para simular el comportamiento del sistema dentro de una aplicación EJS [67].

Una desventaja de esta librería es que las operaciones de los robots son demasiado dependientes de la arquitectura de EJS lo cuál hace que resulte muy difícil manejar más de un robots en una misma simulación.

Finalmente, en la Figura 2.13(a) se muestra una simulación desarrollada con EjsRL para la planificación de trayectorias de un robot de 6 GDL. En la Figura 2.13(b) se muestran otros tres ejemplos. La primera aplicación se trata de un laboratorio virtual y remoto diseñado para el aprendizaje de Robótica, llamado RobUAlab. Este laboratorio está accesible en su

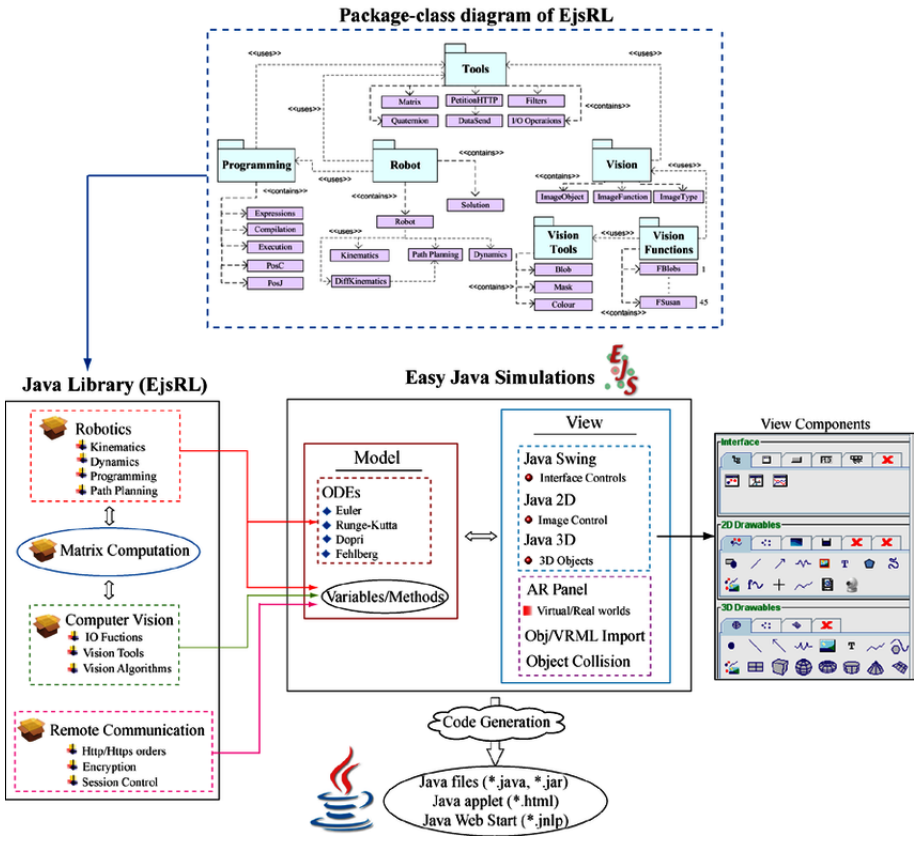
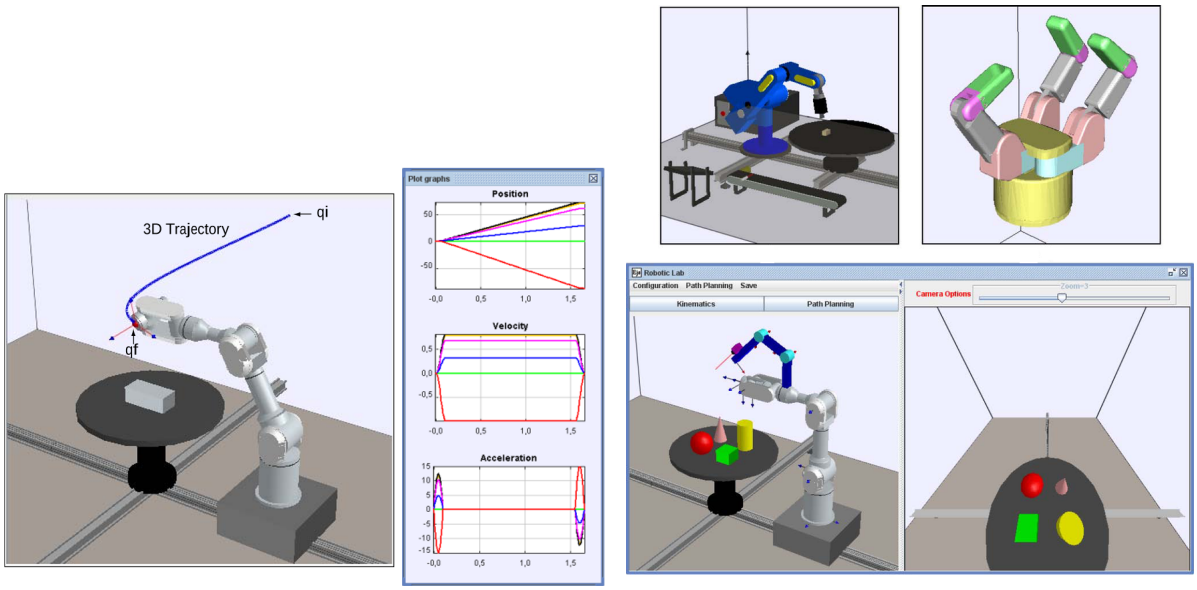



Figura 2.12: Arquitectura software y diagrama de clases de EjsRL

web `http://www.aurova.ua.es/robolab/`. La segunda aplicación consiste en la simulación de una mano robótica que representa prácticamente un sistema real desarrollado por la empresa Barrett Technology. La tercera simulación es un sistema multi-robot compuesto por dos manipuladores acoplados y una cámara ubicada en uno de ellos cuya proyección se muestra en la interfaz gráfica situada a la derecha.

Por último, para concluir con esta revisión del estado del arte nos vamos a centrar en el ámbito de la docencia, donde en los últimos años han aparecido múltitud de laboratorios virtuales y remotos de diferente complejidad. La finalidad de todos ellos es permitir al estudiante aprender los conceptos básicos de Robótica de una manera práctica y sencilla. A continuación se detallan algunos de estos ejemplos.

- Un laboratorio robótico virtual es presentado en [68]. Este laboratorio está basado en la orientación a objetos y arquitectura distribuida. Posee la capacidad de definir sesiones prácticas que pueden ser ejecutadas en un entorno visual. Se ha integrado un lenguaje propio muy básico para robots manipuladores y móviles, común a todos ellos. Es un lenguaje con sentencias




(a) Planificación de trayectorias de un robot de 6 GDL (b) Otras aplicaciones robóticas

Figura 2.13: Ejemplos aplicaciones robóticas avanzadas desarrolladas con EJS + EjsRL

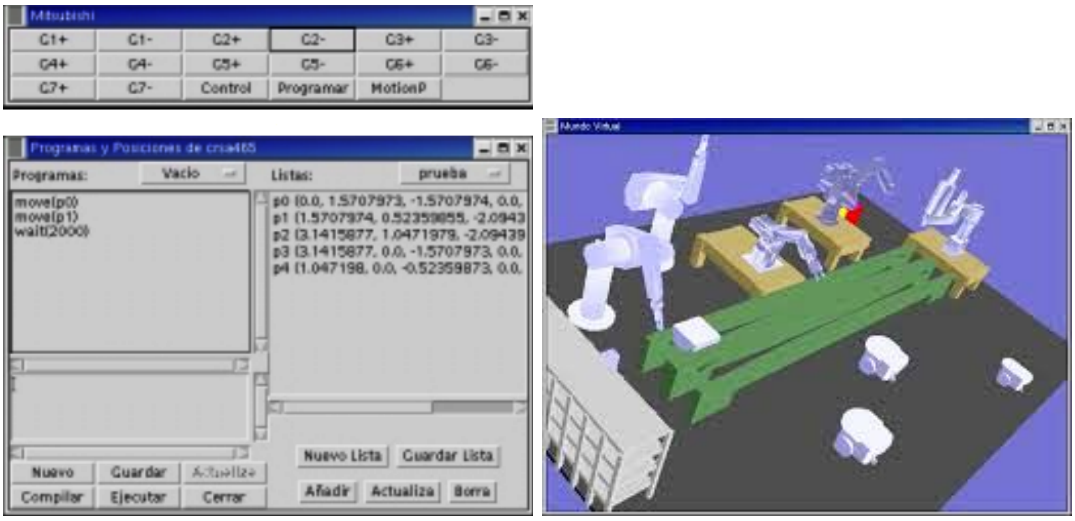
sencillas como: *if ... else*, estructura para el control de decisiones, *move(p)*, mueve el robot a la posición p, *move(f,g)*, mueve g unidades la articulación f, *wait(m)*, espera m milisegundos para ejecutar la siguiente instrucción .... El lenguaje también permite declarar variables enteras y operaciones aritméticas. Dicho lenguaje contiene instrucciones para coordinar los programas entre los distintos robots y se ha implementado una interfaz para definir las tareas como conjuntos de condiciones y posicionar los robots manipuladores (ver Figura 2.14(a)).

En la Figura 2.14(b) se muestra la interfaz gráfica de un laboratorio virtual compuesto por cinco robots manipuladores, tres robots móviles y una cinta transportadora.

- Leonardo es un laboratorio remoto diseñado para un robot Scara de 4 GDL que ofrece al usuario la posibilidad de generar diferentes trayectorias que pueden ser visualizadas al mismo tiempo tanto en el robot real como en el simulado. La conexión con el robot real se establece mediante el sistema "cliente-servidor" TCP/IP [69].

  En la Figura 2.15 se puede observar el esquema de la arquitectura implementada para comunicar la aplicación cliente Leonardo con el robot Scara real.

  La aplicación cliente proporciona al usuario una interfaz sencilla para la planificación de trayectorias, simulación dinámica y el diseño del controlador, así como una interfaz para la visualización y ejecución de diversas operaciones del robot manipulador en un entorno simulado.




(a) Interfaz de programación (b) Interfaz gráfica

Figura 2.14: Interfaz de un laboratorio virtual con robots móviles y manipuladores

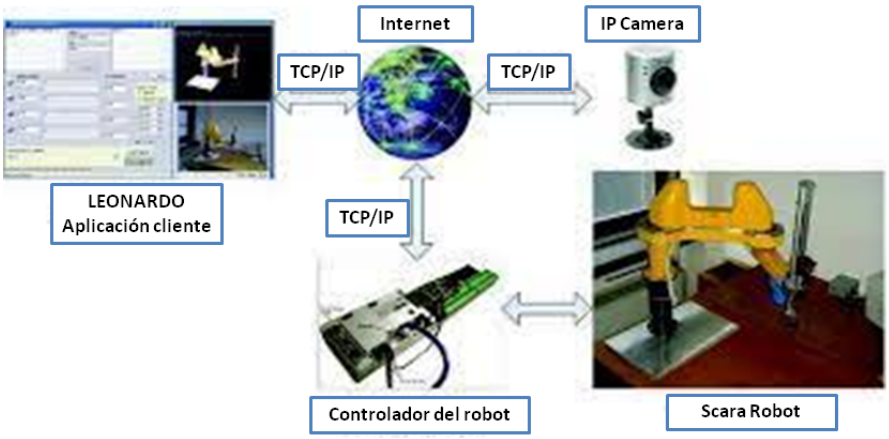



Figura 2.15: Arquitectura implementada entre la aplicación cliente Leonardo y el robot Scara

- Un laboratorio virtual para el control de un robot RV-M1 de Mitsubishi se describe en [70]. Este software desarrollado en el entorno de Matlab, representa el modelo 3D del robot y permite al usuario aprender las cinemáticas de este robot de Mitsubishi (ver Figura 2.16), así como definir aplicaciones de tipo “pick and place”.

- RoboCrane, interfaz gráfica de simulación de un robot Icasbot (IUST Cable Suspended Robot) con 6 GDL y 6 actuadores, ha sido diseñada apoyándose en el software LabVIEW y es compatible con todas las versiones del sistema operativo de Windows. El usuario es capaz de ver y estudiar la posición final del robot en un entorno virtual antes de operar con el robot

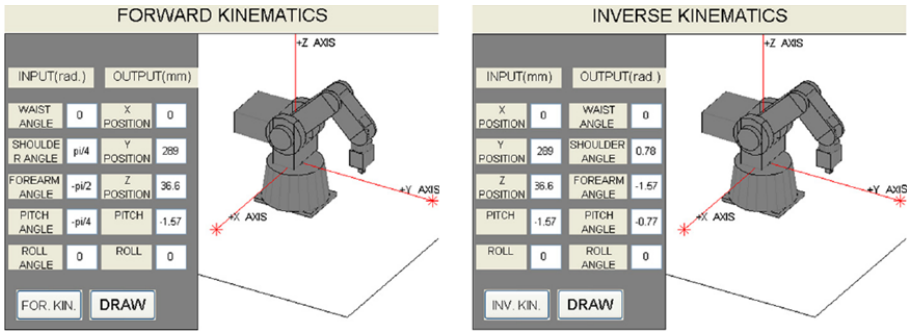



Figura 2.16: Simulación cinemática directa e inversa del robot RV-M1

real. La principal limitación de esta interfaz es que sólo se puede trabajar en Windows [71].

En la Figura 2.17 se puede observar la interfaz gráfica de RoboCrane donde el usuario puede analizar el movimiento del robot según los datos facilitados.

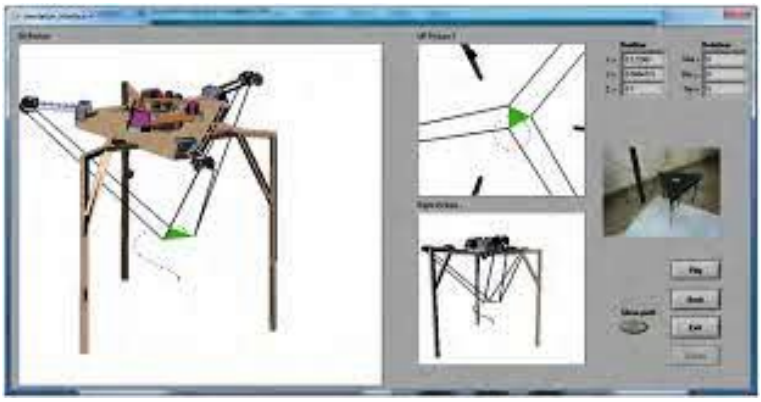


Figura 2.17: Interfaz gráfica de RoboCrane para analizar el movimiento del robot

- HEMERO es una herramienta software desarrollada para Matlab que permite resolver los problemas más comunes que se presentan en el campo de la Robótica y que ha sido aplicada con éxito en la última década en el ámbito de la docencia. HEMERO integra un conjunto de funciones para facilitar los cálculos más frecuentes relacionados con la representación de las posición y orientación, la cinemática, la dinámica y el control de los robots manipuladores. Asimismo se incluyen algunas funciones para la generación de trayectorias de robots manipuladores. En cuanto a la robótica móvil, se trata especialmente la cinemática empleándose modelos de diferentes configuraciones [72].

- En el laboratorio remoto de automática (LRA) de la universidad de León se dispone de un robot ABB con 6 GDL con una tarjeta Ethernet. Cualquier usuario registrado será capaz de

definir la trayectoria a seguir por el robot y programar tareas de ejecución (carga y ejecución de programas RAPID, posicionamiento del punto de control TCP con traslación y rotación, cambio del sistema de referencia de la pinza, apertura/cierre de pinza neumática, ...) [73].

Tras analizar cada una de estas herramientas y entornos se puede concluir que la creación de laboratorios virtuales y remotos es un proceso elaborado, ya que requiere de la simulación de los robots, creación de interfaces gráficas de programación para que el usuario pueda definir operaciones robóticas básicas (tal como especificar trayectorias sin violar algunas restricciones), visualización de los sistemas, y control de la comunicación con el hadware del robot real.

En el resto de tesis se introduce una nueva librería cuyo objetivo es facilitar la creación de nuevos laboratorios virtuales y remotos o el diseño de rápidos prototipados, usando una herramienta de código abierta, compatible en la mayoría de las plataformas, y que permite diseños de alto nivel e implementación y reutilización de estos laboratorios.

## 2.4. Análisis crítico de las herramientas existentes

Incluso cuando todos los entornos y librerías detallados en la Sección 2.3 podrían ser utilizadas para implementar parte de los requerimientos del entorno presentado en esta tesis para la creación de laboratorios robóticos virtuales y remotos, ninguno de ellos está completamente enfocado a cubrir todas las necesidades que cubre este nuevo framework. Las razones más destacadas por las que se debería usar esta nueva librería frente a cualquier otra se detallan a continuación.

Se trata de un framework sencillo y fácil de usar que no requiere de una gran curva de aprendizaje por parte del usuario, a diferencia de lo que ocurre con otros entornos, como puede ser el caso de ROS, cuya curva de aprendizaje es bastante elevada.

Es un entorno de código abierto y totalmente gratuito de manera que todas las funcionalidades de la librería están disponibles para todos los usuarios y no están condicionadas a las diferentes versiones que puedan existir del mismo, gratuitas y de pago, como ocurre, por ejemplo, con el software V-REP.

Es una herramienta compatible en la mayor parte de los sistemas operativos (en todos aquellos que soporten Java) a diferencia de otras herramientas como, por ejemplo, MRDS que fue desarrollada específicamente para sistemas operativos de Microsoft Windows.

La nueva librería incluye todas las operaciones básicas que aparecen en el momento de trabajar con robots manipuladores: posicionamiento, resolución de cinemáticas, planificación de trayectorias, detección de colisiones, definición de restricciones, coordinación de los distintos robots,... Por ejemplo, la librería EjsRL no permite establecer restricciones, ni detectar las posibles colisiones que puedan surgir y tampoco resuelve el tema de la coordinación.

Además, este framework permite el diseño de laboratorios remotos, a diferencia, por ejemplo, de JRoboOp, que incluye todas las operaciones básicas pero no permite diseñar laboratorios remotos, ya que carece de la posibilidad de conexión con el robot real.

En el caso de los laboratorios virtuales, esta nueva librería ofrece una visualización 3D muy realista de todos los elementos que integren este tipo de laboratorios. Dicha visualización está basada en la librería gráfica OSP que se detallará en la Sección 2.5.

Finalmente, la librería ha sido embebida en el software de simulación EJS, detallado en la Sección 2.6, porque es una herramienta con la que estamos familiarizados, fácil de utilizar y pensada para todos aquellos usuarios que no poseen un alto conocimiento de programación.

En el resto de la tesis se presentará en detalle cada una de las características de esta nueva librería y se mostrarán ejemplos de uso que justificarán las ventajas de usar dicho entorno frente a otros existentes.

## 2.5. Proyecto Open Source Physics

En las últimas décadas las computadoras han penetrado en las instituciones educativas de forma que hoy en día es impensable estudiar parte de la Física experimental sin el apoyo de estas. A pesar de estos avances en la enseñanza y la investigación, la física computacional sigue ausente en el típico programa educativo.

Los estudiantes son bombardeados con la realidad simulada, pero pocos estudiantes están preparados para evaluar críticamente estas simulaciones. Los estudiantes deben estar involucrados activamente en la computación y modelado de estas simulaciones de forma que les permita analizar, describir y explicar problemas difíciles, así como predecir los fenómenos físicos y visualizar sus resultados. De esta manera, con la combinación de la física computacional y el modelado informático, con teoría y el experimento pueden lograr un mejor entendimiento y comprensión del fenómeno que no se puede lograr con un sólo enfoque [74]. El proyecto OSP trata de abordar este asunto.

El proyecto Open Source Physics (OSP) es una biblioteca para la simulación de la Física desarrollada en Java bajo licencia libre GPL (Licencia Pública general de GNU) por la Fundación Nacional para la Ciencia y el Davidson College de Estados Unidos [75].

El OSP, www.compadre.org/osp/, [7], tiene como finalidad mejorar la enseñanza de la física computacional, proporcionando un sitio Web que contiene herramientas de modelado por computador, simulaciones, recursos curriculares, como planes de formación, y un libro de física computacional que explica los algoritmos de diferentes simulaciones pedagógicas. Estos recursos se basan en pequeñas simulaciones de un solo concepto empaquetadas con códigos fuente que pueden ser examinados, modificados, recompilados y redistribuidos libremente para enseñar habilidades computacionales fundamentales. Además, este sitio Web incluye foros de discusión donde los usuarios

pueden publicar preguntas e intercambiar ideas [76].

Los modelos dados en OSP se caracterizan por [77]:

- Facilitan la visualización de conceptos abstractos.
- Son interactivas y requieren la interacción del usuario.
- Plantean problemas reales. Estas simulaciones son un punto intermedio entre la teoría y el mundo real.
- La información se muestra en múltiples representaciones.
- Facilitan al usuario el entendimiento de los problemas planteados.

Los estudiantes de todos los niveles se benefician de estas simulaciones interactivas aprendiendo a cuestionar y evaluar las suposiciones y resultados de las simulaciones [6]. Estas simulaciones siguen un proceso pedagógico llamado *"Ciclo de aprendizaje"*.

En la primera fase del ciclo, *"Exploración"*, los estudiantes establecen preguntas o situaciones problemáticas sobre un fenómeno o concepto concreto y hacen predicciones sobre los posibles resultados. En esta fase inicial el estudiante piensa sobre el tema concreto, se plantea preguntas y utiliza sus conocimientos previos para construir una hipótesis sobre el resultado de la simulación.

En la segunda fase del ciclo, el profesor es el encargado de guiar a los estudiantes a través de la introducción y desarrollo de los conocimientos esenciales. En esta fase los estudiantes comienzan a compartir sus observaciones e ideas desde la fase de Exploración.

En la fase final de la aplicación, el profesor plantea nuevos problemas o situaciones para que los estudiantes los puedan resolver, basándose en la exploración inicial.

La transición del trabajo con simulaciones interactivas a modelado computacional puede ser especialmente difícil para los estudiantes. Además de aprender un lenguaje de programación, los estudiantes deben dominar una gama de técnicas, tales como compilar y vincular con gráficos y bibliotecas numéricas, antes de poder crear un programa de computadora en ejecución. Para minimizar estas dificultades, OSP ha desarrollado una serie de herramientas gratuitas de modelado, creación y análisis que se detallan a continuación:

- *Launcher - Paquetes de simulación*: El programa Launcher es una herramienta OSP que aborda el problema de cómo distribuir los programas de OSP y EJS, y los módulos curriculares.

  Launcher es una aplicación Java que puede lanzar (ejecutar) otros programas Java. Dicha herramienta se usa para distribuir las colecciones de programas listos para usar, documentación y material curricular de OSP y Ejs en un solo paquete fácilmente modificable. La entrega del material en paquetes tiene la ventaja de que dicho material es autónomo y sólo depende de

tener una máquina virtual Java en una máquina local y no del tipo de sistema operativo o navegador utilizado.

Los paquetes creados con Launcher se ejecutan haciendo doble clic en el archivo. Luego, Launcher carga automáticamente los programas y recursos especificados. El programa muestra las unidades usando una estructura de árbol para organizar el material de acuerdo al tema, curso, etc., como se muestra en la Figura 2.18. Los materiales también se pueden organizar en una o más pestañas, cada una de las cuales muestra un panel Explorador y un panel Descripción dentro de Launcher. El panel Explorador permite a los usuarios navegar fácilmente por el material basado en árboles y lanzar los programas asociados. La selección de un nodo de árbol (un solo clic) muestra su descripción de html o texto asociada en el panel Descripción [78].


Figura 2.18: El programa Launcher muestra la distribución del contenido curricular dado

- *Tracker - Análisis de video*: Tracker es un paquete de análisis de imágenes y videos, y una herramienta de modelado que se basa en la biblioteca de código Java de OSP. Las características de esta herramienta incluyen el seguimiento de objetos con superposiciones gráficos de la posición, velocidad y aceleración, filtros de efectos especiales, marcos de referencia múltiples, puntos de calibración y perfiles de línea para el análisis de espectros y patrones de interferencia. Está diseñado para ser usado en laboratorios introductorios de física universitaria y conferencias.

  Tracker puede superponer modelos de partículas dinámicas simples en un clip de vídeo. En un experimento típico de modelado de vídeo, los estudiantes capturan y abren un archivo de vídeo digital, calibran la escala y definen ejes de coordenadas apropiados al igual que para el análisis de video tradicional. Pero en lugar de rastrear objetos con el ratón, los estudiantes definen las expresiones de fuerza teóricas y las condiciones iniciales para una simulación de modelo dinámico que se sincroniza con el video y se dibuja en él. El comportamiento del modelo se compara así directamente con el del movimiento del mundo real.

Tracker utiliza la biblioteca de código abierto, por lo que es posible modelar otros modelos más sofisticados. El modelado de vídeo ofrece ventajas sobre el análisis de video tradicional y sobre el modelado mediante simulación[79].

En la Figura 2.19 se muestra un modelo dinámico en Tracker que permite a los estudiantes predecir dónde se debe apuntar para golpear a un mono que está cayendo en paracaídas.

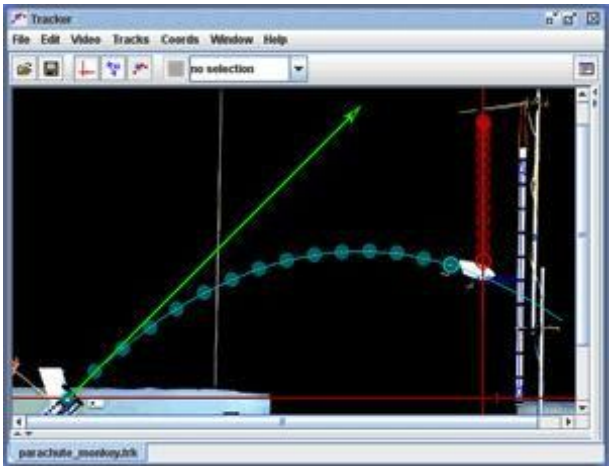



Figura 2.19: Modelo dinámico diseñado con la herramienta Tracker

- *EJS - Simulaciones fáciles de Java*: EJS es una herramienta de creación y modelado que permite a los usuarios crear programas en Java con una programación mínima (ver Figura 2.20).

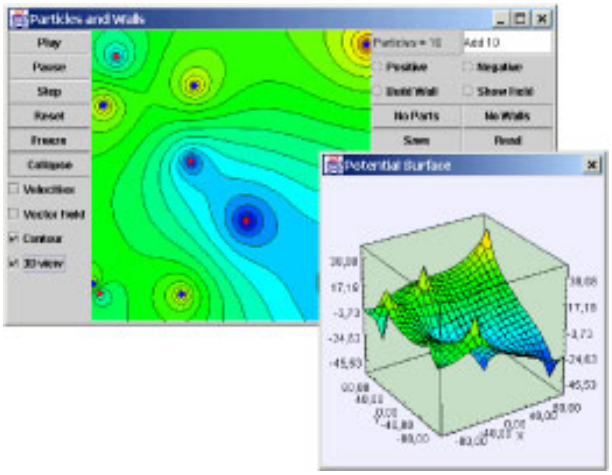



Figura 2.20: Software EJS

Ejs permite crear programas que otros usuarios pueden inspeccionar o modificar fácilmente y es una herramienta ideal para modelar, ya que permite a los usuarios:

- Desarrollar un prototipo de una aplicación para probar una idea o un algoritmo.
- Crear interfaces de usuario sin programación.
- Crear modelos cuya estructura y algoritmos pueden ser inspeccionados y comprendidos por usuarios que no son programadores.
- Crear sus propias simulaciones incluso a los usuarios que no tengan un alto conocimiento de Java.
- Crear rápidamente simulaciones que se distribuirán como applets o como programas autónomos.
- Crear un paquete que contenga varios programas y el material curricular asociado.

Esta herramienta se describe en detalle en la Sección 2.6.

- *Data tools - Análisis de datos*: DataTool es una herramienta de análisis de datos para trazar y ajustar datos del laboratorio, simulaciones, análisis de video, o cualquier otro conjunto de datos organizados en columnas. Un clic de una casilla de verificación en DataTool permite al usuario cambiar la apariencia de los gráficos, ver estadísticas estándar para el conjunto de datos o aplicar ajustes lineales, cuadráticos o cúbicos integrados al conjunto de datos. DataTool también incluye una serie de funciones matemáticas estándar que se pueden aplicar al conjunto de datos, lo que permite un análisis adicional y ampliar el rango de ajustes potenciales a los datos (ver Figura 2.21).

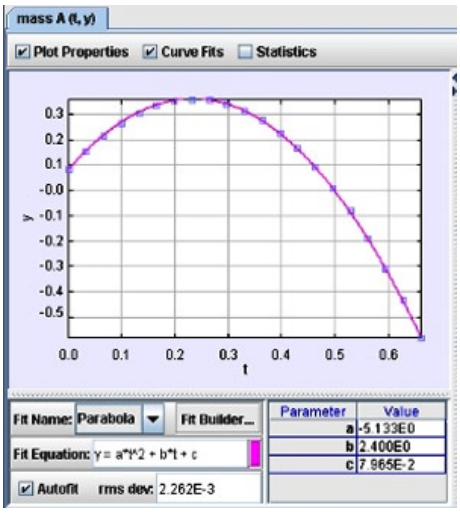



Figura 2.21: Data tools

Las características de DataTool incluyen:

- Cambiar la apariencia de las gráficas.

- Obtener estadísticas estándar para el conjunto de datos.
- Mostrar la pendiente (tangente) y el área bajo la curva.
- Ajustes automáticos de datos a expresiones analíticas predefinidas.
- Fit Builder: ajuste de datos a expresiones analíticas definidas por el usuario.
- Data Builder: aplica manipulaciones matemáticas estándar al conjunto de datos.

Aunque es un conjunto de herramientas independiente, DataTool se basa en la biblioteca de OSP y se integra en muchas de sus simulaciones y de EJS. Esto significa que un clic derecho en las gráficas importa automáticamente los datos en el DataTool para su posterior análisis. Esto hace que sea fácil analizar y comparar datos de simulaciones, análisis de vídeo (especialmente utilizando Tracker) y el laboratorio [80].

Además de estas herramientas, OSP proporciona otros amplios recursos para llevar a cabo simulaciones físicas:

- Un entorno Eclipse para OSP.
- Bibliotecas de código fuente OSP.
- Mejores prácticas de OSP.
- Documentación.

Finalmente, cabe destacar la existencia de una Guía de usuario con ejemplos escrita por Christian Wolfgang que proporciona una descripción de todo el proyecto de OSP y de diferentes programas diseñados con dicha librería [81]. En la Web principal de OSP podemos encontrar este manual como archivo de java (archivo jar). Dicho archivo está estructurado en diferentes capítulos tal y como se puede observar en la Figura 2.22) y contiene todos los programas listos para ejecutar con el fin de mostrar cómo usar la biblioteca OSP [82].

## 2.6. El software Easy Java Simulations

Easy Java Simulations es una herramienta de software gratuita, de código abierto, diseñada por el Doctor Francisco Esquembre de la Universidad de Murcia, para la creación de simulaciones interactivas por ordenador en Java o JavaScript de una manera simple, gráfica e intuitiva [83].

EJS es una herramienta de propósito general e independiente del campo de aplicación. Está enfocada al ámbito de la educación e investigación y ha sido diseñada principalmente para usuarios que no poseen altos conocimientos de programación. Un usuario utiliza la estructura de la interfaz

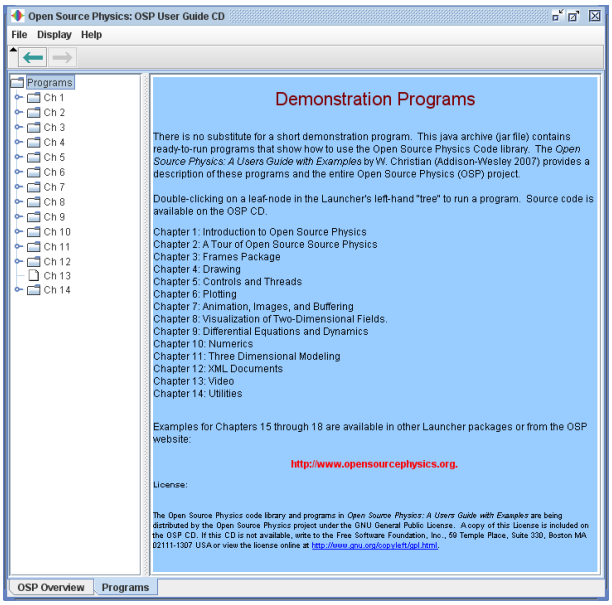



Figura 2.22: Guía de usuario de OSP

de EJS para crear el modelo de su simulación, proporcionando los algoritmos y la lógica, y para diseñar la interfaz gráfica apropiada para cumplir el propósito pedagógico de su simulación.

Dicha herramienta ha sido utilizada con éxito para la enseñanza a distancia [83, 84]; y para la creación de simulaciones virtuales en distintos ámbitos y disciplinas, tales como en Física [85, 86], Ingeniería [87, 88, 89, 90, 91], y también en Robótica [67, 92, 93].

Esta herramienta de simulación, así como su documentación y diversas simulaciones de ejemplo se encuentran disponibles de forma gratuita en su wiki: `http://www.um.es/fem/EjsWiki` [8].

La metodología de trabajo de EJS está basada en el paradigma *"modelo-vista-controlador"* (MVC) dado en la Figura 2.23. El MVC es un patrón de arquitectura de software que surge con el objetivo de reducir el esfuerzo de programación, necesario en la implementación de sistemas múltiples y sincronizados de los mismos datos, a partir de estandarizar el diseño de las aplicaciones. Este patrón separa los datos y la lógica delv modelo de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello el sistema MVC propone la construcción de tres componentes distintos: el *Modelo*, objeto que representa los datos del programa y controla todas sus transformaciones, la *Vista*, objeto que maneja la presentación visual de los datos representados por el Modelo, y el *Controlador*, objeto que proporciona significado a las órdenes del usuario, actuando sobre los datos representados por el Modelo y encargado de centrar la interacción entre la Vista y el Modelo [94, 95].

En resumen, el MVC es un paradigma basado en la idea de reutilización de código y separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior

mantenimiento .

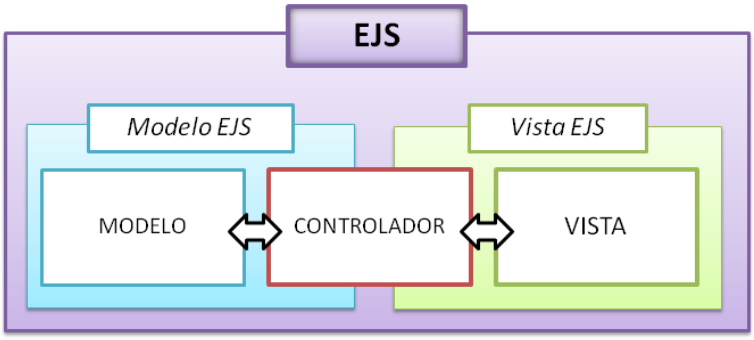



Figura 2.23: Paradigma MVC de EJS

Siguiendo el paradigma MVC, la interfaz de EJS ofrece al usuario tres paneles de trabajo para la creación de sus simulaciones (ver Figura 2.24).

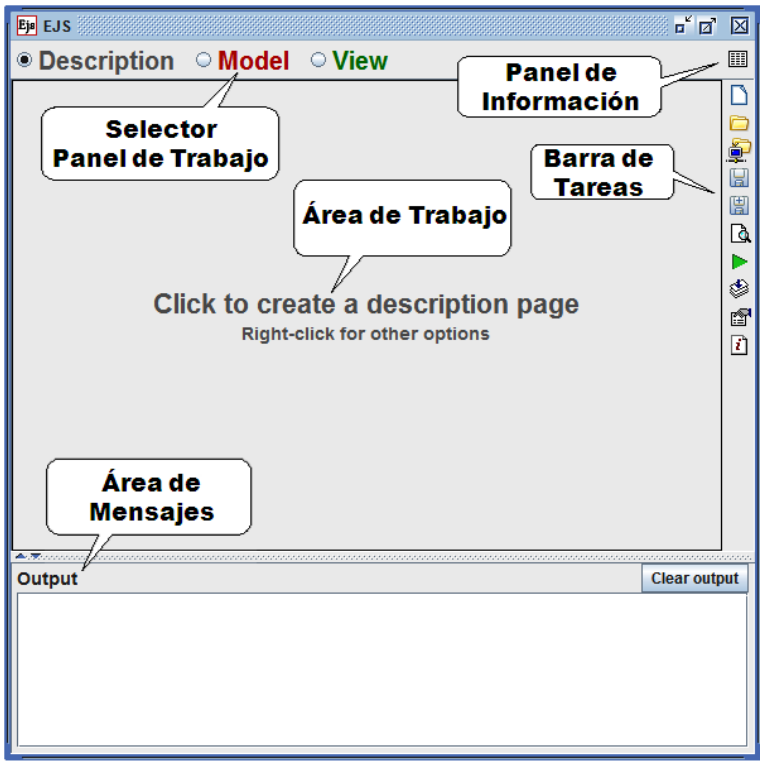



Figura 2.24: Interfaz de usuario de EJS

El primer panel de trabajo, *Description*, permite al usuario crear y editar páginas HTML que describan la simulación. Este texto se distribuye con la simulación final y facilita información de

cómo ejecutar dicha simulación o una guía para el usuario final.

El segundo panel de trabajo, *Model*, está dedicado a la definición del modelo de nuestra simulación. El autor usa este panel para definir las variables que describen el estado del modelo, el conjunto de los valores de estas variables para un correcto estado inicial, y para escribir los algoritmos y la lógica que describen cómo cambia el modelo en el tiempo y cómo responde a la interacción con el usuario. Normalmente, un usuario dedica la mayor parte del tiempo a la creación del modelo de la simulación.

El tercer panel de trabajo, *View*, está dedicado para la tarea de construcción de la interfaz gráfica de la simulación, la cuál permite al usuario controlar dicha simulación y visualizar el estado del modelo en dos o tres dimensiones. Las interfaces gráficas del usuario, o *vistas* en nuestra terminología, son creadas por el usuario de EJS mediante la combinación de elementos gráficos predefinidos que denominamos *elementos de la vista*, de una manera apropiada para el control y visualización del modelo.

Los elementos de la vista son de diferentes tipos, cada uno de ellos especializado en una visualización particular o en una tarea gráfica de entrada/salida de datos. En [96] se puede encontrar el detalle de todos estos elementos de la vista disponibles en EJS. Dichos elementos pueden ser personalizados y *conectados* con las variables del modelo de una manera muy sencilla. Esta conexión bidireccional permite que los cambios que se generen en el modelo sean automáticamente mostrados en la vista y, de forma análoga, la interacción del usuario con la vista también cambie automáticamente las correspondientes variables del modelo.

El control de la simulación queda implementado mediante este mecanismo de conexión simple, el cual facilita al usuario escribir poco código y le permite concentrarse en el dominio específico del modelo y en las propiedades pedagógicas de la vista.

Cada entorno de programación requiere del uso de librerías externas para la ejecución de tareas específicas, y el conocimiento de qué librerías existen y cómo usarlas (sus interfaces de programación o API) es normalmente una tarea difícil que requiere leer mucha documentación e incluir la librería correcta en cada fase de programación. Los elementos de la vista resuelven este problema para todas las tareas gráficas relativas a la interacción y visualización de datos científicos.

Para permitir el acceso al dominio específico de estas librerías, EJS proporciona una sección dentro del panel *Model* denominada *Elements* con un conjunto de librerías externas para diferentes tareas, encapsuladas según la orientación a objetos. De una manera similar a los elementos de la vista, los elementos del modelo vienen dados en una paleta de iconos, son añadidos al modelo arrastrando y soltando el icono correspondiente en la lista de elementos, y son personalizados haciendo doble click sobre ellos y editando las propiedades que aparecen en su correspondiente ventana emergente. Este simple proceso, transparente para el usuario, se encarga de incluir las librerías para las tareas requeridas, y de crear un objeto en tiempo de ejecución que encapsula un número de métodos de Java para las tareas que el elemento esté especializado. Además, cada

elemento dispone de una ventana de ayuda que describe sus principales características y documenta su API.

Los elementos del modelo integrados en EJS han sido desarrollados por diferentes autores y programadores, y proporcionan acceso a una gran variedad de tareas, incluyendo rutinas numéricas, programación en paralelo, acceso a otros programas o hardware, y ahora, con el desarrollo de esta tesis, también se han integrado elementos que dan acceso a robots, de forma virtual y remota (ver Sección 5.1).

En la Sección 5.2 se detalla el procedimiento a seguir para diseñar una simulación robótica en EJS y en la Sección 5.3 se facilita un ejemplo básico que puede servir como guía para la creación de cualquier tipo de simulación con el software EJS.

# Parte II

# Investigación y Diseño

# Capítulo 3

# Conceptualización de los elementos de un laboratorio virtual y remoto

Conceptualmente podemos considerar un robot como una máquina programable que puede manipular objetos y realizar operaciones que previamente sólo los humanos podían desempeñar. Para llevar a cabo la ejecución de estas tareas es necesario resolver algunas cuestiones básicas como son: el posicionamiento del robot, la definición de sus trayectorias, la especificación de restricciones de movimiento y la resolución de las posibles colisiones con el entorno o con él mismo.

En este capítulo se define con detalle qué es un robot manipulador y se analizan todas las características que hay que considerar a la hora de trabajar con ellos. De forma similar, se definen otros componentes robóticos y se plantean otros aspectos adicionales que también resultan necesarios a la hora de modelar un laboratorio robótico virtual o remoto.

## 3.1. Conceptualización de robots manipuladores

A lo largo de esta tesis se ha destacado la importancia que tiene la Robótica hoy en día tanto en el ámbito de la investigación como de la industria. Además se ha hecho hincapié en el uso de las simulaciones robóticas como el medio idóneo para analizar los comportamientos complejos que pueden presentarse a la hora de trabajar con sistemas robóticos.

La aparición del computador, junto con el desarrollo tecnológico que se experimenta en la segunda mitad del siglo XX, suponen el comienzo de la robótica industrial moderna tal y como se conoce hoy en día.

La palabra "robot" ha tenido un origen distinto al dispositivo electromecánico. La primera constancia que se tiene de ella, con el significado actual, data de principios del siglo XX.

El robot es capaz de realizar de modo automático gran cantidad de actividades dentro de la industria, como: manipulación, pintura, soldadura, etc. Su flexibilidad es mayor que la de cualquier máquina y su productividad superior a la de cualquier ser humano en gran cantidad de trabajos. Un robot tiene la capacidad de procesamiento inferior a la de un ser humano, pero puede realizar operaciones limitadas de forma más rápida y precisa. Además, puede trabajar en condiciones adversas para un ser humano, y sobre todo trabajar de modo continuo [97].

Los robots se pueden clasificar de múltiples formas en función de sus diferentes características y parámetros, por ejemplo en: robots manipuladores, robots de servicios, robots móviles, robots de investigación, robots médicos,... [98]. De todos estos grupos, esta tesis se centra en el estudio de robots manipuladores, en establecer una conceptualización y estudiar con detalle todos los aspectos que hay que considerar cuando se emplean en nuestros laboratorios robóticos.

A la hora de establecer una definición formal de lo que es un robot industrial nos encontramos con ciertas dificultades. La primera de ellas surge de la diferencia conceptual entre el mercado japonés y el euro-americano de lo que es un robot y lo que es un manipulador. Así, 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. En segundo lugar, y centrándose ya en el concepto occidental, no es sencillo establecer una definición formal, ya que la evolución de la robótica ha ido obligando a realizar diferentes actualizaciones de su definición [99].

La definición comúnmente aceptada es la de la Asociación de Robótica Industrial (RIA) [100], según la cual:

*"Un robot industrial es un manipulador multifuncional reprogramable, capaz de mover materias, piezas, herramientas, o dispositivos especiales, según trayectorias variables, programadas para realizar tareas diversas."*

Esta definición ha sido ligeramente modificada por la Organización Internacional de Normalización (ISO) [101]:

*"Un robot industrial es un manipulador multifuncional reprogramable con varios grados de libertad, capaz de manipular materias, piezas, herramientas o dispositivos especiales según trayectorias variables programadas para realizar tareas diversas."*

En esta definición se incluye la necesidad de que el robot tenga varios grados de libertad (GDL). Se denomina GDL a la suma del número de movimientos independientes que puede realizar cada articulación respecto a la anterior [102]. En general, el número de GDL en un robot es igual al número de sus articulaciones o al número de sus ejes.

Por último, la Federación Internacional de Robótica (IFR) [103], establece la siguiente definición:

*"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."*

En general, los robots manipuladores son sistemas mecánicos multifuncionales, con un sencillo sistema de control, que permite gobernar el movimiento de sus elementos, de los siguientes modos:

- Manual: cuando el usuario controla directamente la tarea del manipulador.
- De secuencia fija: cuando se repite, de forma invariable, el proceso de trabajo preparado previamente.
- De secuencia variable: cuando se pueden alterar algunas características de los ciclos de trabajo.

Por lo tanto, a la hora de trabajar con un robot manipulador además de conocer sus especificaciones propias (dimensiones, GDL, ...) hay que considerar lo siguiente: cómo se lleva a cabo su posicionamiento, es decir, la resolución de sus cinemáticas, cómo se establecen sus trayectorias y restricciones de movimiento, así como detectar sus posibles colisiones. En las siguientes subsecciones se va a tratar en detalle cada una de ellas.

### 3.1.1. Posicionamiento

En robótica resulta imprescindible poder representar las posiciones y orientaciones en el espacio del robot ya que este necesita localizarse (posicionarse y orientarse) adecuadamente en el espacio para llevar a cabo cualquier tarea [104].

El posicionamiento del robot con respecto a un sistema de referencia se lleva a cabo mediante la resolución de su cinemática. Dicha posición puede venir expresada en coordenadas articulares o cartesianas. Dado un robot de n GDL, sus *coordenadas articulares* vendrán definidas mediante el vector $q = (q_1, \ldots, q_n)$, donde cada $q_i$ representa el valor que cada una de las articulaciones del robot debe tomar para llegar a esa localización final. Las *coordenadas cartesianas* son las proyecciones del efector final del robot respecto a un sistema de referencia dado $p = (x, y, z, r_x, r_y, r_z)$. Las primeras tres coordenadas corresponderían a la posición del robot y el resto a su orientación.

Dependiendo de si la posición del robot viene dada en coordenadas articulares o cartesianas, la resolución de su cinemática plantea dos problemas fundamentales (Figura 3.1): la *cinemática directa* y la *cinemática inversa*.

Ambos problemas presentan una gran dependencia del propio robot, pues la complejidad de su resolución depende básicamente de sus GDL y su geometría.

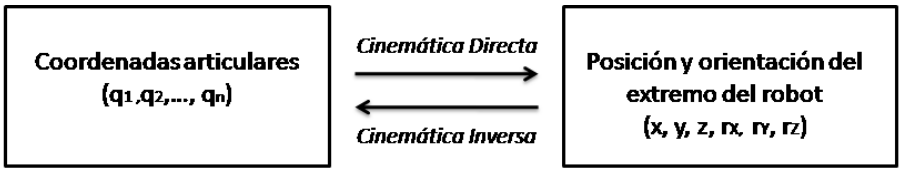



Figura 3.1: Relación entre las cinemáticas de un robot

#### 3.1.1.1. Cinemática directa

La cinemática directa consiste en el cálculo de la posición y orientación del efector final del robot, respecto a un sistema de referencia, conociendo los valores que deben tomar cada una de las articulaciones del robot (posición articular). En la bibliografía se pueden encontrar diversos métodos para su resolución, entre los que podemos destacar los siguientes [102]:

- *Métodos geómetricos*:

  El método geométrico consiste en establecer simples relaciones geométricas que permiten conocer la localización espacial del extremo del robot a partir de los valores de sus articulaciones. Es un método sencillo para robots de pocos GDL, tal y como se puede observar en la Figura 3.2, donde se facilitan las relaciones obtenidas para un robot planar de 2GDL. Sin embargo, este método no tiene un procedimiento específico para resolver el problema planteado, resultando inoperativo para robots de muchos grados de libertad.

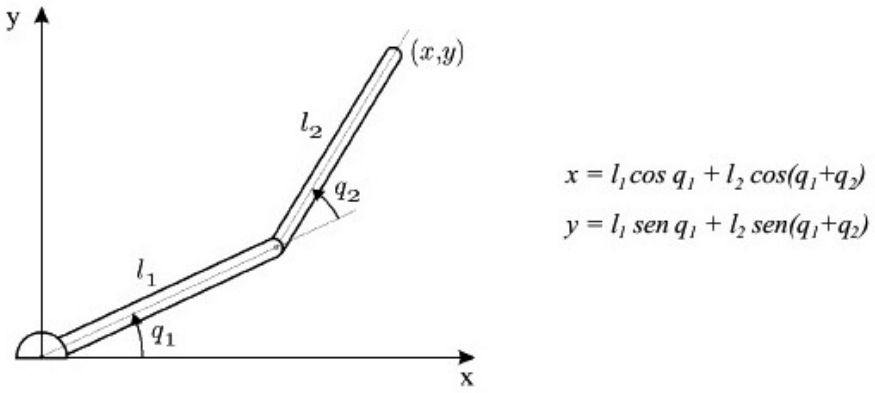



Figura 3.2: Resolución cinemática directa de un robot de 2 GDL por el método geométrico

- *Métodos basados en matrices de transformación homogénea*:

  En este método se utiliza, fundamentalmente, el álgebra vectorial y matricial para representar la localización de un objeto en el espacio tridimensional. Se trata de encontrar una matriz de

transformación homogénea T que relacione la posición y orientación del robot respecto a un sistema de referencia fijo.

Dado un robot de n GDL, se define ${}^{i-1}A_i$ como la matriz de transformación homogénea que representa la posición y orientación relativa entre los sistemas asociados a dos eslabones consecutivos del robot (ver Figura 3.3). De manera análoga, se define como ${}^{0}A_k$ a las matrices resultantes del producto de matrices ${}^{i-1}A_i$ con i desde 1 hasta k.

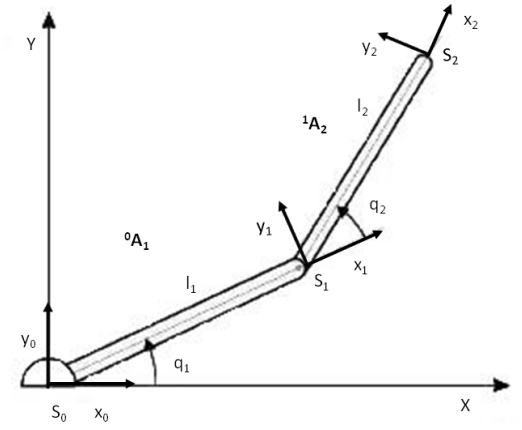



Figura 3.3: Relación entre los sistemas de referencia de los eslabones del robot

Entonces, la matriz de transformación T se define como:

$$T = {}^{0}A_n = {}^{0}A_1 \, {}^{1}A_2 \, {}^{2}A_3 \, \ldots \, {}^{n-1}A_n \tag{3.1}$$

Las matrices ${}^{i-1}A_i$, además, dependen de las constantes geométricas propias del eslabón, y de $q_i$. Por lo tanto, la Ecuación 3.1 quedaría de la siguiente manera:

$$T(q_1, \ldots, q_n) = {}^{0}A_1(q_1) \, {}^{1}A_2(q_2) \, {}^{2}A_3(q_3) \, \ldots \, {}^{n-1}A_n(q_n) \tag{3.2}$$

- *Método de Denavit-Hartenberg (DH)*:

  El método de DH es ampliamente utilizado en el ámbito académico y de investigación en robótica, y permite definir las transformaciones relativas entre eslabones con tan sólo cuatro parámetros, siendo éste el número mínimo de parámetros para configuraciones genéricas (ver la Figura 3.4) [105].

  Dado el eslabón i-ésimo del robot se definen los cuatro parámetros de DH ($\theta_i$, $d_i$, $a_i$, $\alpha_i$) como:

  - $\theta_i$: ángulo que forman los ejes $x_{i-1}$ y $x_i$ medido en un plano perpendicular al eje $z_{i-1}$, según la regla de la mano derecha.

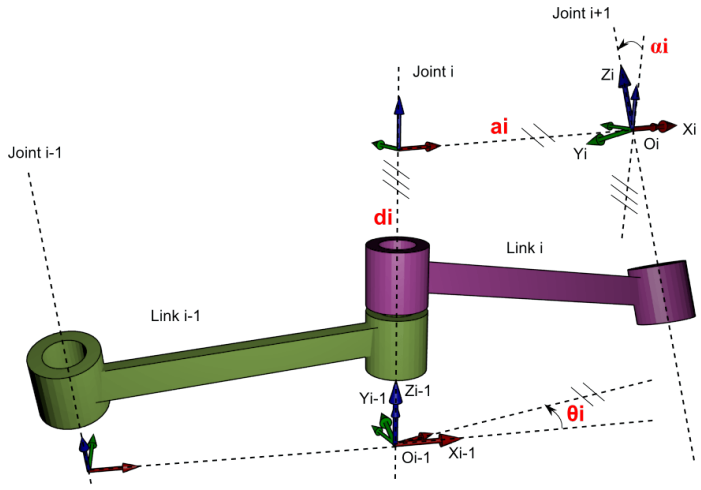



Figura 3.4: Parámetros de Denavit-Hartenberg

- $d_i$: distancia a lo largo del eje $z_{i-1}$ desde el origen del sistema de coordenadas (i-1)-ésimo hasta la intersección del eje $z_{i-1}$ con el eje $x_i$.
- $a_i$: distancia a lo largo del eje $x_i$ que va desde la intersección del eje $z_{i-1}$ con el eje $x_i$ hasta el origen del sistema i-ésimo, en el caso de las articulaciones giratorias. En las articulaciones prismáticas, se calcula como la mínima distacia entre los ejes $z_{i-1}$ y $z_i$.
- $\alpha_i$: ángulo de separación del eje $z_{i-1}$ y el $z_i$, medido en un plano perpendicular al eje $x_i$, usando la regla de la mano derecha.

Estos parámetros dependen únicamente de las características propias del robot, pues relacionan los sistemas de referencia de dos eslabones consecutivos mediante la siguiente matriz de transformación homogénea:

$$ {}^{i-1}A_i = \begin{bmatrix} \cos\theta_i & -\cos\alpha_i \sin\theta_i & \sin\alpha_i \sin\theta_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\alpha_i \cos\theta_i & -\sin\alpha_i \cos\theta_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3} $$

Una vez obtenidas todas las matrices de transformación ${}^{i-1}A_i$, se define la matriz T como el producto en cadena de todas ellas, tal y como se definió anteriormente en la Ecuación 3.2. Dicha matriz define la orientación (submatriz $3 \times 3$) y posición (submatriz $3 \times 1$) del extremo del robot, en función de las coordenadas articulares.

En resumen, el algoritmo de DH consta de los siguientes pasos:

- Numerar los eslabones comenzando con 1 (primer eslabón móvil de la cadena) y acabando con n (el último eslabón móvi).

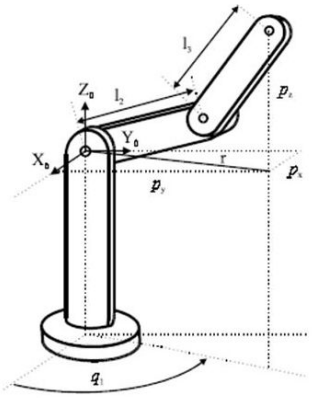
- Numerar cada articulación comenzando con 1 (la correspondiente al primer grado de libertad) y acabando en n.
- Localizar el eje de cada articulación. Si es rotativa, el eje será su propio eje de giro. Si es prismática, será el eje a lo largo del cual se produce el desplazamiento.
- Para i de 0 a n-1, colocar el eje $z_i$ sobre el eje de la articulación i+1.
- Situar el origen del sistema de la base $S_0$ en cualquier punto del eje $z_0$. Los ejes $x_0$ e $y_0$ se situarán de modo que formen un sistema dextrógiro con $z_0$.
- Para i de 1 a n-1, situar el origen del sistema $S_i$ (solidario al eslabón i) en la intersección del eje $z_i$ con la línea de la común a $z_{i-1}$ y $z_i$. Si ambos ejes se cortasen se situaría $S_i$ en el punto de corte. Si fuesen paralelos $S_i$ se situaría en la articulación i+1.
- Situar $x_i$ en la línea normal común a $z_{i-1}$ y $z_i$.
- Situar $y_i$ de modo que forme un sistema dextrógiro con $x_i$ y $z_i$.
- Situar el sistema $S_n$ en el extremo del robot de modo que $z_n$ coincida con la dirección de $z_{n-1}$ y $x_n$ sea normal a $z_{n-1}$ y $z_n$.
- Obtener $\theta_i$ como el ángulo que hay que girar en torno a $z_{i-1}$ para que $x_{i-1}$ y $x_i$ queden paralelos.
- Obtener $d_i$ como la distancia, medida a lo largo de $z_{i-1}$, que habría que desplazar $S_{i-1}$ para que su origen coincidiese con $S_i$.
- Obtener $a_i$ como la distancia medida a lo largo del eje $x_i$ que habría que desplazar el nuevo $S_{i-1}$ para que su origen coincidiese con $S_i$.
- Obtener $\alpha_i$ como el ángulo que habría que girar entorno a $x_i$, para que el nuevo $S_{i-1}$ para que su origen coincidiese con $S_i$.
- Obtener las matrices de transformación ${}^{i-1}A_i$ dadas en la Ecuación 3.3.
- Obtener la matriz de transformación que relaciona el sistema de la base con el del extremo del robot T definida en la Ecuación 3.2.
- La matriz T define la orientación y posición del extremo referido a la base, en función de las n coordenadas articulares.

#### 3.1.1.2. Cinemática inversa

La cinemática inversa, por el contrario, parte de la localización del efector final del robot, en coordenadas cartesianas, para hallar los valores que deben tomar cada una de las articulaciones para posicionar y orientar el extremo del robot en dicha posición. La resolución de esta cinemática puede plantear mayor dificultad que la cinemática directa al tener una fuerte dependencia de la configuración del robot y al no poseer una solución única. En algunas ocasiones, para un mismo

punto existirán múltiples soluciones mientras que en otros casos será imposible hallar alguna. Para resolver este problema se puede optar por dos vías: solución numérica o solución cerrada. La primera posibilidad se descarta ya que, por regla general, resulta más lenta y costosa computacionalmente que una solución cerrada. La solución cerrada intenta buscar una solución basada en expresiones analíticas o polinómicas, que no hagan uso de cálculos iterativos [97]. Se pueden destacar los siguientes métodos:

- *Métodos geométricos:* Los métodos geométricos permiten, normalmente, obtener los valores de las primeras variables articulares, que son las que consiguen posicionar el robot (prescindiendo de la orientación de su extremo). Para ello, utilizan relaciones trigonométricas y geométricas sobre los elementos del robot. En la Figura 3.5 se presenta la resolución de la cinemática inversa para un robot articular de 3 GDL.




$$q_1 = \text{arctg}\left(\frac{p_y}{p_x}\right)$$

$$\left.\begin{aligned} r^2 &= p_x^2 + p_y^2 \\ r^2 + p_z^2 &= l_2^2 + l_3^2 + 2l_2 l_3 \cos q_3 \end{aligned}\right\} \Rightarrow$$

$$\cos q_3 = \frac{p_x^2 + p_y^2 + p_z^2 - l_2^2 - l_3^2}{2l_2 l_3}$$

$$\text{sen}\, q_3 = \pm\sqrt{1 - \cos^2 q_3}$$

$$q_3 = \text{arctg}\left(\frac{\pm\sqrt{1-\cos^2 q_3}}{\cos q_3}\right)$$

$$\text{con} \quad \cos q_3 = \frac{p_x^2 + p_y^2 + p_z^2 - l_2^2 - l_3^2}{2l_2 l_3}$$

Figura 3.5: Resolución cinemática inversa de un robot de 3 GDL por el método geométrico

- *Matriz de transformación homogénea*:

  Partiendo de las relaciones que expresan el valor de la posición y orientación del extremo del robot en función de sus coordenadas articulares, se pueden obtener mediante manipulación las relaciones inversas. En la práctica no es una tarea sencilla, siendo a veces tan compleja que obliga a dejarla.

  El primer paso que se debe realizar es obtener la matriz T que relaciona el sistema de referencia $S_0$ asociado a la base con el sistema de referencia $S_n$ dada en la Ecuación 3.2. Considerando los valores de los parámetros de D-H y su asignación de sistemas de referencia, se pueden obtener las matrices A y la matriz T. Obtenida la expresión de T en función de las coordenadas articulares $(q_1,\ldots,q_n)$, y supuesta una localización para el extremo del robot, se puede resolver dicho problema mediante cálculo matricial.

- *Desacoplo cinemático*: El método de desacoplo cinemático es aplicable a aquellos robots cuyos tres últimos grados de libertad se cortan en un punto, separando los problemas de obtención del modelo cinemático inverso de posición y orientación. Para ello, dada una posición y orientación final deseadas, establece las coordenadas del punto de corte de los 3 últimos ejes calculándose los valores de las tres primeras variables articulares ($q_1$, $q_2$, $q_3$) que consiguen posicionar este punto. A continuación, a partir de los datos de orientación deseada para el extremo y los valores calculados de ($q_1$, $q_2$, $q_3$) se obtiene el resto de las articulaciones.

Además de estos algoritmos generales, en la bibliografía se pueden encontrar trabajos dedicados a la resolución de la cinemática inversa aplicando otras técnicas o enfocados en algún tipo de robot específico. De estre todos estos, en esta tesis se destacan dos de ellos que han servido de base para el diseño de la cinemática inversa de la librería presentada (ver Sección 4.3.1.2).

- En [106] se plantea un algoritmo para la resolución de la cinemática inversa basada en el conocimiento y métodos del Álgebra Lineal. El objetivo de este método es automatizar el cálculo de las soluciones de la cinemática directa e inversa para robots industriales manipuladores en general. Para un robot manipulador con geometría simple se utilizan técnicas de programación basadas en métodos comunes, como por ejemplo, el método basado en la matriz de transformación homogénea descrito anteriormente, y obtener así una solución cerrada.

  Para geometrías más complicadas, utiliza técnicas de eliminación basadas en el cálculo de los principios de Groebner (algoritmo para simplificar los cálculos). De este modo, convertimos un conjunto complicado de ecuaciones de la cinemática en polinomios de una sola variable mucho más sencillos de resolver. En resumen, el procedimiento para la solución de la cinemática inversa parte de las mismas expresiones obtenidas de la cinemática directa (ver Figura 3.6(a)).

$$T_{i-1,i} = \begin{pmatrix} \cos\theta_i & -\cos\alpha_i \sin\theta_i & \sin\alpha_i \sin\theta_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\alpha_i \cos\theta_i & -\sin\alpha_i \cos\theta_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_{0,N} = T_{0,1} * T_{1,2} * \ldots * T_{n-1,n} = \prod_{i=1}^{N} T_{i-1,i} = R$$

(a) Matrices de DH

$$\begin{aligned}
T_{01} * T_{12} * T_{23} * T_{34} &= R \\
T_{01} * T_{12} * T_{23} &= R * T_{43} \\
T_{01} * T_{12} &= R * T_{43} * T_{32} \\
T_{01} &= R * T_{43} * T_{32} * T_{21} \\
T_{12} * T_{23} * T_{34} &= T_{10} * R \\
T_{12} * T_{23} &= T_{10} * R * T_{43} \\
T_{12} &= T_{10} * R * T_{43} * T_{32} \\
T_{23} * T_{34} &= T_{21} * T_{10} * R \\
T_{23} &= T_{21} * T_{10} * R * T_{43} \\
T_{34} &= T_{32} * T_{21} * T_{10} * R
\end{aligned}$$

(b) Sistema de ecuaciones final

Figura 3.6: Ecuaciones para determinar la cinemática inversa con solución cerrada

Reescribiendo dichas ecuaciones y multiplicando ambos lados de la ecuación matricial de la cinemática directa por matrices de transformación inversa como se muestra en la Figura 3.6(b)

generamos ecuaciones equivalentes que tienen las mismas soluciones, pero son más fáciles de resolver. Cada ecuación matricial proporciona 12 ecuaciones no lineales derivadas de las tres filas superiores de la matriz. La cuarta fila de la matriz siempre es trivial.

El problema principal ahora es encontrar ecuaciones adecuadas que puedan ser resueltas analíticamente. Para ello, se facilita una colección de reglas sobre soluciones de las ecuaciones trigonométricas. Cada regla tiene una estructura uniforme y es un elemento independiente de conocimiento. La operación consiste en asociar una solución con una variable de una de las articulaciones desconocidas y modificar las ecuaciones de la cinemática (resumidas en la Figura 3.6).

- En [107] se presenta una nueva formulación para la resolución de la cinemática inversa de un robot Scara mediante grupos de Assur. Dicha formulación minimiza el tiempo de resolución del problema cinemático y permite definir, de manera inequívoca, el modo de trabajo del robot en cualquier instante de la simulación. También permite identificar posibles configuraciones singulares del problema inverso.

  Denominamos grupo estructural o grupo de Assur a aquella cadena cinemática con un número de grados de libertad igual a cero con respecto a los eslabones con los cuales sus elementos libres "entran" en pares cinemáticos y que no se puede dividir en cadenas cinemáticas más sencillas con número de grados de libertad igual a cero [108].

  Los grupos de Assur son bien conocidos en cinemática aunque su uso en cinemática computacional no está muy extendido. Además, sus formulaciones cinemáticas suelen estar restringidas a mecanismos planos.

  El grupo de Assur formado por dos eslabones y tres partes de revolución se denomina RRR y es muy común en la cinemática plana (aplicable por ejemplo, al robot Scara). En la Figura 3.7 se muestran sus dos modos de ensamblaje.

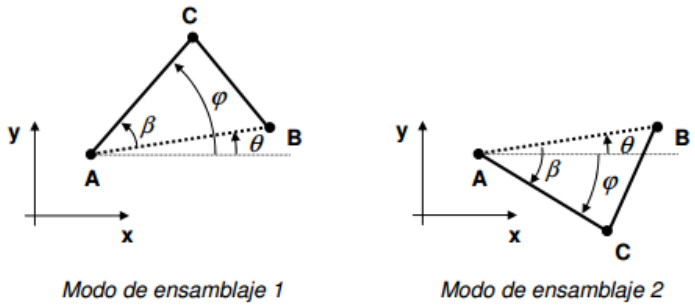



Figura 3.7: Modo de ensamblaje del grupo RRR

La resolución de la cinemática del grupo RRR consiste en calcular la posición del punto C ($x_c$

e $y_c$), su velocidad ($\dot{x_c}$ e $\dot{y_c}$) y aceleración ($\ddot{x_c}$ e $\ddot{y_c}$) así como el ángulo formado por el eslabón AC con el semieje X positivo ($\varphi$) y sus dos primeras derivadas temporales ($\dot{\varphi}, \ddot{\varphi}$) a partir de la posición, velocidad y aceleración de los puntos A y B, las longitudes $l_{AC}$ y $l_{BC}$ y el modo de ensamblaje deseado.

### 3.1.2. Planificación de trayectorias

Una vez estudiado el problema del posicionamiento del robot, es evidente que otra de las finalidades que tienen los robots es poder alcanzar el objeto que se desea para su manipulación o ejecutar ciertos movimientos para alcanzar alguna posición definida. La importancia de la planificación de trayectorias radica en la búsqueda y obtención de estrategias de control para obtener del robot trayectorias adecuadas, seguras y que posean la mayor calidad en su desplazamiento [109].

Una trayectoria se puede definir como una colección de puntos (en coordenadas articulares o cartesianas) que el robot debe seguir en un orden dado, y en un tiempo específico. Para definir una trayectoria es necesario especificar al menos el punto inicial, el punto final y la duración de la misma. Además de estos datos, existen algunos tipos de trayectorias más complejas que permiten establecer puntos intermedios por donde debe pasar el robot en un instante específico.

En la literatura se pueden encontrar diversos tipos de interpoladores para la planificación de diferentes trayectorias. De entre todos estos los más comunes y destacables son los siguientes [102]:

- **Interpolador Lineal**: Una primera interpolación consistiría en que cada articulación pase sucesivamente por los valores $q_i$ en los diferentes instantes $t_i$. Esta solución mantiene la velocidad constante entre cada dos valores sucesivos de las articulaciones, asegurando la continuidad de la posición, pero puede originar saltos bruscos en la velocidad de la articulación y aceleraciones a priori infinitas tal y como se puede observar en la Figura 3.8.

  La trayectoria entre dos puntos ($q_i$ , $q_{i+1}$) vendría dada por la siguiente expresión:

$$\begin{aligned} &q(t) = (q_i - q_{i-1})\tfrac{t-t_{i-1}}{T} + q_{i-1} \quad t_{i-1} < t < t_i \\ &T = t_i - t_{i-1} \end{aligned} \tag{3.4}$$

- **Interpolador Spline Cúbico**: Este tipo de interpolador es muy común y enlaza cada punto con el siguiente mediante un polinomio de tercer grado. De esta forma, a diferencia del interpolador lineal y como se puede observar en la Figura 3.9, dicho interpolador asegura la continuidad tanto en la posición como en la velocidad en los puntos de unión. Para una Spline cúbica, además del punto inicial y final de la trayectoria, es necesario especificar al menos un punto intermedio de la misma y el instante en el que queremos que pase por él.

  La expresión que tendría dicha trayectoria entre dos puntos ($q_i$ , $q_{i+1}$) vendría dada por la

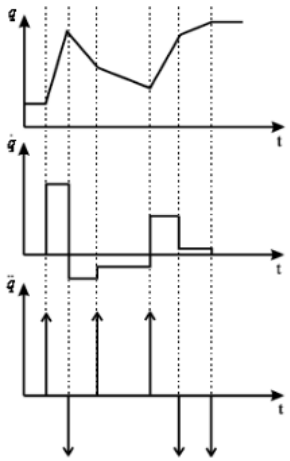



Figura 3.8: Posición, velocidad y aceleración para un interpolador lineal

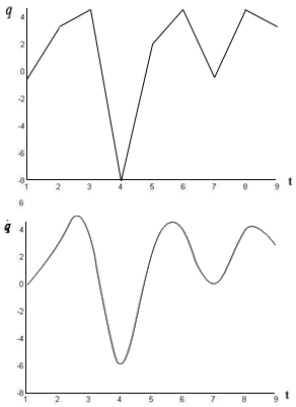



Figura 3.9: Posición y velocidad de un interpolador Spline cúbico

siguiente expresión:

$$
\begin{aligned}
&q(t) = a + b(t - t_i) + c(t - t_i)^2 + d(t - t_i)^3 \quad t_i < t < t_{i+1} \\
&a = q_i \\
&b = \dot{q}_i \\
&c = \tfrac{3}{T^2}(\dot{q}_{i+1} - \dot{q}_i) - \tfrac{1}{T}(\dot{q}_{i+1} - 2\dot{q}_i) \\
&d = \tfrac{2}{T^3}(\dot{q}_{i+1} - \dot{q}_i) + \tfrac{1}{T^2}(\dot{q}_{i+1} + \dot{q}_i) \\
&T = t_{i+1} - t_i
\end{aligned}
\tag{3.5}
$$

Para poder calcular los valores de los coeficientes del polinómio cúbico de la Ecuación 3.5 es necesario conocer a priori las velocidades de paso de cada punto $\dot{q}_i$. Para ello existen diferentes alternativas:

El primer criterio para seleccionar dichas velocidades de paso vendría dado por la Ecuación 3.6.

$$\dot{q}_i = \begin{cases} 0 & si\ signo(q_i - q_{i-1}) \neq signo(q_{i+1} - q_i) \\ \frac{1}{2}[\frac{q_{i+1}-q_i}{t_{i+1}-t_i} + \frac{q_i-q_{i-1}}{t_i-t_{i-1}}] & si \begin{cases} signo(q_i - q_{i-1}) = signo(q_{i+1} - q_i) \\ o\ q_{i-1} = q_i \\ o\ q_i = q_{i+1} \end{cases} \end{cases} \tag{3.6}$$

Es un criterio sencillo de resolver y da como resultado una continuidad razonable en la velocidad. Sin embargo, no establece condición sobre la continuidad de la aceleración.

El segundo criterio consiste en escoger las velocidades de paso de modo que cada spline cúbica sea continua en posición, velocidad y aceleración con los dos polinomios adyacentes. Las velocidades de paso se calcularían con el sistema de ecuaciones lineales de diagonal dominante dado en la Ecuación 3.7:

$$\begin{bmatrix} t_3 & 2(t_2+t_3) & t_2 & 0 & 0 & \dots \\ 0 & t_4 & 2(t_3+t_4) & t_3 & 0 & \dots \\ 0 & 0 & t_5 & 2(t_4+t_5) & t_4 & \dots \\ \dots & \dots & 0 & t_6 & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix} \cdot \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \vdots \\ \dot{q}_k \end{bmatrix} = \begin{bmatrix} \frac{3}{t_2 t_3}[(t_2)^2(q_3-q_2) + (t_3)^2(q_2-q_1)] \\ \frac{3}{t_3 t_4}[(t_3)^2(q_4-q_3) + (t_4)^2(q_3-q_2)] \\ \vdots \\ \frac{3}{t_{k-1} t_k}[(t_{k-1})^2(q_k-q_{k-1}) + (t_k)^2(q_{k-1}-q_{k-2})] \end{bmatrix} \tag{3.7}$$

Este sistema tiene k-2 ecuaciones y k incógnitas, las distintas velocidades de paso por los k puntos. Para que el sistema esté definido se considera además que la velocidad inicial y final de la articulación es nula:

$$\dot{q_1} = \dot{q_k} = 0 \tag{3.8}$$

- **Interpolador Trapezoidal**: Este interpolador se caracteriza por descomponer la trayectoria en tres tramos. En los tramos inicial y final se utiliza un polinomio de segundo orden de modo que la velocidad varía linealmente. En el tramo 1 la velocidad varía desde la velocidad de la trayectoria anterior a la de la presente y en el tramo 3 varía desde la velocidad de la trayectoria presente hasta la de la siguiente. En el tramo central, se utiliza un interpolador lineal de manera que la velocidad se mantiene constante y la aceleración nula.

  En la Figura 3.10 se puede observar la variación de la posición, la velocidad y la aceleración en cada uno de dichos tramos.

  La expresión de la trayectoria trapezoidal en el caso simple de una trayectoria con dos únicos puntos ($q_0$ , $q_1$) con velocidad inicial y final nula vendría dada por la siguiente expresión:

$$q(t) = \begin{cases} q_0 + s\frac{a}{2}t^2 & t \leq \tau \\ q_0 - s\frac{V^2}{2a} + sVt & \tau < t \leq T - \tau \\ q_1 + s(-\frac{aT^2}{2} + aTt - \frac{a}{2}t^2) & T - \tau < t < T \end{cases} \tag{3.9}$$

siendo:

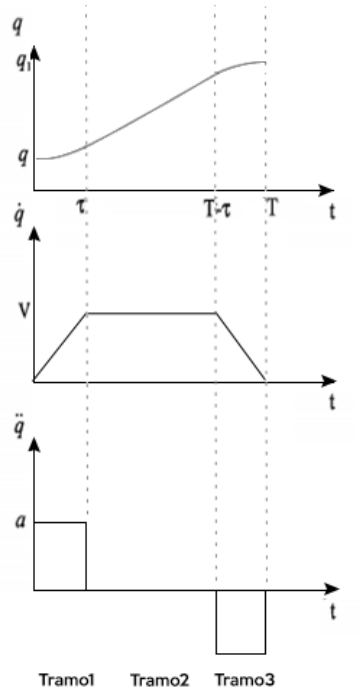



Figura 3.10: Posición, velocidad y aceleración de un interpolador Trapezoidal

- $\tau = \frac{V}{a}$
- T $= s\frac{q_1 - q_0}{V} + \frac{V}{a}$
- V : velocidad máxima permitida
- a : aceleración máxima permitida
- s : $signo(q_1 - q_0)$

En el caso en el que la trayectoria pase por varios puntos concretos $q_0$, $q_1$ y $q_2$ en los instantes t = 0, t = $T_1$, y t =$T_1 + T_2$, respectivamente, dicha expresión quedaría de la siguiente manera:

$$q(t) = \begin{cases} q_0 + \frac{q_1 - q_0}{T_1} t & 0 \leq t \leq T_1 - \tau \\ q_1 + \frac{q_1 - q_0}{T_1}(t - T_1) + \frac{a}{2}(t - T_1 + \tau)^2 & T_1 - \tau < t < T_1 + \tau \\ q_1 + \frac{q_2 - q_1}{T_2}(t - T_1) & T_1 + \tau < t < T_1 + T_2 \end{cases} \tag{3.10}$$

donde:

- a $= \frac{T_1(q_2 - q_1) - T_2(q_1 - q_0)}{2T_1 T_2 \tau}$
- $2\tau$ : Tiempo empleado para variar la velocidad del movimiento

### 3.1.3. Restricciones y detección de colisiones

Para poder llevar a cabo la planificación de trayectorias resulta fundamental conocer o ser capaz de obtener tanto el modelo cinemático (detallada en la Subsección 3.1.1) como el dinámico del robot que se pretenda controlar. La trayectoria posible depende de las características físicas del robot así como de sus articulaciones, porque solo será posible realizar aquellos movimientos alcanzables por el robot y que no requieran de desplazamientos mas allá de sus límites establecidos [109]. Además, dependiendo de las características de las operaciones definidas por el usuario (destino, tipo de trayectoria, tiempo en que se desee que se ejecute el movimiento,...) y la aplicación concreta, será proyectado el trabajo, siendo en ocasiones necesario considerar otras restricciones de tipo de movimiento (suave, rudo), etc.

Por lo tanto, cuando operamos con un robot sus movimientos están condicionados a ciertas limitaciones, algunas debidas a la propia configuración del robot (restricciones físicas de sus motores y articulaciones), y otras que surgen del espacio de trabajo (limitaciones del entorno donde se mueve el robot) o de la operación que se quiere llevar a cabo (como por ejemplo el movimiento de un objeto frágil).

Las restricciones físicas son embebidas generalmente en los controladores de los robots, de manera que el robot opera siempre sin exceder sus límites establecidos para sus articulaciones, velocidad y aceleración. Si se intenta exceder alguno de éstos, generalmente el robot se detendrá y no completará dicho movimiento.

Sin embargo, si durante la ejecución de las operaciones el movimiento del robot está limitado por el entorno que lo rodea, como pueden ser las paredes del laboratorio, otros robots o las maquinarias con las que interacciona, entonces resulta necesario que el usuario pueda establecer estas restricciones antes de iniciar el movimiento del robot.

Las restricciones de movimiento más habituales son las siguientes:

- *Paredes:* Las paredes del laboratorio en el que se mueve el robot pueden ser representadas como planos con los que el robot no puede colisionar.

- *Cajas:* En ocasiones el espacio de trabajo de un robot puede estar limitado dentro de un área concreta que puede ser representada como una caja.

- *Bloques:* Cuando el laboratorio está compuesto por varios robots y maquinarias, cada uno de ellos se convierte en un obstáculo para el resto. Estos obstáculos se pueden representar como bloques con los que el robot no puede colisionar.

- *Esferas:* El espacio de trabajo de un robot también puede estar limitado dentro de un área cuyas dimensiones pueden representarse como una esfera.

Por otro lado, además de la definición de restricciones (físicas o de movimiento) otra cuestión importante a tener en cuenta a la hora de mover un robot manipulador dentro de un entorno estático o dínámico es la posible colisión de éstos con ellos mismos (autocolisión) o con los otros robots que comparten el mismo espacio de trabajo.

En ocasiones, aunque cada una de las articulaciones del robot se mueva dentro de sus límites establecidos (restricciones físicas) puede ocurrir que la combinación resultante de todas ellas haga que el efector del robot colisione con alguna parte del propio robot, lo que se denomina autocolisión.

Además, en cualquier entorno que esté compuesto por más de un robot, es evidente que cuando éstos están en movimiento pueden colisionar, por lo que resulta imprescindible detectar las posibles colisiones entre ellos.

En primer lugar, a la hora de estudiar dicho problema se parte del concepto *Bounding Volume* (BV). Un BV para un conjunto de objetos se puede definir como un volumen cerrado que contiene completamente la unión de los objetos del conjunto. Los BVs se utilizan para mejorar la eficiencia de las operaciones geométricas mediante el uso de volúmenes simples que contienen objetos más complejos. Normalmente, estos volúmenes más simples tienen formas más sencillas de comprobar la superposición entre ellos [110].

En la Figura 3.11 se pueden observar los BVs más comunes y cuyas características se facilitan a continuación [111]:

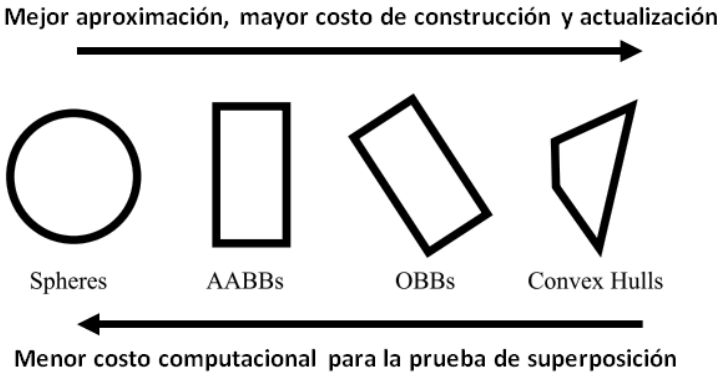



Figura 3.11: Principales tipos de Bounding Volumes

- Una esfera es una superficie de revolución que está definida por su centro $c$ de coordenadas $(c_x, c_y, c_z)$ y su radio $r$. La región determinada por una esfera viene dada por la siguiente expresión:

$$R = \left\{ \ (x, y, z)^T \quad | \quad (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 < r^2 \ \right\} \tag{3.11}$$

- Un Axis-Aligned Bounding Box (AABB) es un paralelípedo rectángular cuyas caras están alineadas con los ejes de coordenadas XYZ o con su sistema de referencia. Un AABB puede ser definido de diferentes formas.

  La primera opción sería especificando el valor mínimo y máximo de la longitud de cada eje, $l_x, l_y, l_z, u_x, u_y, u_z$, cuya región quedaría determinada por la siguiente expresión:

  $$R = \left\{ \ (x, y, z) \quad | \quad l_x \leq x \leq u_x, \ l_y \leq y \leq u_y, \ l_z \leq z \leq u_z \ \right\} \tag{3.12}$$

  Por otro lado, si consideramos las coordenadas de uno de sus vértices $(c_x, c_y, c_z)$ y las longitudes de los ejes $d_x, d_y \ y \ d_z$, entonces la expresión de la región es:

  $$R = \left\{ \ (x, y, z) \quad | \quad c_x \leq x \leq c_x + d_x, \ c_y \leq y \leq c_y + d_y, \ c_z \leq z \leq c_z + d_z \ \right\} \tag{3.13}$$

  Finalmente, también se puede definir considerando el punto central del paralelípedo de coordenadas $(c_x, c_y, c_z)$ y la mitad de las longitudes de cada eje, $r_x, r_y, r_z$, resultando la siguiente ecuación:

  $$R = \left\{ \ (x, y, z) \quad | \quad |c_x - x| \leq r_x, \ |c_y - y| \leq r_y, \ |c_z - z| \leq r_z \ \right\} \tag{3.14}$$

- Un Oriented Bounding Box (OBB) es un paralelípedo rectángular orientado en el espacio y que viene determinado por su centro $c$, la mitad de la longitud de sus ejes, $r_x, r_y$ y $r_z$, y su orientación originada por sus tres vectores ortogonales unitarios $v_x, v_y$ y $v_z$.

  $$R = \left\{ \ c + ar_x v_x + br_y v_y + cr_z v_z \quad | \quad a, b, c \in [-1, 1] \ \right\} \tag{3.15}$$

- El Convex Hulls (envoltura convexa) de un conjunto de puntos X en n dimensiones es la intersección de todos los conjuntos convexos que contienen a X. Es decir, dados k puntos, $x_1, x_2, \ldots, x_k$, la región de un Convex hulls viene dada por [112]:

  $$R = \left\{ \ \sum_{i=1}^{k} \alpha_i x_i \quad | \quad x_i \in X, \ \alpha_i \in \Re, \ \alpha_i \geq 0, \sum_{i=1}^{k} \alpha_i = 1 \ \right\} \tag{3.16}$$

Para poder usar cualquiera de estos BV como parte de un sistema de detección de colisiones, debemos ser capaces de hacer dos cosas: encajar este BV en cualquier colección de objetos y determinar si dos BVs se superponen o no.

Una vez analizados los BVs más comunes, si de entre todos ellos escogemos los OBBs como los BV de referencia, entonces a la hora de determinar si dos OBBs se superponen resulta útil aplicar el algoritmo *Separating Axis Theorem* (SAT) que se enuncia a continuación:

*Dos objetos convexos no colisionan si y sólo si se encuentra un eje separador* $\boldsymbol{n}$ *sobre el que sus proyecciones no intersectan* (Figura 3.12) [113].

Esta teoría aplicada sobre OBBs es bastante eficiente. En el espacio 2D consiste en testear cuatro posibles ejes separadores tal y como se muestra en la Figura 3.13(a) y en el espacio 3D

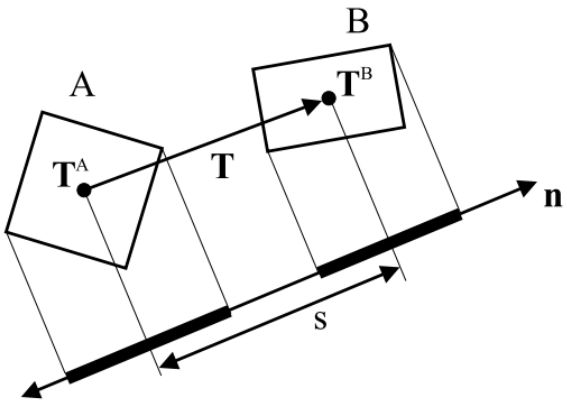
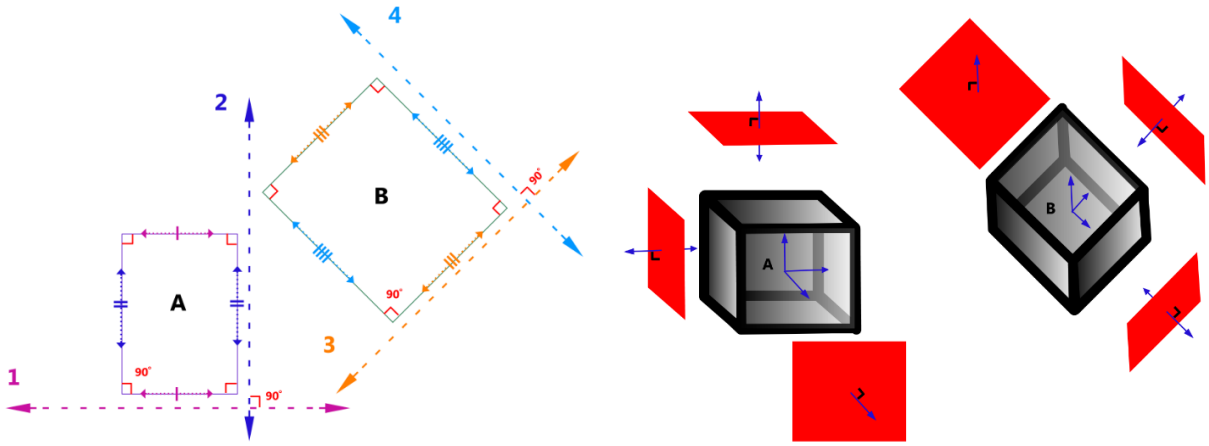



Figura 3.12: Teorema del eje separador

consiste en testear los 15 posibles ejes separadores: 6 originados por las direcciones de las caras de cada uno de los dos OBBs y los 9 productos vectoriales definidos por todos los pares de aristas. En la Figura 3.13(b) se pueden observar los 6 planos de separación (en rojo) entre los dos OBBs, paralelos a sus caras, y los ejes de separación (flechas azules), paralelos a los ejes XYZ de cada OBBs.




(a) Ejes separadores entre dos OBBs en el espacio 2D (b) Planos (en rojo) y ejes (en azul) separadores entre dos OBBs en el espacio 3D

Figura 3.13: Representación de los planos y ejes de separación entre dos OBBs en los espacios 2D y 3D

El único inconveniente que presenta este algoritmo es que no indica el punto ni los ejes donde se produce la intersección.

## 3.2. Otros componentes robóticos

Un laboratorio robótico, en la mayoría de las ocasiones, además de estar compuesto por uno o varios robots (dependiendo de la aplicación) también contiene otra serie de elementos que en esta tesis denominaremos como *componentes robóticos*.

Los componentes robóticos son todos aquellos elementos que complementan las funcionalidades de los robots y ayudan a poder configurar y diseñar el laboratorio concreto. Estos componentes serían los sensores, actuadores, cintas transportadoras, ....

Los sensores permiten la adquisión de información tanto del entorno de trabajo como del estado interno del robot para que realice su tarea eficientemente [97]. En robótica se pueden clasificar en dos grupos: *sensores propioceptivos*, sensores que miden las variables internas del robot (posición, velocidad o ángulo de giro de cada una de las articulaciones), y los *sensores esteroceptivos* o *sensores externos*, sensores que permiten medir variables relativas al entorno del robot, así como reconocer y localizar objetos [114].

Los sensores más utilizados son los siguientes [115]:

- *Sensores de proximidad:* Detectan la presencia de un objeto ya sea por rayos infrarrojos, por sonar, magnéticamente o de otro modo.
- *Sensores de temperatura:* Captan la temperatura del ambiente, de un objeto o de un punto determinado.
- *Sensores magnéticos:* Captan variaciones producidas en campos magnéticos externos. Se utilizan a modo de brújulas para orientación geográfica de los robots.
- *Sensores táctiles:* Sirven para detectar la forma y el tamaño de los objetos que el robot manipula.
- *Sensores de iluminación:* Captan la intensidad luminosa, el color de los objetos, etc. Es muy útil para la identificación de objetos. Es parte de la visión artificial y en numerosas ocasiones son cámaras.
- *Sensores de velocidad, de vibración (acelerómetro) y de inclinación:* Se emplean para determinar la velocidad de actuación de las distintas partes móviles del propio robot o cuando se produce una vibración. También se detecta la inclinación a la que se encuentra el robot o una parte de él.
- *Sensores de fuerza:* Permiten controlar la presión que ejerce la mano del robot al coger un objeto.
- *Sensores de sonido:* Micrófonos que permiten captar sonidos del entorno.

Los actuadores son los sistemas de accionamiento que permiten el movimiento de las articulaciones del robot. Es decir, son dispositivos capaces de transformar energía en la activación de un proceso con la finalidad de generar un efecto sobre un proceso automatizado. El actuador recibe la orden de un regulador o controlador y en función a ella genera la orden para activar un elemento final de control como, por ejemplo, una válvula [116].

Dependiendo del tipo de energía que utilicen los acuadores se clasifican en tres tipos [115]:

- *Actuadores hidráulicos:* Se utilizan para manejar cargas pesadas a una gran velocidad. Sus movimientos pueden ser suaves y rápidos.
- *Actuadores neumáticos:* Son rápidos en sus respuestas, pero no soportan cargas tan pesadas como los hidráulicos.
- *Actuadores eléctricos:* Son los más comunes en los robots móviles. Un ejemplo son los motores eléctricos, que permiten conseguir velocidades y precisión necesarias.

Ejemplos de actuadores son motores, relés y contadores, electro válvulas, pinzas, etc.

Finalmente, existen otros elementos físicos que también forman parte del laboratorio como, por ejemplo, las cintas transportadoras. Estos elementos, como su nombre indica, permiten mover los objetos depositados en la cinta de un punto a otro y de esta manera facilitan a los robots la manipulación de los mismos. Las cintas transportadoras pueden disponer de actuadores y encoder para ponerla en marcha, detenerla y determinar la posición exacta de un objeto que está moviéndose a lo largo de la cinta entre otras cosas (ver Subsección 4.5.3).

## 3.3. Laboratorios robóticos: Modelización de sus elementos

Tal y como se plantea en la actualidad la simulación de procesos, se hace necesario manejar unos modelos que representen dichos procesos y que son computacionalmente complejos, bien sea por la precisión requerida, el número de parámetros y variables a tener en cuenta, o simplemente por la propia estructura del proceso en cuestión [117].

Una vez analizados de forma individual los elementos que forman parte de un laboratorio robótico, es el momento de llevar a cabo una modelización a nivel general. La modelización consiste en identificar un conjunto de valores (variables de entrada, de estado, constantes, y parámetros) que representen el funcionamiento del laboratorio diseñado. Así como establecer un conjunto de relaciones entre dichas variables que permiten conocer cómo responde el modelo a cambios en las variables de entrada. En definitiva, se trata de establecer la relación que existe entre cada uno de los componentes del laboratorio y sus variables de modo que puedan coordinarse y colaborar de forma conjunta para desarrollar las tareas establecidas.

Supongamos que un laboratorio robótico está formado por tres robots manipuladores y dos cintas transportadoras tal y como se muestra en el esquema dado en la Figura 3.14. Cada uno de estos robots tiene establecidas sus propias variables (DOF, posición, velocidad, aceleración,...), sus restricciones (físicas y de movimiento), sus trayectorias, ...De forma similar, el resto de componentes del laboratorio también tienen sus propias variables y parámetros. Por ejemplo, las cintas transportadoras pueden contener un encoder para determinar la posición de los objetos sobre la cinta.

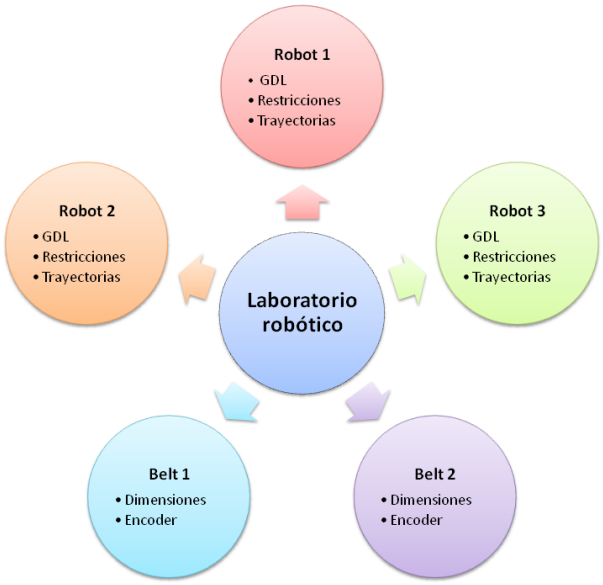



Figura 3.14: Modelización laboratorio robótico

En el momento en el que se plantean las tareas que cada componente debe ejecutar dentro del laboratorio, resulta evidente que deben existir otras variables o parámetros a nivel general que permitan coordinar los diferentes elementos dentro del mismo entorno en el transcurso de las mismas. Por ejemplo, en la Figura 3.15 se facilita un diagrama sencillo en el que se muestra la coordinación entre los distintos elementos para ejecutar algunas tareas. En primer lugar, el programa mueve el Robot1 hasta coger un objeto específico y depositarlo en la cinta transportadora. Entonces, una vez que el elemento se mueve sobre la cinta, el Robot2 se dirige a cogerlo y ejecutaría la siguiente tarea, mientras que el resto de elementos siguen sus tareas establecidas. En este caso, entre otras variables debe existir una que nos indique que el objeto ha sido situado sobre la cinta, de manera que el Robot2 pueda iniciar su movimiento.

En la Sección 4.2 se lleva a cabo la implementación de esta modelización, estableciendo las relaciones que existen entre los distintos elementos de un laboratorio robótico y obteniendo así una perfecta coordinación entre ellos.

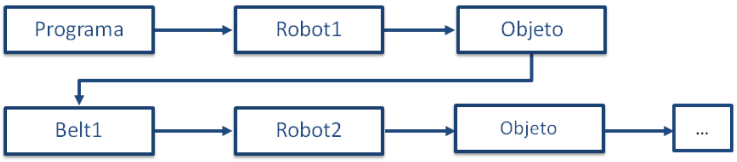



Figura 3.15: Esquema conceptual de la secuencia de tareas de un laboratorio

## 3.4. Aspectos adicionales

Finalmente, para poder concluir con la conceptualización de los elementos de un laboratorio virtual y remoto hay que considerar otros aspectos adicionales que también resultan fundamentales en el diseño de dichos laboratorios. Por un lado, para la creación de un laboratorio virtual resulta imprescindible considerar la visualización de todos los elementos que lo integran para lograr un entorno lo más realista posible. Por otro lado, en el caso de un laboratorio remoto, es necesario tener en cuenta otras dos cuestiones en el momento de establecer las conexiones remotas: el lenguaje de programación propio de cada uno de los robots con los que se quiera trabajar, es decir, el conjunto de comandos que entiende el robot, y los protocolos de comunicación usados para establecer dicha conexión.

- *Visualización 3D:*

  Dentro del proceso de creación de un entorno virtual, una de las tareas que más tiempo consume es el modelado gráfico de los componentes que lo integran. Además del tiempo que conlleva este diseño, también hay que tener en cuenta la calidad, es decir, el nivel de realismo y nivel de interacción del entorno virtual.

  Para el desarrollo de estos elementos se pueden emplear lenguajes de programación de alto nivel (Java, .NET, Python, ...) [118] o herramientas específicas para el desarrollo de entornos virtuales 3D, tanto comerciales (Sketchup [1], SolidWorks [2], ...) como de código abierto (X3D-Edit [3], EJS [4], ...).

  En la Subsección 4.3.5 se facilita el detalle de cómo se pueden modelar estos elementos robóticos para el diseño de entornos virtuales mediante el uso de Java 3D y en la Sección 5.1 se lleva a cabo la misma implementación, pero en este caso haciendo uso del software EJS.

- *Lenguajes propios de programación:*

[1] Goole Sketchup: `http://sketchup.google.com/intl/es/`

[2] SolidWorks: `http://www.solidworks.es/`

[3] X3D-Edit: `http://www.web3d.org/x3d4`

[4] EJS: `http://www.um.es/fem/EjsWiki/`

Programar un robot consiste en indicar paso a paso las diferentes acciones (moverse a un punto, abrir o cerrar la pinza, ...) que éste deberá realizar durante su funcionamiento automático. La flexibilidad en la aplicación del robot y, por tanto, su utilidad dependerá en gran medida de las características de su sistema de programación [102].

No hay un lenguaje de programación normalizado para robots. Cada fabricante tiene el suyo propio, con sus características, sintaxis y estructura de datos propia, sobre el que va haciendo sus nuevas versiones. Aunque cada uno de estos lenguajes de programación orientados a robots tiene una complejidad distinta, todos presentan una serie de características comunes que se detallan a continuación [97]:

- Inicialización: Todos estos lenguajes disponen de mecanismos de inicialización del robot, es decir, posicionar el robot en un punto concreto HOME, así como tareas de calibrado o inicialización de sensores internos.
- Manejo de localizaciones: El lenguaje debe soportar la posibilidad de expresar, de distintas maneras, posiciones y orientaciones. También debe permitir manejar estas posiciones y orientaciones, así como de realizar transformaciones.
- Puntos de paso: Disponen de comandos de movimientos que permiten especificar distintos tipos de trayectorias entre puntos o indicar los movimientos con localización relativa o absoluta. Además, en ocasiones es necesario especificar puntos intermedios de paso para evitar que el robot pueda encontrarse con obstáculos que impidan que éste realice la trayectoria únicamente con puntos finales.
- Características de propósito general: Debe disponer de características que se podrían encuadrar dentro de las de un lenguaje de propósito general. De esta manera, debe ser capaz de manejar tipos de datos convencionales y el flujo de control del programa permitiendo instrucciones condicionales, bucles o acceso a variables.
- Paralelismo: Los lenguajes de programación de robots permiten manejar el paralelismo existente entre las distintas tareas que controlan. Así, por ejemplo, en un robot se debe permitir realizar el control paralelo de cada una de las articulaciones, así como manejar las señales procedentes de los sensores que se generan simultáneamente.
- Sincronización: Un robot interactúa con el mundo real, y, por tanto, debe realizar acciones que dependerán de factores externos o reacionar ante determinadas situaciones de error.
- Variables sensoriales: Durante la programación de un robot va a ser necesario acceder a la información procedente de los sensores.
- Entorno de programación: Finalmente, un elemento imprescindible de un lenguaje es un entorno de programación que permita al programador realizar tareas como edición, ejecución o depuración del programa, con lo que en muchas ocasiones se proporciona el entorno de programación junto con el propio robot.

Se tiende a la estandarización de lenguajes, pero por ahora sigue siendo un problema cuando se quiere trabajar en un mismo laboratorio con diferentes robots, cada uno con un lenguaje propio.

- *Protocolos de comunicación remota:*

  La evolución de las comunicaciones, junto con el desarrollo de herramientas y procesos informáticos como Internet y las tecnologías Web, han revolucionado la forma en que se comparte la información en todos los ámbitos de la investigación y educación.

  Internet se usa como medio de comunicación en el desarrollo de sistemas de interacción remota, aprovechando su fácil accesibilidad, alta disponibilidad, alta flexibilidad y bajo costo. Sin embargo, al implementar aplicaciones sobre Internet se debe tener en cuenta diversas condiciones que pueden afectar su desempeño, tales como [119]:

  - Retraso temporal: causado por el efecto de encolamiento, el tiempo de procesamiento, el tiempo de transmisión en los interruptores y de propagación en las conexiones.
  - Reducido ancho de banda: causado por la congestión en la red y la interconexión de múltiples redes de datos entre el transmisor y receptor.
  - Pérdida de paquetes: se origina por exceder la capacidad de la red, lo que puede ocasionar una pérdida de información parcial o total.
  - Jitter: se define como la variabilidad instantánea del retraso temporal, es decir, el tiempo desde la generación de un paquete hasta que se recibe puede fluctuar de un paquete a otro.

  El éxito de Internet se basa mucho en el empleo del protocolo *TCP/IP*, establecido como protocolo estándar de comunicación en Internet desde el 1 de Enero de 1983. En realidad, el protocolo TCP/IP es un conjunto de dos protocolos de comunicación que permiten el intercambio de información de forma independiente de los sistemas en que ésta se encuentre almacenada. En dicho intercambio cada uno desempeña una actividad diferente:

  - Protocolo *IP* (Internet Protocol): Es el primero que se activa y su labor consiste en dividir en lo que se llama paquetes IP toda la información que hay que remitir.
  - Protocolo *TCP* (Transfer Control Protocol): Se activa una vez creados los paquetes y su función consiste en transmitir los paquetes a su destino.

  Cuando el paquete IP llega a su destino, se activa de nuevo el protocolo IP para reconstruir, a partir de los paquetes IP, la información enviada [120].

  Existen otros protocolos basados en la arquitectura TCP/IP, por ejemplo, *Telnet* (Telecommunication Network), protocolo de Internet estándar que permite conectar terminales y apli-

caciones en Internet. Este protocolo proporciona reglas básicas que permiten vincular a un cliente con un intérprete de comandos (del lado del servidor).

El protocolo Telnet se aplica en una conexión TCP para enviar datos en formato ASCII codificados en 8 bits, entre los cuales se encuentran secuencias de verificación Telnet. Por lo tanto, brinda un sistema de comunicación orientado bidireccional (semidúplex) codificado en 8 bits y fácil de implementar [121]. El acceso a otra computadora se realiza en modo terminal (sin ningún tipo de interfaz gráfica) y permite solucionar fallos a distancia, de tal modo que el usuario puede tratar el problema sin necesidad de estar físicamente junto al equipo en cuestión. Telnet también posibilita la consulta remota de datos o el inicio de una sesión con una máquina UNIX (en este caso, múltiples usuarios pueden abrir sesión de manera simultánea y trabajar con la misma computadora) [122].

El principal problema de Telnet es que no es un protocolo de transferencia de datos seguro, ya que los datos que transmite circulan en la red como texto sin codificar (de manera no cifrada).

En general, todos estos protocolos están basados en la arquitectura *cliente/servidor*, la cual permite la creación de aplicaciones distribuidas. La principal ventaja de esta arquitectura es que facilita la separación de las funciones según su servicio, permitiendo situar cada función en la plataforma más adecuada para su ejecución. Además, también presenta las siguientes ventajas [123]:

- Las redes de ordenadores permiten que múltiples procesadores puedan ejecutar partes distribuidas de una misma aplicación, logrando concurrencia de procesos.
- Existe la posibilidad de migrar aplicaciones de un procesador a otro con modificaciones mínimas en los programas.
- Posibilita el acceso a los datos independientemente de donde se encuentre el usuario.

La arquitectura cliente/servidor nos permite la separación de funciones en tres niveles:

- Lógica de presentación: Se encarga de la entrada y salida de la aplicación con el usuario. Sus principales tareas son: obtener información del usuario, enviar la información del usuario a la lógica de negocio para su procesamiento, recibir los resultados del procesamiento de la lógica de negocio y presentar estos resultados al usuario.
- Lógica de negocio (o aplicación). Se encarga de gestionar los datos a nivel de procesamiento. Actúa de puente entre el usuario y los datos. Sus principales tareas son: recibir la entrada del nivel de presentación, interactuar con la lógica de datos para ejecutar las reglas de negocio (business rules) que tiene que cumplir la aplicación y enviar el resultado del procesamiento al nivel de presentación.

- Lógica de datos. Se encarga de gestionar los datos a nivel de almacenamiento. Sus principales tareas son: almacenar los datos, recuperar los datos, mantener los datos y asegurar la integridad de los datos.

## 3.5. Coordinación de robots manipuladores

La coordinación de dos o más robots consiste en la compatibilización de la ejecución de sus respectivos movimientos, de forma que ejecuten sus tareas sin que ocurra colisión entre ellos. Esto se logra por medio del ajuste de los caminos geométricos (GP, de geometric path), es decir, del ajuste de la secuencia de configuraciones desde una configuración inicial hasta una configuración final, para que nunca se crucen, o de los perfiles de velocidad (VP, de velocity profile) con que los robots se mueven, para que no pasen por el mismo sitio al mismo tiempo [124].

Los métodos existentes para la coordinación de robots se clasifican en acoplados y desacoplados según sea la integración de los procesos de generación de los caminos geométricos y de los perfiles de velocidad con el proceso de coordinación.

- *Métodos acoplados:* Planifican los caminos geométricos y los perfiles de velocidad de todos los robots en una única fase, siendo los procesos de generación y de coordinación de las trayectorias inseparables (ver Figura 3.16).

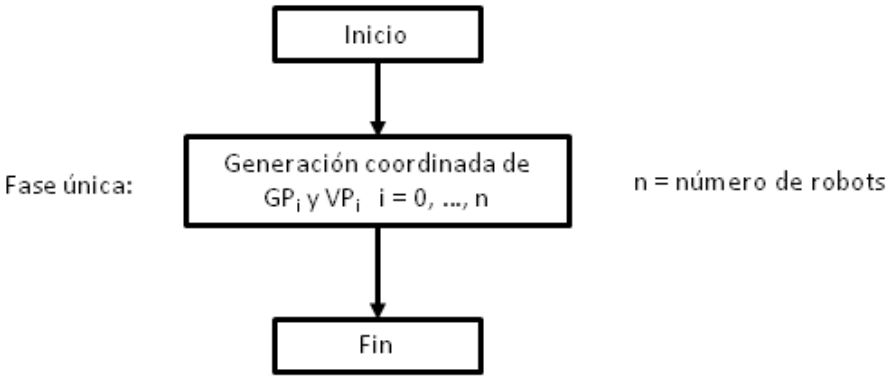
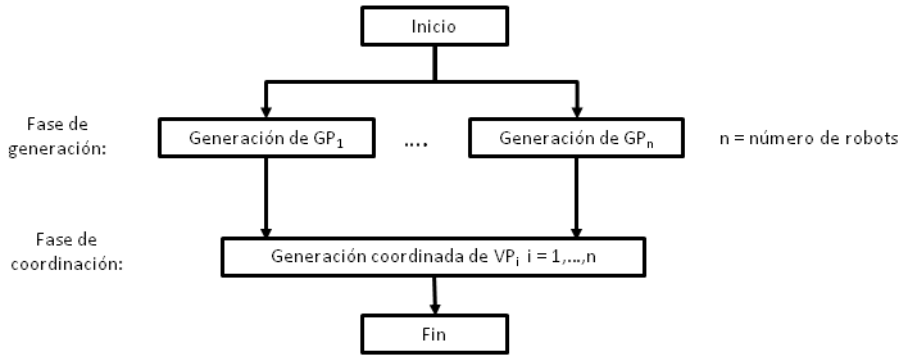


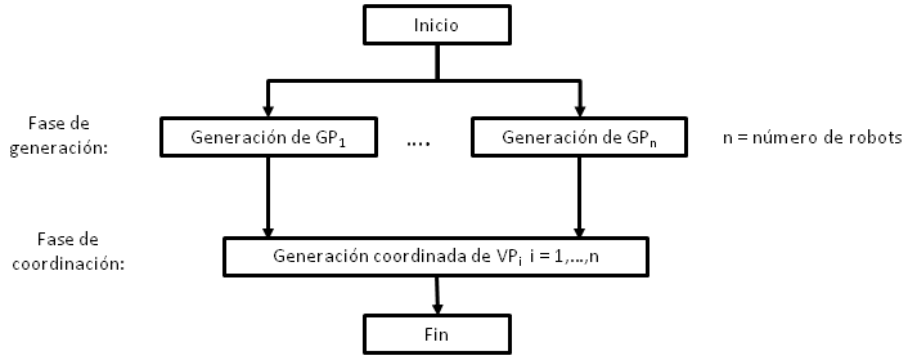

Figura 3.16: Método acoplado de coordinación de trayectorias

- *Métodos desacoplados:* Los métodos desacoplados presentan una fase de coordinación separada de la planificación de los caminos. Estos métodos pueden o no ajustar el camino geométrico (ver Figura 3.17). En el caso de que no se ajuste el camino geométrico, la coordinación se realiza introduciendo retardos puros en la ejecución de los movimientos o modificando los

perfiles de velocidad. El retardo puro consiste en la introducción de tiempos de espera en instantes donde la velocidad del robot que sufre el retardo es nula.




(a) Método desacoplado de coordinación de trayectorias




(b) Método desacoplado de coordinación de caminos geométricos

Figura 3.17: Métodos desacoplados

La coordinación puede ser realizada a priori, denominada *off-line*, o en tiempo de ejecución de los movimientos de los robots, denominada *on-line*.

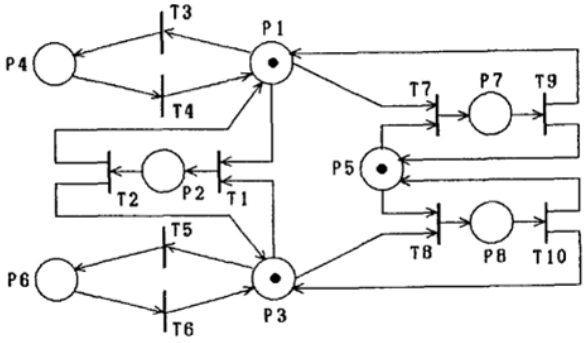
Existen dos casos de coordinación *off-line*. Uno, denominado *off-line fija*, es cuando la coordinación es determinada a priori y no es alterada hasta el final de la ejecución de las tareas coordinadas. El segundo caso, denominado coordinación *off-line variable*, es una generalización del primero, siendo la coordinación determinada a priori, pero pudiendo elegir alternativas a lo largo de la ejecución.

La coordinación de los movimientos se puede llevar a cabo también mediante la existencia de prioridades. Si en algún momento dado alguno de los robots tiene asignada la prioridad más alta en la ejecución de sus movimientos, los demás deberán adaptar los suyos con el fin de evitar colisiones. Este concepto puede ser aplicado recurrentemente, definiendo una relación de orden de prioridades entre todos los robots del sistema.

Finalmente, se puede destacar el paradigma de las redes de Petri que se utiliza como modelo de sincronización explícita entre los procesos, de forma que determinados estados sirvan como punto para aplicar la generación de trayectorias libres de colisión.

Las redes de Petri son un formalismo matemático utilizado para el modelado de sistemas con eventos concurrentes, asíncronos, distribuidos, paralelos y/o estáticos, que es muy utilizado en el ámbito de la automatización y la robótica. Este paradigma es, por tanto, un buen candidato a ser usado como formalismo de coordinación [125, 126].

En la Figura 3.18 se muestra un ejemplo de una red de Petri que representa el comportamiento de dos robots, $R_1$ y $R_2$, que operan independientemente y en coordinación. Además, los dos robots operan y están sincronizados independientemente con una cinta transportadora $C$ [127].




$P_1$: $R_1$ is in standby.
$P_2$: $R_1$ and $R_2$ are operating in coordination.
$P_3$: $R_2$ is in standby.
$P_4$: $R_1$ is operating independently.
$P_5$: C is in standby.
$P_6$: $R_2$ is operating independently.
$P_7$: $R_1$ and C are operating in coordination.
$P_8$: $R_2$ and C are operating in coordination.

$T_1$: the beginning of $R_1$ and $R_2$ coordinated operation.
$T_2$: the end of $R_1$ and $R_2$ coordinated operation.
$T_3$: the beginning of independent operation of $R_1$.
$T_4$: the end of independent operation of $R_1$.
$T_5$: the beginning of independent operation of $R_2$.
$T_6$: the end of independent operation of $R_2$.
$T_7$: the beginning of coordinated operation of $R_1$ and C.
$T_8$: the beginning of coordinated operation of $R_2$ and C.
$T_9$: the end of coordinated operation of $R_1$ and C.
$T_{10}$: the end of coordinated operation of $R_2$ and C.

Figura 3.18: Coordinación de dos robots mediante una red de Petri

Nuestro trabajo facilita la implementación de cualquier formalismo, al posibilitar la creación de laboratorios virtuales y remotos de robótica donde pueden implementarse y probarse diversas estratégias de coordinación con carácter previo a la instalación en una planta real.

# Capítulo 4

# Diseño de una nueva API para laboratorios virtuales y remotos

Una vez analizados con detalle cada uno de los componentes que integran un laboratorio robótico y planteados sus problemas fundamentales, en este capítulo se va a presentar una nueva API, desarrollada en Java, enfocada a la resolución de dichas cuestiones. De esta manera, se pretende que los usuarios puedan crear laboratorios robóticos virtuales y remotos cómodamente sin necesidad de tener que preocuparse por la resolución e implementación de dichas cuestiones.

## 4.1. Definición de la nueva API para la creación de laboratorios virtuales y remotos

Partiendo de la conceptualización realizada en el capítulo anterior sobre qué es un robot manipulador y qué operaciones básicas puede desempeñar (ver Sección 3.1), así como la revisión del resto de componentes que pueden formar parte de un laboratorio robótico (ver Sección 3.2) y la modelización del concepto de este tipo de laboratorios (ver Sección 3.3), se ha decidido establecer un HAL en Java que cubra todos estos aspectos.

El objetivo de introducir un HAL es facilitar a los usuarios los diseños de los programas. Un HAL define una interfaz en términos de constructores del lenguaje de programación utilizada para desarrollar las aplicaciones [128]. La definición de un modelo de programación estructurado y una interfaz general simplifica enormemente la implementación de los componentes [129]. En la literatura el uso del HAL está muy extendido pudiendo encontrar diversos ejemplos de diferentes

aplicaciones. Por ejemplo, en [130] se proporciona un HAL para abordar la complejidad del hardware de robots humanoides y, en [131] se presenta un HAL para el control de sistemas multi-robots. El sistema de control permite manejar diferentes tipos de robots en un equipo, mientras que cada miembro es capaz de compartir e intercambiar cualquier información con otros miembros del equipo.

El HAL presentado en esta tesis incluye una nueva API, desarrollada también en Java por diversos motivos. Por un lado, Java se ha convertido en uno de los lenguajes de programación más populares en los últimos años. Según TIOBE, indicador de popularidad de los lenguajes de programación [132], Java ocupa la primera posición del ranking de popularidad entre los diversos lenguajes de programación existentes, tal y como se puede observar en la Figura 4.1.

| May 2017 | May 2016 | Change | Programming Language | Ratings |
|---|---|---|---|---|
| 1 | 1 | | Java | 14.639% |
| 2 | 2 | | C | 7.002% |
| 3 | 3 | | C++ | 4.751% |
| 4 | 5 | ^ | Python | 3.548% |
| 5 | 4 | v | C# | 3.457% |
| 6 | 10 | ^^ | Visual Basic .NET | 3.391% |
| 7 | 7 | | JavaScript | 3.071% |
| 8 | 12 | ^^ | Assembly language | 2.859% |
| 9 | 6 | v | PHP | 2.693% |
| 10 | 9 | v | Perl | 2.602% |

Figura 4.1: Ranking de los 10 principales lenguajes de programación

Además, Java es un lenguaje orientado a objetos de manera que permite ejecutar una programación más flexible y es fácil de integrar en aplicaciones web. Estas son algunas de las razones por las que este lenguaje es cada vez más común en la creación de simulaciones para educación, investigación, creación de prototipos robóticos...

Por otro lado, el objetivo inicial del desarrollo de esta librería es su integración en la herramienta de simulación EJS, desarrollada completamente en Java (ver Sección 2.6). De ahí que Java fuese la base para el desarrollo de este nuevo framework. Con esta integración en EJS podemos afirmar que el uso de esta nueva librería proporciona tanto a los programadores como a los no-programadores la capacidad de crear sofisticados laboratorios de robótica virtuales y remotos de alto nivel compuestos por uno o varios de estos robots (y/o de otros componentes robóticos) de una forma muy sencilla y reutilizable sin necesidad de tener altos conocimientos ni de Robótica ni de Java.

Esta nueva API es estándar a cualquier robot manipulador y está enfocada en cubrir todas sus operaciones fundamentales: posicionamiento, ejecución de trayectorias, definición de restricciones

de movimiento (tanto físicas del robot como del entorno que lo rodea) y detección de colisiones. Además, permite su visualización en un entorno virtual 3D, para facilitar el diseño de los laboratorios virtuales, y permite establecer la conexión con el robot real, en el caso del diseño de un laboratorio remoto.

En la Figura 4.2 se proporciona un diagrama UML (Unified Modeling Language) con la estructura de esta nueva librería Java para el diseño de laboratorios robóticos tanto virtuales como remotos. En este esquema se encuentran todas las clases que conforman dicho entorno y, lo más importante, se establece la relación que existe entre cada una de ellas, de modo que permite conocer su interacción y coordinación a la hora de diseñar un laboratorio robótico. Cada una de estas clases será analizada en detalle en las siguientes secciones.

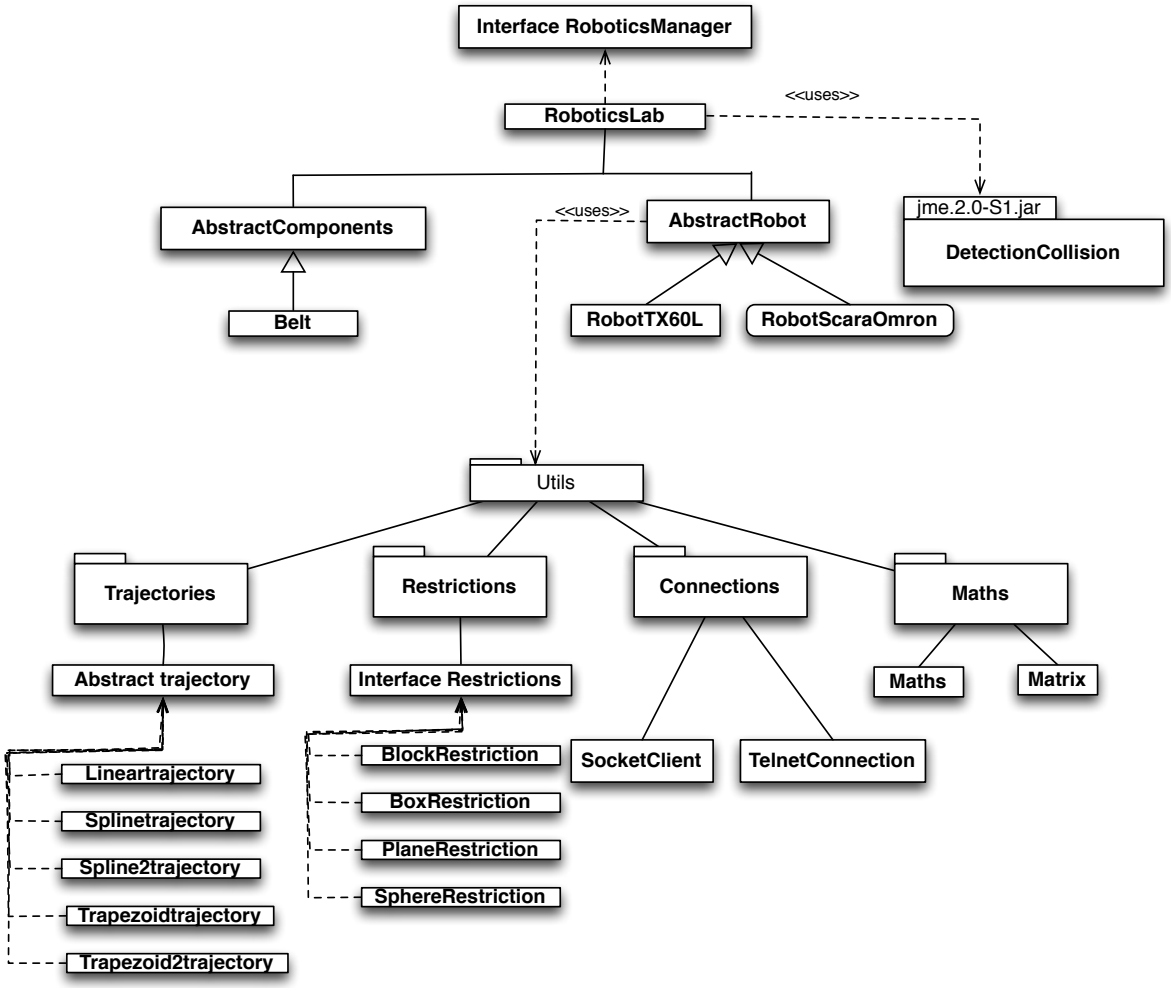



Figura 4.2: Diagrama UML de las clases que componen la nueva API

## 4.2. Diseño y análisis de la clase RoboticsLab

En primer lugar, a la hora de diseñar un laboratorio robótico, la nueva librería facilita al usuario una superclase denominada *RoboticsLab*. Dicha clase permite combinar y relacionar de forma eficaz varios robots o componentes en un mismo laboratorio, y acceder así a todas las propiedades que ofrece este nuevo framework.

El uso de esta nueva clase ofrece múltiples ventajas a la hora de definir un laboratorio robótico, entre las que destacan las siguientes:

- El movimiento de todos los elementos del laboratorio se realiza de una forma coordinada, facilitando el control de la detección de colisiones.
- Permite establecer restricciones generales en el laboratorio que deben satisfacer todos sus elementos independientemente de las restricciones propias de cada uno.
- Facilita la visualización de todos los objetos que componen el laboratorio en un mismo escenario virtual 3D.

En la Tabla 4.1 se describen todos los métodos que componen esta clase y que el usuario puede emplear en el diseño de sus laboratorios robóticos. Estos métodos pueden ser clasificados en tres grupos:

- Métodos para la configuración general del laboratorio: Estos métodos están enfocados en la gestión de las restricciones generales del laboratorio y en la visualización del entorno. Las restricciones que el usuario tiene disponibles serán detalladas en la Subsección 4.3.3.
- Métodos para la configuración de los robots integrados en el laboratorio: Estos métodos permiten al usuario añadir (y/o eliminar) un robot al laboratorio así como gestionar sus movimientos.
- Métodos para la configuración del resto de componentes robóticos: Estos métodos permiten al usuario añadir (y/o eliminar) un componente robótico al laboratorio.

La clase *RoboticsLab* se integra como un elemento más dentro del laboratorio de una manera muy sencilla. Para ello, bastaría con escribir la siguiente línea de código:

Código 4.1: Definición de una clase RoboticsLab

```
RoboticsLab roboticsLab = new RoboticsLab();
```

donde *"roboticsLab"* sería el nombre que se le ha asignado como ejemplo al nuevo laboratorio robótico.

Tabla 4.1: Métodos que componen la clase RoboticsLab

| **Métodos disponibles para la configuración del laboratorio** |
|---|
| **void addRestrictionLab(Restriction rest)**<br>*Añade una restricción al laboratorio* |
| **void removeRestrictionLab(Restriction rest)**<br>*Elimina del laboratorio la restricción indicada* |
| **boolean hasRestrictionsLab()**<br>*Devuelve "True"si el laboratorio tiene algún restricción y "False" en caso contrario* |
| **DrawingPanel3D getRoboticsPanel3D()**<br>*Devuelve el panel 3D donde se representa el laboratorio* |
| **Element encapsuleObject(Element element1, Element element2, char axis)**<br>*Unifica dos elementos en uno único para facilitar su manipulación respecto a uno de los ejes X, Y o Z* |
| **Métodos disponibles para los robots** |
| **void addRobot(AbstractRobot robot, double [] position, boolean addView)**<br>*Añade un robot al laboratorio en la posición dada* |
| **void addRobot(AbstractRobot robot, Group robotGroup, boolean addView)**<br>*Añade un robot al laboratorio en la posición del grupo 3D dado* |
| **boolean hasRobots()**<br>*Devuelve "True"si el laboratorio tiene algún robot y "False" en caso contrario* |
| **int getNumberOfRobots()**<br>*Devuelve el número de robots que se han añadido al laboratorio* |
| **void removeRobot(AbstractRobot robot)**<br>*Elimina del laboratorio el robot indicado* |
| **void trajectoryStep(AbstractRobot robot, double step)**<br>*Mueve el robot al siguiente punto de la trayectoria verificando que satisface sus restricciones, las del laboratorio y que no exite ningún tipo de colisión con ningún otro elemento ni con él mismo* |
| **void moveRobot(AbstractRobot robot, double [] q) throws Exception**<br>*Mueve el robot a la posición indicada en coordenadas articulares verificando que satisface sus restricciones, las del laboratorio y que no exite ningún tipo de colisión con ningún otro elemento ni con él mismo* |
| **void moveRobotC(AbstractRobot robot, double [] point) throws Exception**<br>*Mueve el robot a la posición indicada en coordenadas cartesianas verificando que satisface sus restricciones, las del laboratorio y que no exite ningún tipo de colisión con ningún otro elemento ni con él mismo* |
| **Métodos disponibles para los componentes** |
| **void addComponent(AbstractComponent component, double [] position, boolean addView)**<br>*Añade un componente al laboratorio en la posición dada* |
| **void addComponent(AbstractComponent component,Group componentGroup, boolean addView)**<br>*Añade un componente al laboratorio en la posición del grupo 3D dado* |
| **boolean hasComponents()**<br>*Devuelve "True"si el laboratorio tiene algún componente y "False" en caso contrario* |
| **int getNumberOfComponents()**<br>*Devuelve el número de componentes que se han añadido al laboratorio* |
| **void removeComponent(AbstractComponent component)**<br>*Elimina del laboratorio el componente indicado* |

Independientemente de las restricciones que el usuario pueda establecer dentro del laboratorio, hay que tener en cuenta que cada vez que se incorpora un nuevo elemento al laboratorio la librería automáticamente establece una retricción general con su posición y dimensión. Dicha restricción se utilizará para evitar situar dos elementos en una misma posición y facilitar la detección de colisiones

entre los distintos elementos del entorno.

En el Capítulo 6 se presentarán diferentes ejemplos de laboratorios virtuales y remotos donde se hace uso del elemento *RoboticsLab* y de todas sus propiedades definidas en esta sección.

## 4.3. Diseño y análisis de la clase AbstractRobot

En esta sección se va a analizar el diseño de la clase *AbstractRobot*, en la cual se ha recogido el concepto de robot manipulador y las características fundamentales que hay que considerar a la hora de diseñar un laboratorio robótico con estos tipos de robots. Finalmente, también se detalla el procedimiento que un implementador debería seguir para integrar robots concretos en esta librería, es decir, se especifican qué métodos deben implementarse para cada robot específico.

### 4.3.1. Posicionamiento del robot

Generalmente, todos los robots tienen embebidos en sus controladores la resolución de sus cinemáticas e incluso algunos algoritmos de planificación de trayectorias. Sin embargo, con el fin de facilitar el diseño de los laboratorios virtuales, surge la necesidad de integrar dichos algoritmos en esta librería, en concreto en la clase *AbstractRobot*.

En esta clase se han definido unos métodos sencillos para llevar a cabo el posicionamiento de cada robot, basándonos en cualquiera de los sistemas de coordenadas, articular o cartesiano, resumidos en la Tabla 4.2. Además, contiene otras utilidades como por ejemplo, métodos para abrir o cerrar la herramienta del robot o métodos para obtener información básica del robot como puede ser conocer la posición home del robot en coordenadas articulares o en cartesianas.

Estos métodos son muy sencillos de utilizar en las simulaciones. Por ejemplo, dado un robot de n GDL definido en el laboratorio con el nombre "robotName", para llevarlo a una posición dada "position" bastaría con escribir las siguientes líneas de código:

Código 4.2: Ejemplo de cómo posicionar un robot

```
double[] position = {90, 45, 0.03, 180}; //definimos la posición donde queremos mover el robot
try{
  robotName.moveToQ(position); //movemos el robot a dicha posición
}catch (Exception exc) {exc.printStackTrace();}
```

Independientemente de estos métodos iniciales que nos permiten mover el robot a una posición dada, resulta imprescindible solventar el problema de la cinemática directa e inversa para resolver completamente el problema del posicionamiento de un robot. Dichas cuestiones se tratarán en las siguientes subsecciones.

Tabla 4.2: Métodos disponibles para posicionar un robot y obtener información básica sobre su configuración

| **Posicionamiento del robot** |
|---|
| **void home() throws Exception**<br>*Mueve el robot a su posición inicial por defecto* |
| **void moveToQ(double [] q) throws Exception**<br>*Mueve el robot a la posición articular dada ($q_1$, ..., $q_n$)* |
| **void moveToC(double [] point) throws Exception**<br>*Mueve el robot a la posición dada en coordenadas Cartesianas (x, y, z, $r_x$, $r_y$, $r_z$)* |
| **void openTool() throws Exception**<br>*Abre la herramienta del robot* |
| **void closeTool() throws Exception**<br>*Cierra la herramienta del robot* |
| **Otras utilidades** |
| **int getDOF()**<br>*Devuelve el número de grados de libertad del robot* |
| **int getCartesianArrayLength()**<br>*Devuelve la longitud del vector de coordenadas cartesianas del robot* |
| **double [] getHome()**<br>*Devuelve la posición por defecto de las articulaciones del robot* |
| **double [] getCartesianHome()**<br>*Devuelve la posición por defecto del actuador del robot en coordenadas Cartesianas* |

#### 4.3.1.1. Resolución de la cinemática directa

En la Sección 3.1.1.1 se describieron los principales algoritmos existentes hoy en día para el cálculo de la cinemática directa. De entre todos ellos, en esta librería se ha seleccionado el algoritmo de *"Denavit-Hartenberg"* como la base para dicha resolución.

Para la integración del algoritmo de DH, la librería se apoya en los métodos y parámetros que se facilitan en el diagrama de la Figura 4.3. De entre todos ellos, a la hora de resolver la cinemática directa de un robot particular sólo sería necesario facilitar los parámetros de DH (variable DHParameters), los GDL del robot (variable DOF) e implementar el método calculateDH(double [] q). El resto de clases y métodos ya han sido implementados en la nueva API, independientemente del robot.

En las Subsecciones 4.5.1 y 4.5.2 se facilita la implementación detallada para dos robots concretos.

#### 4.3.1.2. Resolución de la cinemática inversa

A diferencia de la cinemática directa, la resolución de la cinemática inversa es más compleja, ya que depende de la configuración propia del robot y no presenta solución única. En la Subsección 3.1.1.2 se detallaron algunos de los principales métodos de resolución, de manera que la interfaz *AbstractRobot* declara el método:

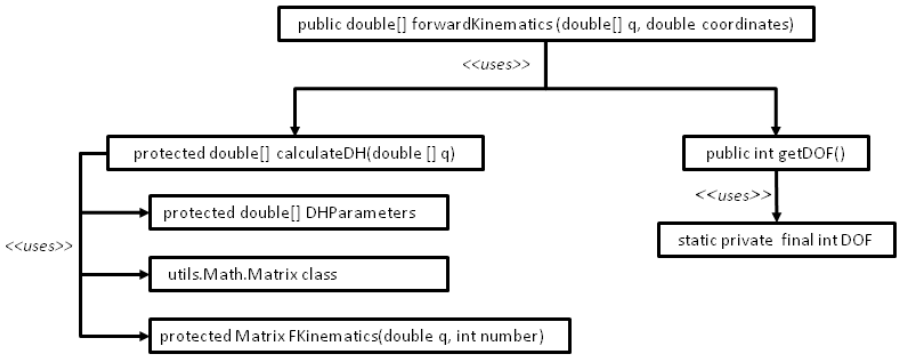



Figura 4.3: Diagrama de clases para la implementación de la cinemática directa

*double[] inverseKinematics(double[] coordinates, double[] q, boolean shoulder, boolean elbow, boolean wrist)*

donde el implementador debe llevar a cabo la implementación del algoritmo seleccionado para el robot concreto.

Los detalles de la implementación de los algoritmos seleccionados en cada uno de los robots integrados en la librería se pueden encontrar en las Subsecciones 4.5.1 y 4.5.2.

### 4.3.2. Diseño de la planificación de trayectorias

En la Subsección 3.1.2 se destacaron tres tipos de interpoladores para la planificación de trayectorias: Lineal, Spline cúbico y Trapezoidal. Dichos interpoladores han sido implementados en la nueva librería de manera que la clase *AbstractRobot* los tiene disponibles para poder establecer diferentes tipos de trayectorias a sus robots (tal y como se podía observar en la Figura 4.2).

La librería dispone de esos tres tipos de interpoladores y otras variedades de los mismos, dependiendo de qué variables se usen para su definición. Todos estos métodos han sido resumidos en la Tabla 4.3.

Cada tipo de trayectoria se puede establecer tanto en coordenadas articulares como en coordenadas cartesianas. Por ejemplo, para la definición de una trayectoria lineal en coordenadas articulares el usuario dispone del método *"Trajectory setLinearTrajectory(double [] qi, double [] qf, double duration)"*, mientras que si prefiere establecerla en coordenadas cartesianas se apoyaría en el método análogo *"Trajectory setCartesianLinearTrajectory(double [] xi, double [] xf, double duration)"*.

Se puede observar que la definición de una trayectoria en coordenadas articulares, $q_i$, es independiente de la cinemática propia del robot. Por lo tanto, la implementación de la planificación de trayectorias puede ser fácilmente usada por cualquier robot de la librería. En el caso de las coordenadas cartesianas, la clase *AbstractRobot* ha resuelto dicha cuestión apoyándose en los métodos de las cinemáticas del robot y en los métodos de las trayectorias dados en coordenadas articulares.

Tabla 4.3: Métodos disponibles para definir trayectorias

| **Constructores para implementar trayectorias** |
|---|
| **Trajectory setLinearTrajectory(double [] qi, double [] qf, double duration)**<br>*Define y devuelve una trayectoria Lineal con las variables dadas en coordenadas articulares* |
| **Trajectory setCartesianLinearTrajectory(double [] xi, double [] xf, double duration)**<br>*Define y devuelve una trayectoria Lineal con las variables dadas en coordenadas cartesianas* |
| **Trajectory setSplineTrajectory(double [] qi, double [] qInt, double [] qf, double tInt, double duration)**<br>*Define y devuelve una trayectoria Spline con un punto intermedio y las variables dadas en coordenadas articulares* |
| **Trajectory setCartesianSplineTrajectory(double [] xi, double [] xInt, double [] xf, double tInt, double duration)**<br>*Define y devuelve una trayectoria Spline con un punto intermedio y las variables dadas en coordenadas cartesianas* |
| **Trajectory setSpline2Trajectory(double [] qi, double [] qInt1, double [] qInt2, double [] qf, double tInt1, double tInt2, double duration)**<br>*Define y devuelve una trayectoria Spline con dos puntos intermedios y las variables dadas en coordenadas articulares* |
| **Trajectory setCartesianSpline2Trajectory(double [] xi, double [] xInt1, double [] xInt2, double [] xf, double tInt1, double tInt2, double duration)**<br>*Define y devuelve una trayectoria Spline con dos puntos intermedios y las variables dadas en coordenadas cartesianas* |
| **Trajectory setTrapezoidTrajectory(double [] qi, double [] qf, double duration)**<br>*Define y devuelve una trayectoria Trapezoidal con las variables dadas en coordenadas articulares* |
| **Trajectory setCartesianTrapezoidTrajectory(double [] xi, double [] xf, double duration)**<br>*Define y devuelve una trayectoria Trapezoidal con las variables dadas en coordenadas cartesianas* |
| **Trajectory setTrapezoid2Trajectory(double [] qi, double [] qInt, double[] qf, double tInt1, double duration)**<br>*Define y devuelve una trayectoria Trapezoidal con un punto intermedio y las variables dadas en coordenadas articulares* |
| **Trajectory setCartesianTrapezoid2Trajectory(double [] xi, double [] xInt, double[] xf, double tInt1, double duration)**<br>*Define y devuelve una trayectoria Trapezoidal con un punto intermedio y las variables dadas en coordenadas cartesianas* |

De este modo, la implementación de cada uno de estos métodos ha sido resuelta y está disponible para todos los robots incluidos en la librería.

Una vez que la trayectoria ha sido definida y asignada al robot (apoyándose en los métodos anteriores), existen otra serie de métodos para mover el robot a lo largo de dicha trayectoria, así como otras utilidades que el usuario pueda necesitar para completar su simulación (ver Tabla 4.4). Por ejemplo, con el método *void trajectoryStep(double dt)* el robot se movería al siguiente punto de su trayectoria y el método *boolean trajectoryFinished(double deltaT)* nos verificaría si el robot ha finalizado su trayectoria.

Es importante destacar que todos los métodos de movimiento del robot (tanto los de posicionamiento como los de planificación de trayectorias) han sido definidos para lanzar una excepción de Java. Esta excepción se producirá si ocurre algún error al intentar ejecutar dicho movimiento, bien sea porque se viola alguna restricción del robot (física o de movimiento) o porque se pierde la comunicación con el robot real en el caso de tener un laboratorio remoto.

Finalmente, en el caso de que el usuario esté diseñando un laboratorio virtual en un entorno OSP (ver más detalles en la Subsección 4.3.5) puede complementar todas estas acciones con la

Tabla 4.4: Métodos disponibles para mover un robot a lo largo de una trayectoria y otras utilidades

| **Métodos para mover el robot a lo largo de una trayectoria** |
|---|
| **void trajectoryHome() throws Exception**<br>*Mueve el robot a la posición inicial de la trayectoria* |
| **void trajectoryEnd() throws Exception**<br>*Mueve el robot a la posición final de la trayectoria* |
| **void trajectoryAtTime(double t) throws Exception**<br>*Mueve el robot al punto de la trayectoria correspondiente al instante de tiempo "t"* |
| **void trajectoryStep(double dt) throws Exception**<br>*Mueve el robot a la posición siguiente de la trayectoria* |
| **Otras utilidades** |
| **double [] trajectoryGetPoint(double t)**<br>*Devuelve el punto de la trayectoria correspondiente al instante de tiempo "t"* |
| **double trajectoryTime()**<br>*Devuelve el tiempo actual de la trayectoria* |
| **boolean trajectoryFinished(double deltaT)**<br>*Devuelve si el robot está en el punto final de la trayectoria o no (después de moverse un "deltaT")* |

visualización de estas trayectorias. Para ello, será necesario que se apoye en los métodos que se facilitan en la Tabla 4.5.

Tabla 4.5: Métodos disponibles para la visualización de trayectorias

| **Métodos para la visualización de las trayectorias** |
|---|
| **void drawPoint(double t, boolean alsoInvalids)**<br>*Dibuja el punto de la trajectoria del instante de tiempo "t"* |
| **void drawPoints(double[] times, boolean alsoInvalids)**<br>*Dibuja los puntos de la trajectoria de los instantes de tiempo "times"* |
| **void drawTrajectory(int nPoints, boolean alsoInvalids)**<br>*Dibuja un número de puntos de la trayectoria* |
| **void trajectorySetVisible(boolean show)**<br>*Hace que la trayectoria sea visible o no* |
| **void trajectoryClear()**<br>*Limpia la trayectoria dibujada* |
| **void trajectorySetColor(Color color)**<br>*Establece un color específico a la trayectoria* |
| **void trajectorySetLineWidth(float width)**<br>*Establece un ancho específico al dibujo de la trayectoria* |

### 4.3.3. Diseño de restricciones

En el momento en el que se posiciona un robot o se mueve a través de una trayectoria previamente definida, sus movimientos están siempre condicionados bien por sus propias restricciones físicas o bien por el entorno en el que se mueve el robot (ver Subsección 3.1.3).

En primer lugar, las restricciones físicas deben implementarse en la clase que extiende la interfaz *AbstractRobot* para implementar un robot concreto, de manera que cualquier robot opera siempre sin exceder sus límites establecidos. Es decir, el movimiento de sus articulaciones, velocidad y aceleración estará dentro de su rango máximo permitido. Por lo tanto, si un usuario intenta exceder alguno de éstos, el método utilizado en ese momento para mover el robot devolverá una excepción de Java y el robot no completará dicho movimiento. Además de estos límites establecidos por defecto según la configuración propia del robot, la clase *AbstractRobot* permite establecer límites adicionales en el caso de la velocidad y la aceleracción de las articulaciones, apoyándose en los métodos dados en la Tabla 4.6. En esta lista, aparecen también otra serie de métodos que nos permiten conocer cada uno de los límites establecidos para la posición, velocidad y aceleración de cada una de las articulaciones.

Tabla 4.6: Métodos disponibles para establecer las restricciones físicas del robot

| **Información sobre las restricciones físicas del robot** |
|---|
| **double getJointMinimum(int joint)**<br>*Devuelve el mínimo valor permitido para la articulación dada* |
| **double getJointMaximum(int joint)**<br>*Devuelve el máximo valor permitido para la articulación dada* |
| **double getJointSpeedMaximum(int joint)**<br>*Devuelve el máximo valor permitido para la velocidad de la articulación dada* |
| **double getJointAccelerationMaximum(int joint)**<br>*Devuelve el máximo valor permitido para la aceleración de la articulación dada* |
| **Especificación de restricciones físicas del robot** |
| **boolean setSpeedLimit(int joint, double maxSpeed)**<br>*Establece un nuevo límite para la velocidad de la articulación dada* |
| **boolean setAccelLimit(int joint, double maxAccel)**<br>*Establece un nuevo límite para la aceleración de la articulación dada* |

Es segundo lugar, para definir una restricción de movimiento, es decir, todas aquellas restricciones del entorno con las que el robot se puede encontrar en el desarrollo de sus tareas, la nueva librería que se está detallando en esta tesis incluye una interfaz de Java denominada *Restriction*. Para establecer una restricción de movimiento basta con crear un objeto que implemente esta interfaz y cuya estructura se facilita en el Código 4.3.

Código 4.3: Interfaz de Java para establecer restricciones de movimiento

```
1public interface Restriction {
2  public boolean allowsPoint(double x, double y, double z);
3  abstract public void action(double x, double y, double z) { } // To be overwritten by user
4}
```

Dicha interfaz está constituida por dos métodos. El método *allowsPoint(double x, double y, double z)* que devolverá el valor *true* si el punto del espacio dado es válido para la operación, es decir, si las coordenadas del extremo del robot satisfacen la restricción definida y el valor *false* en

cualquier otro caso. Como resultado de la violación, el método *action(double x, double y, double z)* lleva a cabo la ejecución de la acción implementada, bien de corrección del movimiento o bien de información (*warning*) de que la restricción ha sido violada.

Para la implementación de esta interfaz, el usuario tiene dos opciones de proceder. Esta interfaz puede ser implementada de forma directa o el usuario puede apoyarse en los cuatro prototipos de restricciones simples que por conveniencia han sido implementadas previamente en la API y cuyas características se describen a continuación:

- **Planar Restriction**: Esta restricción representaría el caso en el que un robot no puede atravesar una pared o un plano virtual, perpendicular a uno de los ejes de coordenadas.

  El constructor de esta restricción sería el siguiente:

  *PlaneRetriction(COORDINATE coord, double limit, boolean greater)*

  donde *coord* sería la coordenada del eje donde está situada nuestra restricción, X, Y o Z, *limit* es el valor donde está situado el plano y *greater* se definirá como *true* si *coord* > *limit* y *false* en caso contrario.

- **Box Restriction**: Esta restricción representaría el caso en el que el robot debe operar dentro de una caja de dimensiones establecidas.

  El constructor vendría dado de la siguiente manera:

  *BoxRetriction(double [] bounds)*

  donde *bounds* son las dimensiones mínimas y máximas de la caja que se quiere establecer como espacio de trabajo $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$

- **Block Restriction**: En el caso en el que el robot no pueda tocar un obstáculo, esta restricción se podría representar como una caja (un bloque) de dimensiones específicas.

  El constructor se definiría de forma análoga al caso anterior, especificando las dimensiones de la caja, pero en este caso en lugar de trabajar dentro de ella no se podria atravesar:

  *BlockRetriction(double [] bounds)*

  donde *bounds* son las dimensiones mínimas y máximas de la caja que se quiere establecer como obstáculo $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$

- **Sphere Restriction**: En este último caso, se considera el caso en el que el robot debe operar dentro de una esfera de dimensiones específicas.

  El constructor en este caso vendría dado por:

*SphereRestriction(double [] center, double radius)*

donde *center* serían las coordenadas del centro de la esfera $(x_0, y_0, z_0)$ y *radius* el radio de la esfera que se quiere establecer.

Si el usuario desea establecer una de estas restricciones en su laboratorio debe seguir el siguiente procedimiento:

I) Utilizar el constructor correspondiente a la restricción seleccionada en la simulación de su laboratorio.

II) Sobreescribir el método *action(double x, double y, double z)* donde el usuario establece la acción que quiere que se lleve a cabo en caso de que la restricción sea violada.

En la Sección 4.6 se encuentra un ejemplo detallado de cómo se define y establece una restricción plana para un robot en una simulación concreta.

Finalmente, en el Código 4.4 se facilita el detalle de la implementación de la interfaz *Restriction* que se ha llevado a cabo para incorporar la restricción *Planar* en la nueva librería (el resto de implementaciones se han llevado a cabo de forma análoga). El objetivo de mostrar dicha implementación es que pueda servir de ejemplo en el caso de que sea necesario ampliar esta lista de restricciones predefinidas con otros nuevos tipos para cubrir las necesidades del usuario.

Código 4.4: Implementación de la interfaz Restriction para la restricción de tipo PlaneRestriction

```
1 public class PlaneRestriction implements Restriction {
2   static public enum COORDINATE {X, Y, Z};
3   private COORDINATE mCoord;
4   private double mLimit;
5   private boolean mGreater;
6
7   public PlaneRestriction(COORDINATE coord, double limit, boolean greater) {
8     mCoord = coord;
9     mLimit = limit;
10    mGreater = greater;
11  }
12
13  public boolean allowsPoint(double x, double y, double z) {
14    switch (mCoord) {
15      case X: {
16        if (mGreater) return (x > mLimit);
17        return (x < mLimit);
18      }
19      case Y:{
20        if (mGreater) return (y > mLimit);
21        return (y < mLimit);
22      }
23      case Z: {
24        if (mGreater) return (z > mLimit);
```

```
25          return (z < mLimit);
26        }
27      }
28      return true;
29    }
30
31    public void action(double x, double y, double z) { } // To be overwritten by user
32  }
```

De nuevo, tanto estas restricciones de movimiento como las restricciones físicas del robot serán siempre chequeadas por cualquier instrucción de movimiento. En caso de no verificarse, es decir, si algún punto del robot la violara, se obtendrá una excepción y se llevará a cabo la acción que el usuario haya definido previamente, en el caso de las restricciones de movimiento, o se detendrá el movimiento del robot, en el caso de las restricciones físicas.

### 4.3.4. Diseño de la detección de colisiones

En esta subsección se va a detallar cómo se ha llevado a cabo la resolución del problema de la detección de colisiones planteado en la Subsección 3.1.3, es decir, cómo se ha conseguido que el robot opere sin colisionar con él mismo (autocolisión) ni con otros robots o elementos que compartan el espacio de trabajo.

Partiendo de los OBBs como los BV de referencia, el primer paso que se llevaría a cabo en el momento de implementar la configuración de un robot dentro de la librería sería subdividir el volumen del robot en diferentes OBBs, un OBB por cada una de sus articulaciones.

Un OBB viene definido por la Ecuación 3.15, de manera que en la librería se implementaría según se muestra en el Código 4.5. En este código se establece la posición y dimensiones del volumen dado (ver Líneas 5-7) y se definiría su orientación con sus vectores de posición (ver Líneas 10-23). Todos los OBBs de un robot y sus vectores son guardados en diferentes listas (ver Líneas 1-3) para poder comprobar posteriormente su posible colisión. La definición de estos OBBs se actualizarán automáticamente (internamente en el programa) cada vez que el robot se mueva a una nueva posición.

Código 4.5: Definición de un OBB para una de las articulaciones de un robot

```
1  protected Hashtable<OrientedBoundingBox,Element> obbTable = new Hashtable<
       OrientedBoundingBox,Element>();
2  protected java.util.ArrayList<OrientedBoundingBox> obbList = new java.util.ArrayList<
       OrientedBoundingBox>();
3  protected Hashtable<OrientedBoundingBox,Element[]> obbTrihedronTable = new Hashtable<
       OrientedBoundingBox,Element[]>();
4  ...
5  ElementBox OBB0 = new ElementBox();
6  OBB0.setXYZ(x0, y0, z0); // where (x0, y0, z0) are the coordinates of the position of the OBB
7  OBB0.setSizeXYZ(a, b, c); // where (a, b, c) are the sizes of each axis
8  robotGroup.addElement(OBB0);
```

```
9  {
10   OrientedBoundingBox obb0 = new OrientedBoundingBox();
11   obb0.setExtent(new Vector3f((float)(OBB0.getSizeX()/2), (float)(OBB0.getSizeY()/2), (
         float)(OBB0.getSizeZ()/2)));
12   obbTable.put(obb0, OBB0);
13   obbList.add(obb0);
14   Element[] trihedron = new Element[3];
15   for (int i=0; i<3; i++) {
16     trihedron[i] = new ElementArrow();
17     trihedron[i].getStyle().setLineColor(Color.RED);
18     trihedron[i].getStyle().setLineWidth(4);
19   }
20   trihedron[0].getStyle().setLineColor(Color.RED);
21   trihedron[1].getStyle().setLineColor(Color.YELLOW);
22   trihedron[2].getStyle().setLineColor(Color.BLUE);
23   obbtTrihedronTable.put(obb0, trihedron);
24 }
25 ...
```

Una vez establecidos todos los OBBs de un robot, los métodos de movimiento chequearán que no existe ninguna colisión entre ellos antes de ejecutar cualquier movimiento. Para estas comprobaciones la librería se apoya en una librería auxiliar *jMonkeyEngine*, resolviendo así el problema de la autocolisión.

La librería *jMonkeyEngine* es un motor de juegos de código abierto y gratuito, hecho especialmente para desarrolladores de juevos que quieren crear juegos 3D usando tecnología moderna. El software está completamente programado en Java y es de fácil uso [133]. Dicho entorno está compuesto por una gran variedad de paquetes y clases con distintas funcionalidades. De entre todos ellos, la Figura 4.4 muestra aquellos que han sido empleados en la resolución de la detección de colisiones entre dos OBBs.

Por otro lado, si el laboratorio estuviese compuesto por más de un robot, entonces la superclase *RoboticsLab* (ver Sección 4.2) sería la encargada de comprobar internamente que cuando un robot se mueve en el laboratorio éste no colisione con ningún otro que forme parte del mismo. Para llevar a cabo dicha comprobación la clase RoboticsLab se apoya en el siguiente método privado:

*private boolean detectionCollision(AbstractRobot robot)*

Básicamente este método se encarga de comprobar que no existe ninguna colisión entre ninguna de las combinaciones posibles de OBBs que pueda existir entre dos robots. Este proceso se realiza de forma análoga a cómo se comprueba la autocolisión, pero en este caso entre dos grupos de OBBs. Por ejemplo, se partiría del $OBB_1$ del primer robot y se comprobaría que no colisiona con ninguno de los OBBs establecidos en el segundo robot. En el siguiente paso, se tomaría el $OBB_2$ y se chequearía con todos los OBBs del segundo robot. Este proceso se repetiría hasta llegar al último $OBB_n$, completando así todas las combinaciones posibles entre estos dos grupos de OBBs.

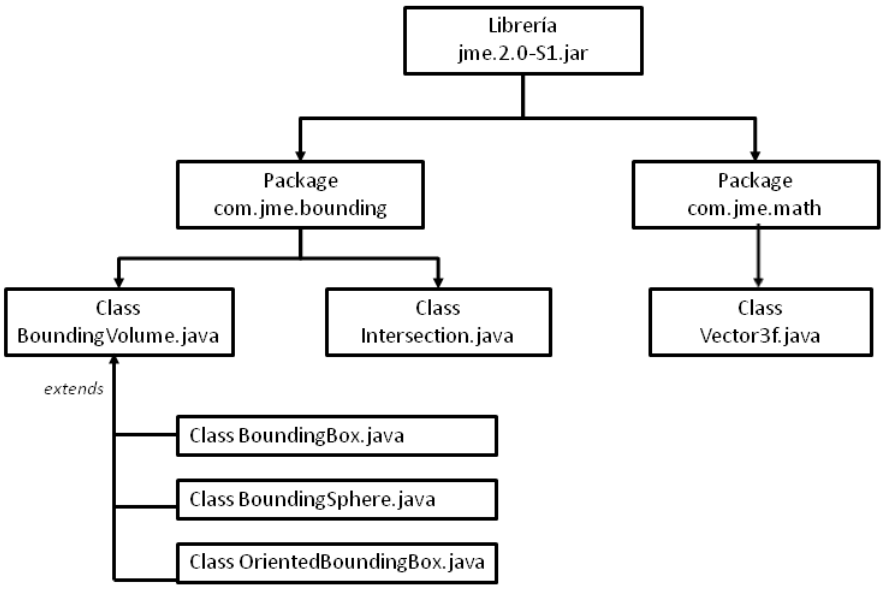



Figura 4.4: Diagrama de clases de la librería auxiliar jMonkey utilizadas por nuestra librería.

Si en alguna de las comprobaciones resultase colisión, entonces el algoritmo se detendría y no sería necesario seguir comprobando el resto de combinaciones.

Es importante resaltar que la detección de colisiones ha sido resuelta internamente en la librería y por lo tanto, no sería necesaria ninguna acción por parte del usuario a la hora de emplearla en el diseño de sus laboratorios robóticos. Automáticamente, cuando un robot se mueva, el programa comprobará internamente que no colisiona con él mismo ni con los otros robots. La única acción necesaria a llevar a cabo sería la definición de los OBBs a la hora de integrar un nuevo robot en la librería.

### 4.3.5. Diseño de la visualización 3D del robot

La librería, y todos sus elementos, han sido diseñados para proporcionar una conveniente visualización 3D basada en el proyecto OSP (detallado en la Sección 2.5). En particular, la librería OSP se apoya en el entorno OSP 3D [86], un poderoso marco para la visualización en 3D de datos científicos y de ingeniería.

La creación de escenarios 3D se puede convertir en una tarea compleja y difícil de entender para usuarios que no tengan altos conocimientos en Java 3D. La librería OSP 3D está enfocada precisamente a este problema, proporcionando una interfaz simple que utiliza Java 3D de manera muy transparente. De modo que los usuarios pueden crear sofisticadas escenas realistas en 3D, con programación en Java puro, sin la necesidad de aprender conceptos o técnicas de Java 3D. Para

crear estos mundos tridimensionales la librería utiliza elementos básicos para representar objetos como cilindros, esferas, cajas, objetos VRML (Virtual Reality Modeling Language) ...

El paquete OSP 3D tiene dos elementos fundamentales: la clase *DrawingPanel3D* que representa un escenario tridimensional y la clase abstracta *Element*, clase base para todos los objetos tridimensionales que pueden aparecer en un escenario 3D. En la Figura 4.5 se encuentra un diagrama con estas clases principales y en [134] se pueden encontrar todas las páginas de referencia de todas las clases que componen dicha librería.

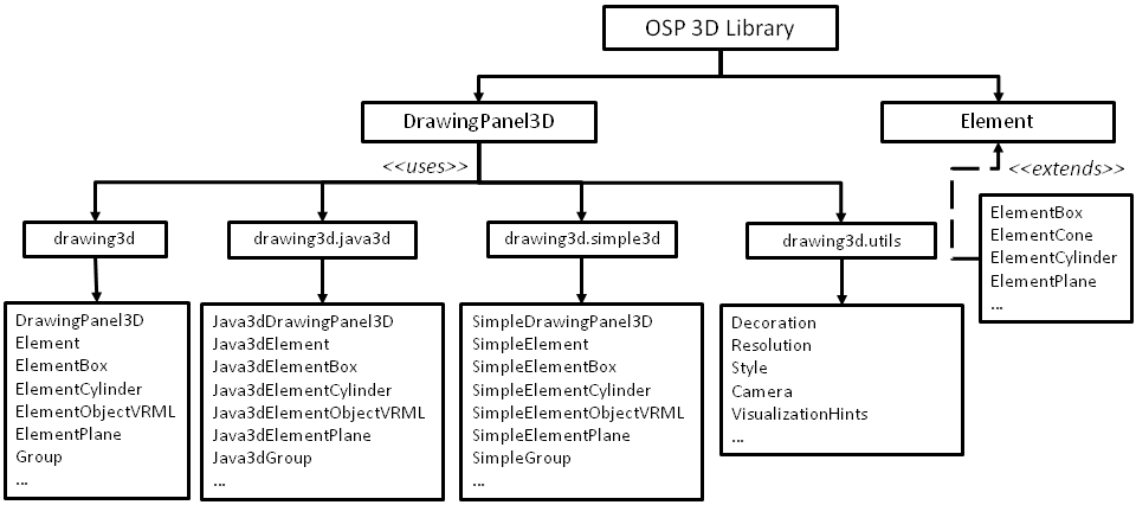



Figura 4.5: Diagrama de clases de la librería OSP 3D

En la Figura 4.6 se muestra un ejemplo de la definición de un elemento *DrawingPanel3D* en el que se ha incluido una caja, es decir, un elemento de tipo *ElementBox*. Este entorno permite al usuario interaccionar, haciendo rotaciones del elemento o zoom sobre el mismo, a diferencia de la implementación estática que se habría obtenido con Java 3D.

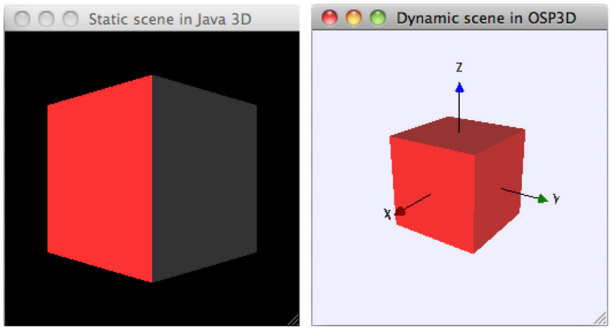



Figura 4.6: Escenario 3D estático con Java 3D (izquierda). Escenario 3D dinámico usando OSP 3D (derecha)

En el caso del diseño de un laboratorio virtual es necesario obtener la visualización de todos los elementos integrados en la nueva librería de Java. Para tener dicho modelo disponible, a la hora de integrar un nuevo elemento en la librería se debe implementar el método *createViewGroup()*. Este método crearía un objeto de tipo *org.opensourcephysics.drawing3d.Group* del paquete OSP 3D - normalmente usando objetos VRML - que posteriormente se podría representar en el elemento *DrawingPanel3D* del laboratorio. Los detalles de implementación de este método se pueden encontrar en la Sección 4.5, ya que obviamente dependen del objeto en particular. Esta implementación se haría sólo cuando se integra un nuevo elemento en la librería, de modo que los usuarios pueden visualizar los objetos previamente integrados sin necesidad de realizar ninguna implementación. Simplemente llamando al método createViewGroup() que nos devolverá un elemento de tipo *Group* con el modelo 3D del objeto sin necesidad de preocuparse por los detalles de la configuración de la vista 3D.

Además, internamente, cada vez que un objeto se mueve a una nueva posición haciendo uso de los métodos correspondientes de la API, los elementos OSP 3D creados por el método *createViewGroup()* se actualizan automáticamente a la nueva posición, acción que nos permitirá visualizar cómo el objeto se está moviendo.

### 4.3.6. Diseño de la conexión remota: lenguajes propios de programación

En la Sección 3.4 se planteó el problema que nos encontramos a la hora de trabajar con diferentes robots, cada uno con su lenguaje propio de programación. Por ello cada vez resulta más importante establecer una estandarización de estos lenguajes de programación de robots.

Una de las soluciones posibles a este problema consiste en establecer un código intermedio que sirva de interfaz común entre lenguajes o procedimientos de programación genéricos (como los que puede usar un simulador o una herramienta general de programación de robots). Este código intermedio debe ser generado de manera automática desde el lenguaje de partida (preproceso) y ser a continuación traducido al lenguaje de programación del robot (postproceso) [102], tal y como se representa en la Figura 4.7.

Siguiendo este esquema, la nueva librería de Java actuaría como código intermedio. De este modo todas las sentencias que el usuario establezca en el diseño de su laboratorio remoto para cada uno de sus robots serán procesadas automáticamente y traducidas a su correspondiente lenguaje de programación, pudiendo así controlar el robot real sin necesidad de que el usuario conozca los detalles de ese lenguaje.

Los detalles de cómo se ha definido este código intermedio para conectar las sentencias de la librería en Java con las sentencias propias del lenguaje del robot se verán en concreto en las implementaciones de cada uno de los elementos de la librería (ver Sección 4.5).

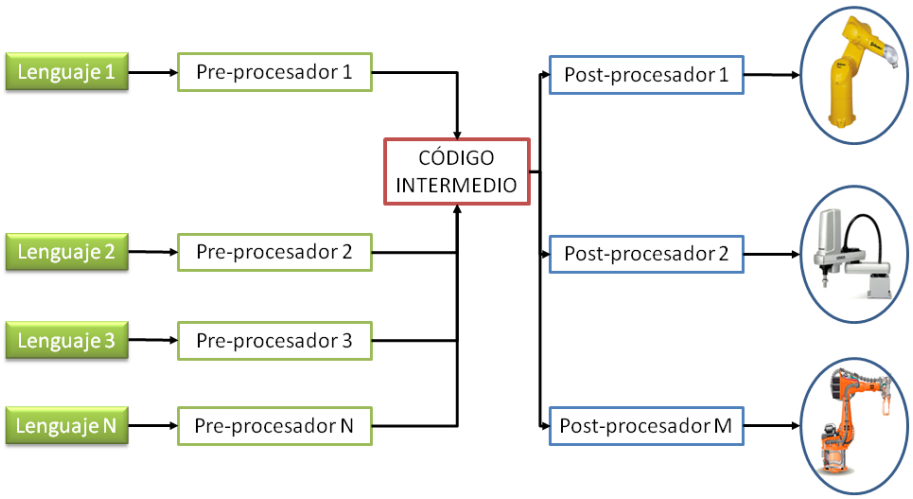



Figura 4.7: Uso de un código intermedio entre lenguajes genéricos y robots

### 4.3.7. Diseño de la conexión remota: protocolos de comunicación

Para el diseño de los laboratorios remotos es necesario establecer la conexión con el robot real. Para ello, resulta necesario establecer el protocolo de comunicación correspondiente para el control del sistema concreto. En esta nueva API se han implementado dos de los protocolos más comunes: TCP/IP y Telnet (detallados en la Sección 3.4).

El protocolo *TCP/IP* está basado en la arquitectura cliente-servidor tal y como se representa en el esquema dado en la Figura 4.8. En el lado del cliente hay un usuario, capaz de controlar el robot desde un lugar remoto. Para poder acceder al controlador del robot y que éste acepte y ejecute los diferentes comandos que el usuario le envía de forma remota, se ha desarrollado una aplicación en la parte del controlador que recibe estos mensajes, los traduce al lenguaje propio del robot y los ejecuta. Esta aplicación está desarrollada de forma que al recibir un parámetro específico por parte de la aplicación cliente, es capaz de asociarlo a la acción correspondiente que el robot debe realizar y ejecutar dicha tarea en el robot real. En las Subsecciones A.1.7.2 y A.1.7.3 se puede encontrar la definición detallada de las clases cliente y servidor implementadas en la librería.

El protocolo Telnet establece directamente un socket para la comunicación que le permite enviar los datos en tiempo real al servidor del robot. Este protocolo permite al usuario acceder a un servidor remoto desde su ordenador local. Puede enviar datos, ejecutar programas o cualquier otra operación como si estuviese en el servidor remoto pasando las órdenes directamente al servidor conectado. En la Subseccion A.2.2.2 se facilitan los detalles de implementación para establecer dicha conexión.

Este conjunto de protocolos integrados en la API puede ser fácilmente ampliado con otros, si se requiere para cubrir las necesidades de nuevos elementos. Es decir, si se implementa un nuevo robot en la librería y éste utiliza un protocolo de comunicación distinto a los previamente definidos,


Figura 4.8: Esquema del protocolo de comunicación TCP/IP

entonces será necesario implementar dicho protocolo.

Una vez que estos protocolos han sido integrados en la librería, la API ofrece unos métodos básicos, dados en la Tabla 4.7, mediante los que el usuario puede establecer de una forma cómoda y sencilla la conexión remota con el robot real, sin necesidad de conocer los detalles de cómo implementar el protocolo de comunicación correspondiente. Como regla general, si la comunicación entre el ordenador y el robot real se pierde mientras se está ejecutando la simulación, los métodos están definidos para lanzar una excepción de Java, *Exception*, y un mensaje de error aparecerá en la consola.

Tabla 4.7: Métodos disponibles para establecer la conexión remota con el robot real

| **Métodos comunes a todos los robots** |
|---|
| **void connect(String hostIP, int portNumber, String user, String password) throws Exception**<br>*Establece la conexión remota mediante la IP, número de puerto, usuario y contraseña dadas* |
| **void disconnect() throws Exception**<br>*Cierra la conexión remota previamente establecida* |

### 4.3.8. Integración de robots manipuladores

A lo largo de esta sección se han detallado todas las características y métodos que la clase *AbstractRobot* ofrece al usuario. En general, esta clase ha sintetizado el concepto de un robot manipulador y ha tratado sus cuestiones fundamentales, ofreciendo una serie de métodos y propiedades comunes a todos los robots, independientemente de sus características propias.

Por lo tanto, a la hora de incluir un robot concreto en esta librería, es necesario que dicho

elemento extiende la clase *AbstractRobot*. Es decir, basta con que se definan los métodos abstractos resumidos en la Figura 4.9, los cuales reúnen las características propias del robot, por ejemplo, número de GDL, posición home, parámetros de DH, cinemática inversa, valores máximos y mínimos de las articulaciones, visualización 3D del modelo del robot... El resto de propiedades y funcionalidades comunes que ofrece la clase *AbstractRobot*, detalladas a lo largo de la sección, serán heredadas automáticamente.

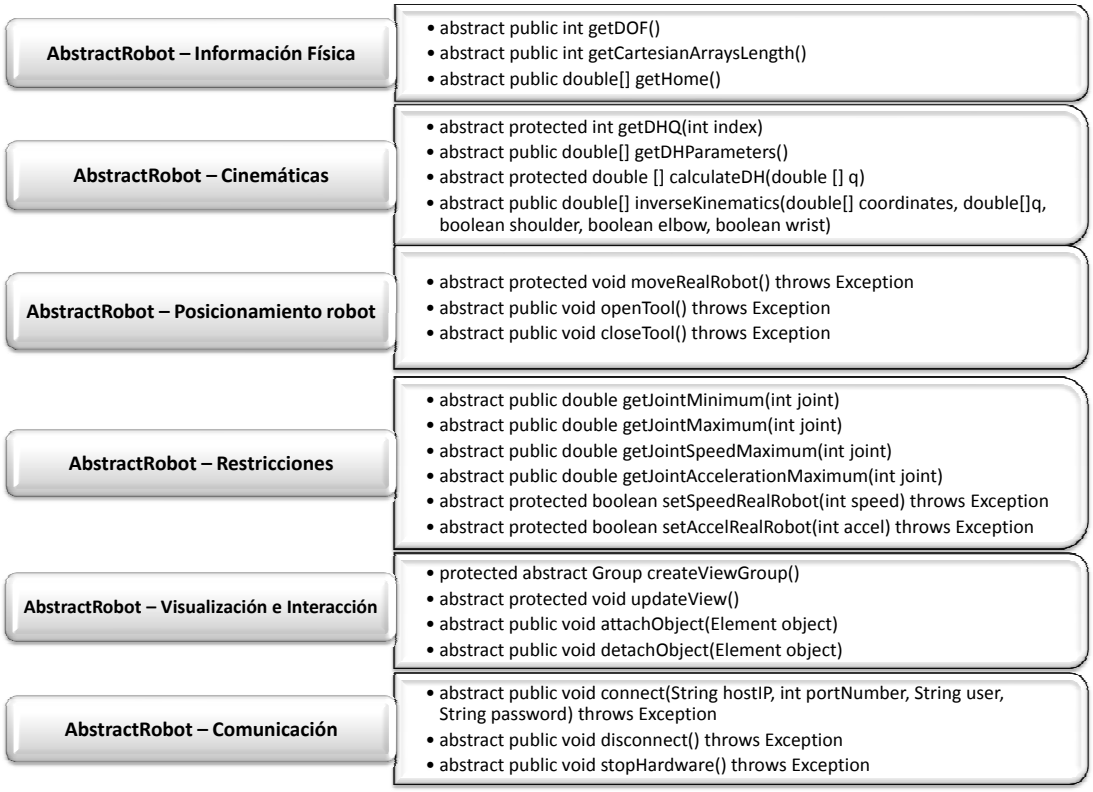



Figura 4.9: Métodos de la clase AbstractRobot a implementar por el usuario en la subclase.

En las Subsecciones 4.5.1 y 4.5.2 se lleva a cabo la implementación detallada de estos métodos de la clase *AbstractRobot* para dos robots manipuladores concretos.

## 4.4. Diseño y análisis de la clase AbstractComponent

De forma análoga al procedimiento que se ha llevado a cabo con la conceptualización de los robots manipuladores, se ha definido una clase abstracta *AbstractComponent* para sintetizar el

concepto de componentes robóticos, es decir, cualquier otro elemento que junto con los robots forman el laboratorio robótico: cintas transportadoras, sensores, actuadores,... (ver Sección 3.2).

Debido a la naturaleza tan distinta de estos componentes robóticos, resulta difícil establecer propiedades comunes a todos ellos. Por ejemplo, la funcionalidad de una cinta transportadora es distinta a la de un sensor de proximidad. Sin embargo, la clase *AbstractComponent* recoge una serie de métodos comunes a todos estos elementos y que se deben implementar en el caso de querer incorporar nuevos componentes en la librería. Estos métodos, dados en la Tabla 4.8, están enfocados principalmente a cubrir dos aspectos. Por un lado, implementar la configuración física del elemento, es decir, sus dimensiones y la posición que ocupa dentro del laboratorio. Por otro lado, la configuración de su visualización en un escenario tridimensional, básicos en el diseño de laboratorios virtuales.

Tabla 4.8: Métodos de la clase AbstractComponent()

| **Métodos de la clase AbstractComponent - Información Física** |
|---|
| **abstract public double getLength()**<br>*Devuelve la longitud del componente robótico* |
| **abstract public double getWidth()**<br>*Devuelve el ancho del componente robótico* |
| **abstract public double getHigh()**<br>*Devuelve la altura del componente robótico* |
| **public double[] getInitialPosition()**<br>*Devuelve la posición inicial que ocupa el objeto dentro del laboratorio* |
| **Métodos de la clase AbstractComponent - Visualización3D** |
| **abstract protected Group createViewGroup()**<br>*Establece el modelo 3D del elemento para su posterior visualización* |
| **abstract protected void updateView()**<br>*Actualiza la vista del elemento si éste ha sufrido algún cambio* |
| **public void addToViewGroup(Group group)**<br>*Añade el modelo 3D del objeto en el entorno tridimensional para su visualización* |
| **public void removeFromViewGroup()**<br>*Elimina la vista del objeto del entorno de visualización* |

A la hora de incorporar un componente robótico a la librería, es necesario que este elemento implemente la clase *AbstractComponent*, es decir, debe implementar todos los métodos abstractos que la integran. Independientemente de estos métodos básicos, el elemento concreto podría añadir en su clase tantos métodos y parámetros como funcionalidades se puedan modelar. En la Subsección 4.5.3 se muestra la implementación de esta clase en el caso de una cinta transportadora, así como todas las características propias que presenta este elemento.

## 4.5. Implementación de componentes concretos

Una vez analizadas las características fundamentales de esta nueva librería en Java para el diseño de laboratorios virtuales y remotos y las principales clases que la componen (ver secciones anteriores), hay que destacar que esta librería ha sido implementada en detalle para dos robots manipuladores de diferentes GDL y una cinta transportadora (ver Figura 4.10).

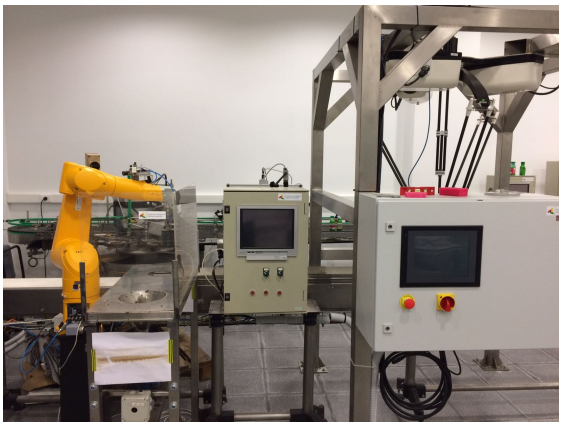
(a) Robot TX60L (b) Robot Scara (c) Cinta Transportadora

Figura 4.10: Elementos integrados en la nueva librería

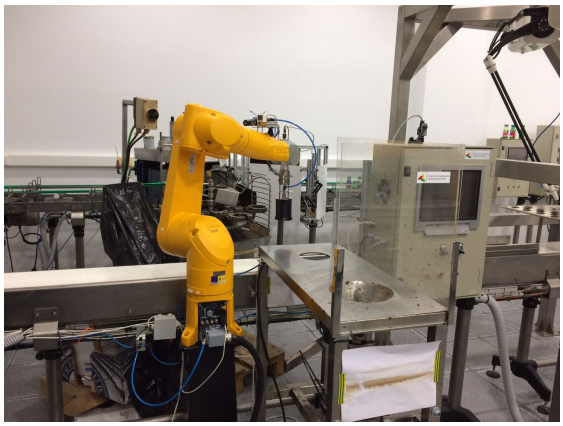
Los motivos por los que se han escogido estos dos robots manipuladores en concreto, el TX60L de Stäubli y el Scara de Omron, son los siguientes: ambos cubren perfectamente todas las necesidades de esta nueva librería y ambos están accesibles físicamente en el laboratorio de nuestro grupo de investigación de Ingeniería aplicada de la Universidad de Murcia (UM), lo cual nos permite trabajar directamente con los robots reales y permite diseñar ejemplos de laboratorios remotos fácilmente.

En la Figura 4.11 se muestran diferentes fotográfias de todos los componentes del laboratorio de este grupo de investigación. En las Figuras 4.11(a) y 4.11(b) se puede observar el robot TX60L, integrado en la librería, un robot paralelo Delta, dos tipos de cintas transportadoras (una de ellas incluida también en la librería), una mesa de corte y exprimido, cuadros de control, etc. En la Figura 4.11(c) se muestra el robot Scara, integrado en la librería, y un robot móvil.

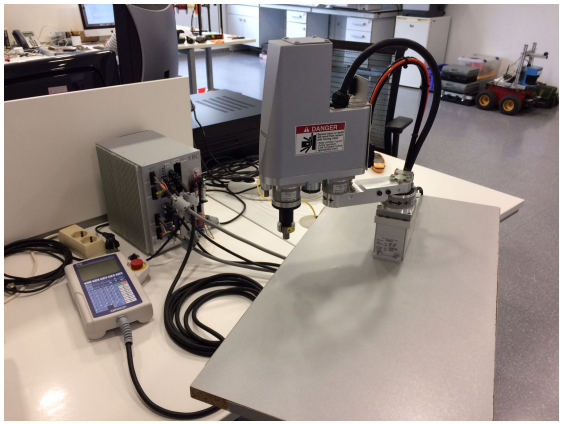
Además, es importante resaltar que, independientemente de los elementos que han sido integrados, la librería puede ser extendida a otros robots o componentes de una forma sencilla. De forma análoga a cómo se han implementado estos elementos y cuyos procedimientos se van a detallar en las siguientes subsecciones, bastará con extender la correspondiente clase abstracta, es decir, la clase *AbstractRobot* para el caso de un nuevo robot o la clase *AbstractComponent* en el caso de un nuevo componente robótico.



(a) Robot TX60L, robot paralelo Delta y otros componentes robóticos I



(b) Robot TX60L, robot paralelo Delta y otros componentes robóticos II



(c) Robot Scara y robot móvil

Figura 4.11: Robots y componentes del laboratorio del grupo de Ingeniería Aplicada de la UM.

### 4.5.1. Robot TX60L

El *robot TX60L* de Stäubli, dado en la Figura 4.10(a), es un robot industrial con seis GDL que ofrece una máxima flexibilidad. Con sus tres traslaciones según el respectivo eje X, Y o Z y sus tres giros o rotaciones (yaw, pitch, roll) relacionadas con estos mismos ejes, se puede posicionar cualquier elemento, objeto u herramienta en el espacio (ver Figura 4.12). Generalmente, el posicionado se consigue por medio de sus tres primeras articulaciones a partir de la base y la orientación con el resto de articulaciones.

Por otro lado, su exclusivo cierre de trabajo esférico permite maximizar el uso del espacio de trabajo de la célula. La estructura reforzada y completamente cerrada del brazo del robot hace que sea ideal para aplicaciones en entornos agresivos, mientras que una muñeca especialmente diseñada es resistente a virutas de cortes y es ideal para entornos de herramientas de mecanizado [9].

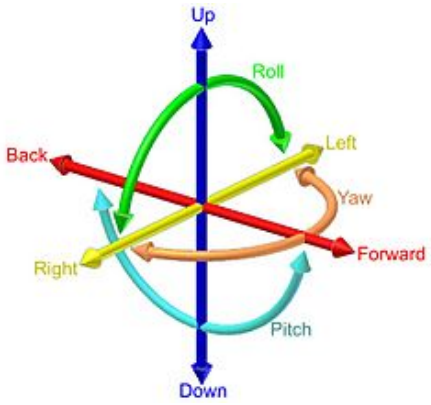



Figura 4.12: Los 6 GDL de un robot: adelante/atrás (forward/back), arriba/abajo (up/down), izquierda/derecha (left/-right), cabecear (pitch), guiñar (yaw), rodar (roll)

Este robot se ajusta perfectamente a todas las características implementadas en la nueva librería para robots, de manera que para integrarlo como elemento y que los usuarios lo tengan accesible para diseñar sus laboratorios robóticos virtuales y remotos basta con extender la clase *AbstractRobot*, cuyo procedimiento se describió en la Subsección 4.3.8.

Inicialmente, este procedimiento comenzaba con la definición de los métodos correspondientes a la información física del robot y posicionamiento del mismo, cuyos detalles se pueden encontrar en las Subsecciones A.1.1 y A.1.2, respectivamente.

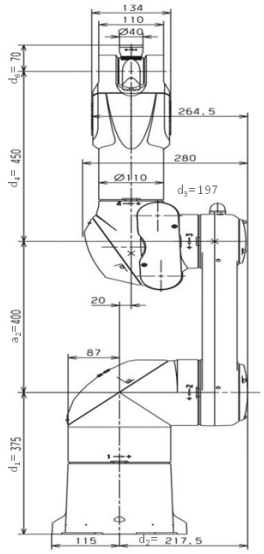
Para la implementación de la cinemática directa (ver Subsección A.1.3), es necesario proporcionar los parámetros de DH ($\theta_i$, $d_i$, $a_i$, $\alpha_i$) de cada una de sus articulaciones con i = (0,...,6), siendo 6 el número de GDL del robot. Estos parámetros están recogidos en la matriz dada en la expresión 4.1, los cuales han sido calculados apoyándose en las dimensiones del robot dadas en la Figura 4.13.

La implementación de la cinemática inversa para este robot se ha llevado a cabo mediante el algoritmo dado en [106] y que se describió en la Subsección 3.1.1.2. Los detalles de esta implementación se pueden encontrar en la Subsección A.1.4.

En la Tabla 4.9 se facilitan los valores entre los que están comprendidos los movimientos de las articulaciones, así como las velocidades y aceleraciones máximas permitidas que servirán de base para la implementación de las correspondientes restricciones físicas del robot (ver Subsección A.1.5).

En la Subsección A.1.6 se encuentra el detalle de los métodos correspondientes a la visualización del modelo 3D del robot y la interacción con el mismo. En la Figura 4.14 se puede observar el nivel de realismo que se obtiene con dicha implementación. Para ello, basta con comparar el modelo 3D del robot TX60L real (ver Figura 4.14(a)) con su modelo 3D virtual (ver Figura 4.14(b)).

Finalmente, para concluir la implementación de este elemento, hay que considerar que el robot



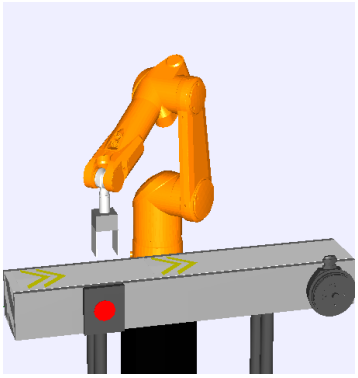

$$DH = \begin{bmatrix} 0 & -\Pi/2 & \Pi/2 & 0 & 0 & 0 \\ d_1 & d_2 & -d_3 & d_4 & 0 & d_6 + lengthTool \\ 0 & a_2 & 0 & 0 & 0 & 0 \\ -\Pi/2 & 0 & \Pi/2 & -\Pi/2 & \Pi/2 & 0 \end{bmatrix} \quad (4.1)$$

Figura 4.13: Dimensiones del robot TX60L

Tabla 4.9: Especificaciones de las restricciones físicas del robot TX60L

| **Rango de movimiento** | **Velocidades máximas** | **Aceleraciones angulares máximas** |
|---|---|---|
| Eje 1 = ± 180º | Eje 1 = 435º/s | $q_1 = 1373º/s^2$ |
| Eje 2 = ± 127,5º | Eje 2 = 385º/s | $q_2 = 1373º/s^2$ |
| Eje 3 = ± 152º | Eje 3 = 500º/s | $q_3 = 3096º/s^2$ |
| Eje 4 = ± 270º | Eje 4 = 995º/s | $q_4 = 2802º/s^2$ |
| Eje 5 = +133, 5º/ − 122, 5º | Eje 5 = 1065º/s | $q_5 = 1707º/s^2$ |
| Eje 6 = ± 270º | Eje 6 = 1445º/s | $q_6 = 8167º/s^2$ |

(a) Robot TX60L real



(b) Robot TX60L virtual

Figura 4.14: Modelos 3D del robot TX60L real y su virtual.

TX60L permite su control de forma remota haciendo uso de la arquitectura TCP/IP (detallada en la Subsección 4.3.7). La implementación de las correspondientes clases cliente y servidor así como la descripción de su lenguaje propio de programación se facilitan en la Subsección A.1.7.

### 4.5.2. Robot Scara

El *robot Scara* (Selective Compliant Assembly Robot Arm) de Omron (ver Figura 4.10(b)) es un brazo articulado con cuatro GDL. Es un robot idóneo para el movimiento de objetos poco pesados en ciclos de gran velocidad [10]. En general, presenta una gran repetitividad, rápidez y gran carga de trabajo. Los dos primeros GDL permiten posicionar el objeto (coordenadas cartesianas $x$ e $y$), el tercero orientarlo, y el cuarto controlar la altura del objeto (coordenada cartesiana $z$) [98].

Al igual que el robot TX60L, este robot encaja perfectamente en esta nueva librería para el diseño de laboratorios robóticos. Su integración se ha llevado a cabo de manera análoga al robot TX60L y sus detalles se encuentran en la Sección A.2.

En la Figura 4.15 se muestran las dimensiones del robot Scara de Omron, necesarias para el cálculo de los parámetros de DH dados en la Expresión 4.2 que serán la base para la definición de la cinemática directa del robot Scara.

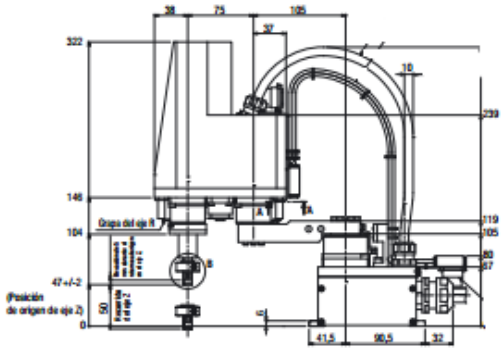


Figura 4.15: Dimensiones del robot Scara

$$DH = \begin{bmatrix} q_1 & q_2 & 0 & q_4 \\ 0,105 & 0 & q_3 & -0,105 \\ 0,105 & 0,075 & 0 & 0 \\ 0 & 0 & 0 & \Pi \end{bmatrix} \quad (4.2)$$

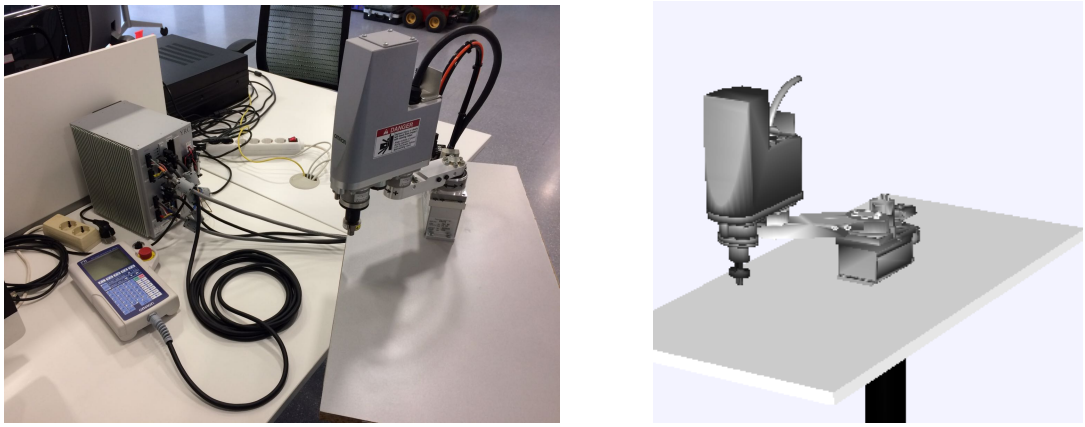
En este caso, para la resolución de la cinemática inversa se ha empleado el algoritmo basado en grupos de Assur dado en [107] y detallado en la Subsección 3.1.1.2. En la Subsección A.2.1 se pueden encontrar las ecuaciones de este algoritmo así como los detalles de dicha implementación.

Por otro lado, en la Tabla 4.10 se resumen los valores extremos para el posicionamiento, velocidad y aceleración de cada una de las cuatro articulaciones de este robot y que se utilizarán en la definición de las restricciones físicas.

En la Figura 4.16 se puede observar el nivel de realismo que se obtiene con la implementación del modelo 3D del elemento. Además, para dotar aún de más realismo dicha implementación, se ha modelado exactamente la misma base sobre la que está situado el robot real.

Tabla 4.10: Especificaciones de las restricciones físicas del robot Scara

| **Rango de movimiento** | **Velocidades máximas** | **Aceleraciones angulares máximas** |
|---|---|---|
| Eje X = ± 125º | Eje X = 3,3 m/s | $q_1 = 19m/s^2$ |
| Eje Y = ± 145º | Eje Y = 3,3 m/s | $q_2 = 19m/s^2$ |
| Eje Z = $-0{,}003m/+0{,}05m$ | Eje Z = 0,9 m/s | $q_3 = 16m/s^2$ |
| Eje R = ± 360º | Eje R = 1700º/s | $q_4 = 767rad/s^2$ |



(a) Robot Scara real (b) Robot Scara virtual

Figura 4.16: Modelos 3D del robot Scara real y su virtual.

Por último, cabe destacar que en este caso el sistema de control remoto empleado está basado en el protocolo de comunicación Telnet (ver Subsección 4.3.7). Los detalles de dicha implementación, los de la definición de la clase cliente y las principales características del lenguaje propio de programación del Scara se pueden encontrar en la Subsección A.2.2.

### 4.5.3. Componente Belt

La *cinta transportadora* (Belt) (ver Figura 4.10(c)), como su nombre indica, es el elemento que se usa para mover objetos de un punto a otro. Los objetos se mueven, a través de un plano horizontal y con movimiento continuo.

Para la integración de este elemento en la nueva librería se ha creado una subclase de la clase *AbstractComponent*, como se detalla en la Sección 4.4. Además de los métodos básicos que ofrece dicha clase, en el caso de una cinta transportadora se han añadido otros parámetros y métodos que permiten al usuario controlar el movimiento de la cinta.

Este elemento ofrece al usuario la posibilidad de añadir un encoder para el control de su movimiento. Un encoder, también conocido como codificador o decodificador, es un dispositivo cuya

función es la de convertir el movimiento mecánico (giros del eje) en pulsos digitales o analógicos que pueden ser interpretados por un controlador de movimiento [135].

Por lo tanto, el encoder permite calcular la posición exacta de un objeto sobre la cinta en marcha. Conocidas las frecuencias del motor y del variador, el número de revoluciones del encoder, su factor de reducción, el número de pulsos y diámetro del rodillo de la cinta, entonces el cálculo de la distancia lineal de la cinta vendrá dado por las siguientes expresiones:

$$\begin{cases} revMotor = \frac{variatorFrequency \cdot revolutions}{motorFrequency} \\ RPM = \frac{revMotor}{reductionFactor} \\ linealDistance = \frac{RPM \cdot (diametroRoller \cdot \Pi)}{60}\ (m/seg) \\ pulses = \frac{distanceBelt \cdot encoderPulses}{diametroRoller \cdot \Pi} \end{cases} \tag{4.3}$$

En la Tabla 4.11 se recogen todos los métodos que el usuario puede emplear a la hora de usar este elemento en sus simulaciones. Los detalles de las implementaciones de todos estos métodos se pueden encontrar en el Apéndice A.3.

En la Subsección A.3.1 se encuentra la implementación completa de los métodos necesarios para obtener la información física del elemento, en la Subsección A.3.2 la implementación de la visualización de este elemento en un entorno 3D y en la Subsección A.3.3 la definición del resto de métodos para el control del movimiento de la cinta.

En la Figura 4.14 también se puede comparar el resultado del modelo 3D de la cinta transportadora con el modelo de la cinta real del laboratorio.

## 4.6. Ejemplo básico de una simulación robótica diseñada con la librería Java

Finalmente, para concluir este capítulo, se va a presentar un primer ejemplo básico de la programación en Java de un laboratorio robótico virtual y remoto compuesto por un robot Scara virtual y su robot real correspondiente.

Básicamente, este laboratorio consiste en lo siguiente: se incorpora un elemento robot Scara a la simulación y se establece la conexión con su robot real. Tras establecerse la conexión, se llevan ambos robots a su posición *home*, se define una restricción de tipo *Plane* y se mueven ambos robots a través de una trayectoria lineal. El Código 4.6 muestra cómo se llevaría a cabo el diseño de dicho laboratorio:

Código 4.6: Ejemplo básico de una simulación de un robot Scara en Java

```
1 public class ExampleScara {
2   public static void main(String[] args) {
3
```

Tabla 4.11: Métodos definidos para el componente robótico “Belt”

| **Constructores del elemento Belt** |
|---|
| **Belt(boolean encoder)**<br>*Constructor general del elemento al que se le puede añadir un encoder dependiendo de si la variable encoder es “true” o “false”* |
| **Belt(double length, double width, double high, boolean encoder)**<br>*Constructor del elemento en el que se pueden especificar las dimensiones de la cinta transportadora (largo, ancho y alto) además de la posibilidad de añadir un encoder dependiendo de si la variable encoder es “true” o “false”* |
| **Métodos sobre la información física del elemento Belt** |
| **double getLength()**<br>*Devuelve la longitud de la cinta transportadora* |
| **double getWidth()**<br>*Devuelve el ancho de la cinta transportadora* |
| **double getHigh()**<br>*Devuelve la altura de la cinta transportadora* |
| **Métodos para configuración de la visualización del elemento Belt** |
| **void attachObject(Element object)**<br>*Método para depositar un objeto sobre la cinta transportadora* |
| **void detachObject(Element object)**<br>*Método para quitar un objeto de la cinta transportadora* |
| **Métodos para el control del movimiento** |
| **void connect()**<br>*Conecta el movimiento de la cinta* |
| **void disconnect()**<br>*Detiene el movimiento de la cinta* |
| **void setVariatorFrequency(int variatorFrequency)**<br>*Establece la frecuencia del movimiento de la cinta transportadora* |
| **int getVariatorFrequency()**<br>*Devuelve el valor de la frecuencia del movimiento de la cinta transportadora* |
| **double getNumberOfPulses()**<br>*Devuelve el número de pulsos del encoder de la cinta transportadora* |
| **double getLinearSpeed()**<br>*Devuelve la distancia lineal de la cinta transportadora en m/seg* |

```
4   // Instanciación de un robot Scara
5   final AbstractRobot robot = new RobotScaraOmron();
6
7   // Conexión con el robot real. Los valores del hostIP, número de puerto,
8   // usuario y contraseña han sido omitidos por razones de seguridad.
9   try{
10    robot.connect(hostIP, portNumber, userName, password);
11  }catch (Exception e){System.err.println(''Could not connect to robot'');}
12
13  // Se mueven los robots a la posición "home"
14  try{
15  robot.home();
16  }catch(Exception e){System.err.println(''Error going home!'';}
17
```

```
18    // Definición de una trayectoria lineal entre el punto inicial "origin" y el final "endPoint"
19    double [] origin = { 0, 0, 0, 0 };
20    double [] endPoint = { 45, 45, 0.02, 180 };
21    double duration = 5.0;
22    robot.setLinearTrajectory(origin, endPoint, duration);
23
24    // Definición de una restricción de tipo "Plane" en Y = 0.1
25    robot.addRestriction(new PlaneRestriction(COORDINATE.Y, 0.1, false) {
26      public void action(double x, double y, double z) {
27        System.out.println(``Error: Point violates the Y < 0.1 restriction:\n'' +
28        ``X ='' + x + ``, Y ='' + y + ``, Z ='' + z);
29      }
30    });
31
32    // Creación del entorno virtual 3D
33    Group robotGroup = new Group();
34    robotGroup.setTransformation(Matrix3DTransformation.rotationZ(Math.PI));
35    robotGroup.setPosition(new double[] { 0, 0, 0 });
36    final DrawingPanel3D mPanel3D =new DrawingPanel3D(DrawingPanel3D.IMPLEMENTATION_JAVA3D);
37    mPanel3D.getVisualizationHints().setDecorationType(VisualizationHints.DECORATION_CUBE);
38    mPanel3D.setPreferredMinMax(-1.5, 1.5, -1.5, 1.5, 0.0, 2.5);
39    mPanel3D.getCamera().adjust();
40    mPanel3D.addElement(robotGroup);
41    robot.addToViewGroup(robotGroup);
42
43    //Visualización de la restricción de tipo Plano
44    ElementBox planeRestriction = new ElementBox();
45    planeRestriction.setXYZ(-0.4, -0.4, 0.7);
46    planeRestriction.setSizeXYZ(0.8, 0.02, 1.8);
47    planeRestriction.getStyle().setFillColor(Color.RED);
48    robotGroup.addElement(planeRestriction);
49
50    JFrame mFrame = new JFrame(``Scara Robot view'');
51    mFrame.getContentPane().setPreferredSize(new Dimension(400, 400));
52    mFrame.getContentPane().add(mPanel3D.getComponent(),BorderLayout.CENTER);
53    mFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
54    mFrame.pack();
55    mFrame.setVisible(true);
56
57    // Ejecución de la simulación
58    new Thread() {
59      public void run() {
60        double step = 0.1;
61        try {
62          robot.trajectoryHome();
63          while (!robot.trajectoryFinished(step)) {
64            robot.trajectoryStep(step);
65            Thread.sleep((long) (1000 * step));
66            mPanel3D.update();
67          }
68        }catch (Exception exc) {exc.printStackTrace();}
69      }
70    }.start();
71  }// final del main()
```

72 } // *final de la clase*

En la Línea 5 del Código 4.6 se define el robot Scara de cuatro GDL, de la línea 9 a la 11 se establece la conexión remota con el robot real, y de la línea 14 a la 16 se mueven ambos robots a su posición *home*.

Una vez hecho esto se define una trayectoria lineal entre dos puntos dados en coordenadas articulares (ver líneas de la 19 a la 22), y se añade una restricción de tipo Plano al movimiento de los robots (líneas 25 a la 30). Esta restricción representa un plano en $Y = 0{,}1$ que el robot no debe traspasar.

El código comprendido entre las líneas 33 y 55 corresponde al diseño de la visualización de la simulación completa, es decir, la visualización del robot junto a su restricción plana en un entorno tridimensional.

Finalmente de las líneas 58 a la 70 se crea un *thread* de Java que moverá a los robots (tanto al virtual como al real conectado por remoto), siguiendo la trayectoria establecida con un intervalo de tiempo de 0.1 segundos. En la Figura 4.17 se puede observar el resultado de este laboratorio virtual.

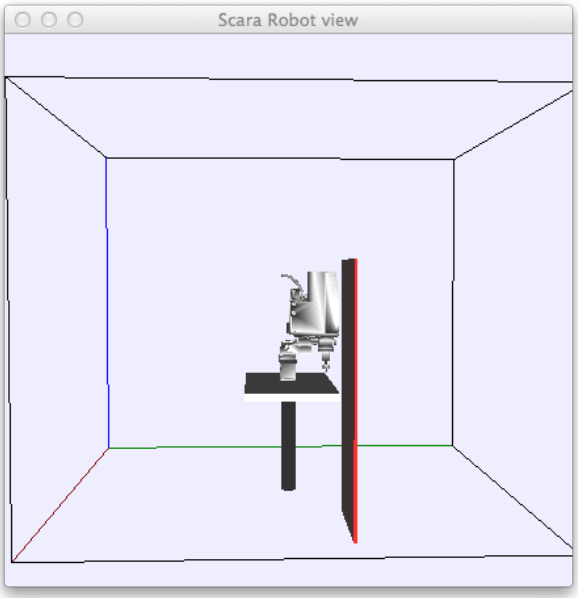



Figura 4.17: Visualización de un laboratorio virtual integrado por un robot Scara

Hay que enfatizar que la variable *robot* dada en este código ha sido definida de forma genérica como un elemento *AbstractRobot*. Esta clase abstracta tal y como se ha detallado en la Sección 4.3 encapsula la API que todos los robots deben implementar. Esto significa que se necesitan pequeños cambios en el código para programar las mismas tareas para diferentes robots. Por ejemplo, si la

línea 5 se reemplaza por la siguiente:

*final AbstractRobot robot = new RobotTX60L();*

y se adapta la línea 10 con las variables de conexión específicas y las líneas 19 y 20 con las coordenadas correspondientes de los puntos de la trayectoria del robot, entonces se obtendría la misma simulación pero con un robot TX60L de seis GDL. El resto de código podría ser reutilizado exactamente de la misma manera sin ninguna otra modificación.

Con este ejemplo se puede afirmar que la operación con elementos robóticos a un alto nivel junto a la posibilidad de reutilización de código son unas de las principales ventajas a destacar de esta nueva librería de Java. En el Capítulo 6 se proporcionan diferentes ejemplos de laboratorios robóticos virtuales y remotos de mayor complejidad.

# Capítulo 5

# Integración de la API en el software EJS

En este capítulo se presenta la integración de la nueva librería de robótica, presentada a lo largo de esta tesis, en el software de simulación EJS.

El principal beneficio de esta nueva integración es que los usuarios de EJS pueden desarrollar laboratorios robóticos virtuales y remotos sin necesitar de tener altos conocimientos ni de Java ni de Robótica.

## 5.1. Elementos Robóticos en EJS

Una vez desarrollada la nueva librería robótica para el diseño de laboratorios virtuales y remotos (presentada en los capítulos anteriores), se decidió integrarla en el software de simulación EJS (ver Sección 2.6).

La integración de dicha librería en EJS presenta múltiples ventajas. El diseño de los laboratorios es mucho más sencillo haciendo uso de esta herramienta, ya que reduce el número de líneas de código que el usuario debe implementar y sobre todo simplifica la configuración de la visualización del laboratorio virtual. Esta herramienta tiene integrados por defecto todos los elementos básicos de la librería OSP 3D que pueden ser utilizados por el usuario en sus simulaciones con una simple acción de "arrastrar y soltar". En general, EJS permite tanto a los programadores de Java como a los que no tienen un alto conocimiento de Java diseñar laboratorios robóticos sin necesidad de tener altos conocimientos ni de Java ni de Robótica.

La librería ha sido embedida en EJS mediante la definición de nuevos elementos del Modelo. Recordemos que los elementos del Modelo de EJS permitían a los usuarios acceder fácilmente a

una serie de librerías externas con diferentes funcionalidades (numéricas, programación en paralelo, otros hardware,...). En el diagrama que se muestra en la Figura 5.1 se puede observar además de la estructura básica del software EJS, esta lista de elementos del modelo que EJS tiene disponibles, incluyendo los correspondientes a la integración de esta nueva librería.

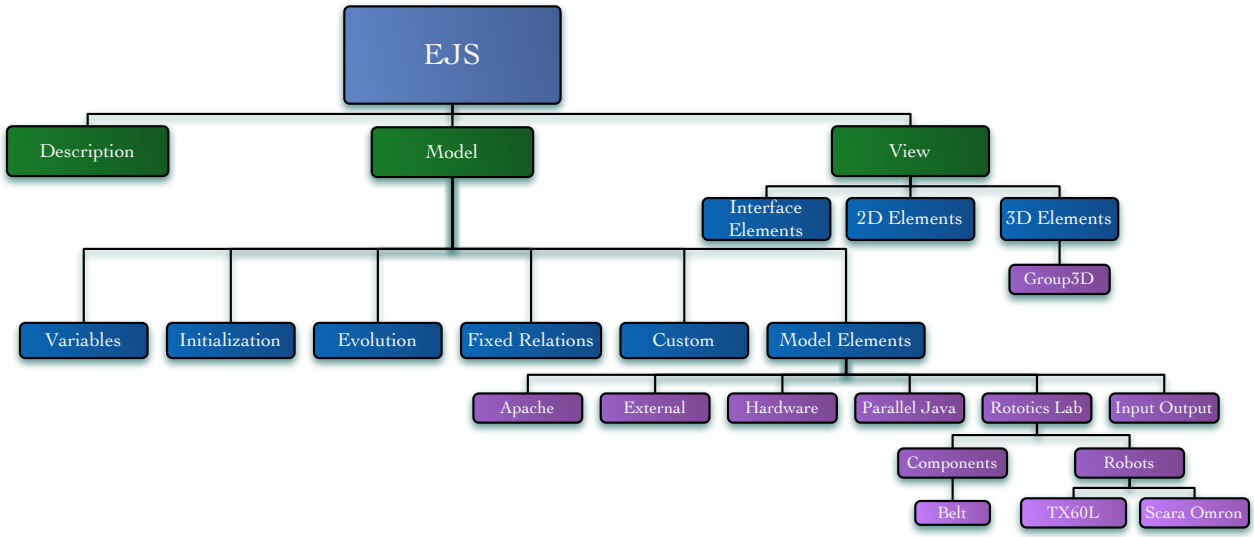



Figura 5.1: Interacción de la API y EJS

En conclusión, estos nuevos elementos robóticos enfocados en el desarrollo de simulaciones de laboratorios virtuales y remotos con robots manipuladores, *Robotics Lab, robot TX60L, robot Scara y belt*, incorporan todas las operaciones y propiedades de la nueva API.

En el resto del capítulo se describe con detalle cada uno de estos nuevos elementos robóticos, así como el procedimiento que se debe seguir para utilizarlos en el diseño de simulaciones robóticas.

### 5.1.1. Definición de un elemento robótico en EJS

Desde que el soporte de robots ha sido embebido en EJS en la forma de *elementos del modelo*, un usuario puede disponer de ellos en cualquiera de sus simulaciones siguiendo el mismo procedimiento que emplearía con el resto de elementos del modelo. Estos nuevos elementos del modelo son los siguientes:

- *RoboticsLab*: Elemento que encapsula todas las características y funcionalidades de la clase RoboticsLab presentada en la Sección 4.2
- *RobotTX60L*: Elemento que representa la implementación en la librería del robot TX60L de Stäubli (ver Subsección 4.5.1).

- *RobotScaraOmron*: Elemento que representa la implementación del robot Scara de Omron (ver Subsección 4.5.2).
- *Belt*: Elemento que representa la implementación de una cinta transportadora (ver Subsección 4.5.3).

Los nuevos elementos robóticos se encuentran disponibles dentro de la sección *RoboticLabs*, situada en el panel de trabajo *Model-Elements* de EJS, tal y como se puede observar en la Figura 5.2.

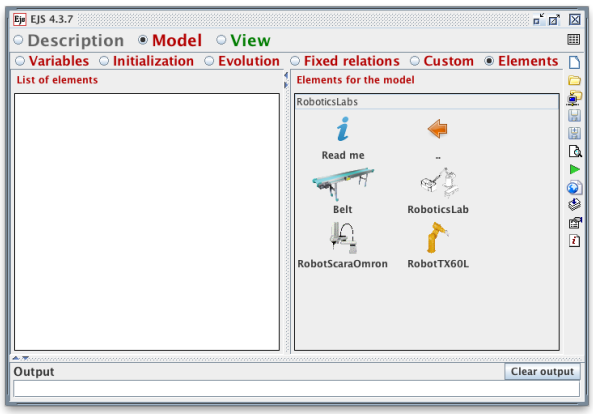


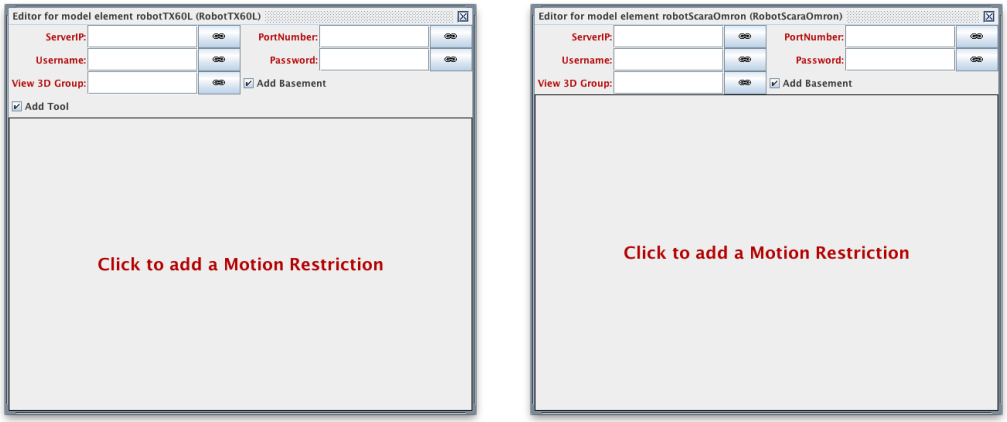

Figura 5.2: Elementos robóticos del modelo EJS

Para añadir uno de estos nuevos elementos a la simulación basta con que el usuario seleccione el icono correspondiente y mediante la acción de "arrastrar-soltar" lo añada en el espacio en blanco situado a la izquierda de la paleta de elementos del modelo denominado *"List of elements"* (ver Figura 5.2). Entonces, se le asigna un nombre para completar su definición y automáticamente el programa creará un nuevo objeto de Java que proporciona acceso internamente a un robot o componente de un laboratorio robótico, según corresponda con el elemento elegido.

El usuario puede añadir a su simulación tantos robots o componentes como necesite para su diseño, independientemente del tipo, repitiendo el mismo procedimiento tantas veces como elementos necesite incorporar.

Una vez añadidos los elementos, éstos pueden ser configurados y personalizados haciendo uso de su editor, al que se accede haciendo doble click sobre el icono del elemento. Evidentemente, las opciones de configuración dependen del elemento seleccionado. En el caso de los robots *"RobotTX60L"* y *"RobotScaraOmron"* (ver Figura 5.3), ambos tienen en común, por un lado, los parámetros necesarios para establecer la conexión remota con su robot real correspondiente, en el caso del diseño de un laboratorio remoto. Estos parámetros son: el IP del servidor, *"ServerIP"*, el número de puerto, *"PortNumber"*, el nombre de usuario, *"Username"* y la contraseña, *"Password"*. Y, por otro lado, el campo *"View 3D Group"* que permitirá añadir la visualización del modelo 3D del robot

de forma automática (y animada, cuando el robot se mueva) en la vista 3D del laboratorio virtual simplemente proporcionando un elemento de la vista de tipo *"3D group"*, previamente creado en el panel *View* de EJS tal y como se explicará en la Sección 5.2. Además, ambos elementos contienen un editor de restricciones cuyo fin es facilitar el proceso de creación de estas y que será detallado en la Subsección 5.1.3.




(a) Editor del robot TX60L (b) Editor del robot Scara

Figura 5.3: Editores de los robots integrados en EJS

El resto de parámetros dependen de la configuración propia del elemento, como puede ser la posibilidad de añadir una herramienta al extremo del robot o una base donde poder situarlo.

En el caso de la cinta transportadora *"Belt"*, su configuración básicamente se traduce en la posibilidad de poder proporcionar las diferentes dimensiones del elemento, incluir o no un encoder, y representar su modelo 3D en el laboratorio virtual tal y como se puede apreciar en la Figura 5.4.

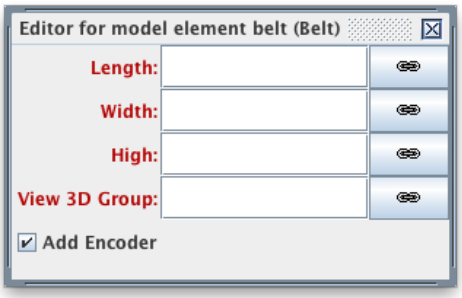



Figura 5.4: Editor de la cinta transportadora, Belt, disponible en EJS

Finalmente, el elemento *RoboticsLab* (ver Figura 5.5) sólo contiene un editor de restricciones que permitirá establecer restricciones genéricas para todos los elementos que integren el laboratorio.


Figura 5.5: Editor del elemento RoboticsLab disponible en EJS

En el momento en el que los elementos han sido añadidos y configurados en la lista de elementos de la simulación, se puede acceder a ellos desde cualquier parte del código del modelo siguiendo la metodología basada en la orientación a objetos.

Cada uno de los elementos proporciona una interfaz propia de programación estándar que incluye las instrucciones básicas y las utilidades requeridas para operar con él. Estas APIs están accesibles por parte del usuario desde la página de ayuda que todos los elementos tienen disponibles haciendo doble click sobre el elemento que se desee. En la Figura 5.6 se muestra como ejemplo la página de ayuda que ofrece el elemento *RobotScaraOmron*, en la cual se facilitan todos los métodos disponibles para este elemento y que coincidirán con los métodos que se definieron a lo largo del Capítulo 4.

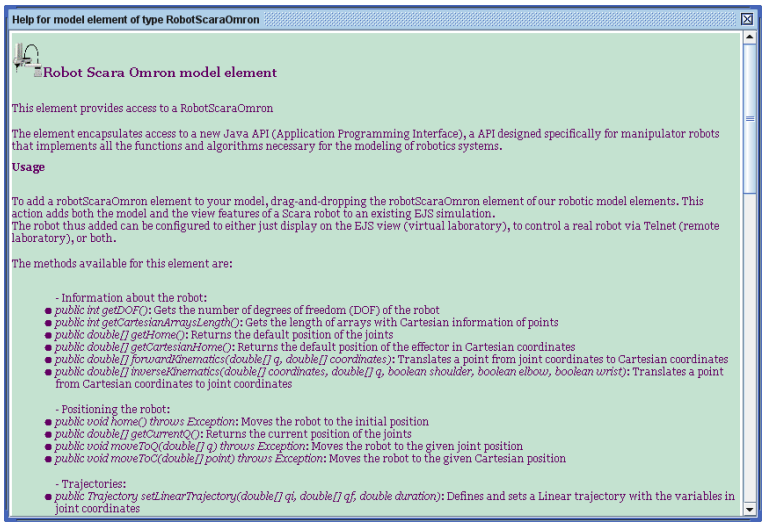



Figura 5.6: Página de ayuda del elemento del modelo “RobotScaraOmron”

### 5.1.2. Definición de trayectorias en EJS

Es posible que la operación más sencilla que un robot pueda requerir sea enviarlo a su posición inicial conocida como "home". Esta ejecución se llevaría a cabo mediante una simple instrucción:

```
try {
  robotName.home();
} catch(Exception exc) {exc.printStackTrace();}
```

donde *robotName* es el nombre que el usuario le ha asignado a este elemento en el momento de añadirlo al modelo.

Además de este método *home()*, en la Subsección 4.3.1 se detalló que la API provee otros dos métodos para mover el robot a una posición dada bien en coordenadas cartesianas o en articulares, *moveToC(double[] point)* y *moveToQ(double[] q)* (ver Tabla 4.2).

Por otro lado, si se quiere establecer movimientos más sofisticados, tales como dar instrucciones al robot para seguir un tipo de movimiento concreto, en la Subsección 4.3.2 se facilitó el conjunto de todas las posibles trayectorias integradas en la API y que están disponibles tanto para el usuario de Java como para el de EJS.

El procedimiento a seguir para establecer una trayectoria a un robot concreto en EJS sería exactamente el mismo que el que se describió en el caso de utilizar directamente la librería de Java. Por ejemplo, para definir una trayectoria lineal en EJS desde un punto inicial de coordenadas cartesianas `(0,0,0,0)` a un punto final `(1,2,0,0)` en 2 segundos bastaría con escribir en el modelo las siguientes sentencias:

```
double[] startPoint = {0,0,0,0};
double[] endPoint = {1,2,0,0};
double duration =  2;
robotName.setCartesianLinearTrajectory(startPoint, endPoint, duration);
```

Una vez establecida la trayectoria, el movimiento del robot se ejecuta dentro del Modelo de EJS en el subpanel *Evolution*, apoyándose en los métodos dados en la Tablas 4.4 y resumidos también en la página de ayuda del elemento.

En la Sección 5.3 se facilita un ejemplo sencillo de un laboratorio robótico formado por un robot Scara al que se le establece una trayectoria lineal entre dos puntos.

### 5.1.3. Definición de restricciones en EJS

Las restricciones que pueden surgir a la hora de trabajar con robots han sido clasificadas en dos tipos: las restricciones físicas propias del robot y las restricciones de movimiento. En la Subsección 4.3.3 se analizó dicha cuestión y se detalló cómo se ha resuelto este problema en la nueva API. Para ello, se han definido una serie de métodos para la resolución de las restricciones físicas y

se han implementado diferentes tipos de restricciones básicas (planos, cajas, esferas,...) en las que el usuario puede apoyarse para la definición de las restricciones de movimiento.

En EJS las restricciones físicas se tratan de la misma manera que se hacía anteriormente, es decir, haciendo uso directamente de los métodos dados en la Tabla 4.6 (los cuales también se facilitan en la página de ayuda de los elementos robóticos).

Sin embargo, el proceso de definir las restricciones de movimiento se ha simplificado considerablemente con la integración de la librería en EJS, resultando mucho más simple para el usuario. Esta simplificación se debe a que los elementos del modelo presentan un editor para definir este tipo de restricciones y mediante la ejecución de cinco pasos elementales se pueden establecer tantas restricciones como se necesite para cubrir las necesidades del entorno de una forma cómoda y sencilla.

Una vez que el elemento ha sido añadido a la simulación, en la Figura 5.7 se facilita un esquema con el procedimiento que se debe seguir para establecer una restricción de movimiento en EJS y cuyos pasos se detallan a continuación:

- 1º Paso: Acceder al editor del elemento haciendo doble click sobre el icono del mismo .
- 2º Paso: Hacer click sobre el editor de restricciones *" Click to add a Motion Restriction"* que aparce en el editor del elemento.
- 3º Paso: En la pantalla emergente que aparece automáticamente tras el segundo paso se debe facilitar el nombre de esta nueva restricción.
- 4º Paso: Seleccionar el tipo de restricción que se quiere definir entre los casos posibles: *Plane, Box, Block, Sphere, Custom.*
- 5º Paso: Según el tipo de restricción seleccionada en el paso anterior, aparecerá un editor con un código predefinido y donde el usuario sólo tiene que facilitar algunos parámetros para completar su definición. Por ejemplo, en el caso de un plano se deberán facilitar las coordenadas del mismo en el método *boolean allowsPoint (double x, double y, double z)* , y la acción a llevar a cabo en caso de violarse la restricciónen en el método *void action (double x, double y, double z).*

Finalmente, es importante resaltar que cuando se establece una restricción de movimiento en el elemento *RoboticsLab*, automáticamente esta se convierte en una restricción general que deben satisfacer todos los elementos del laboratorio, independientemente de las restricciones que cada uno de los elementos tenga establecidas en sus correspondientes editores.

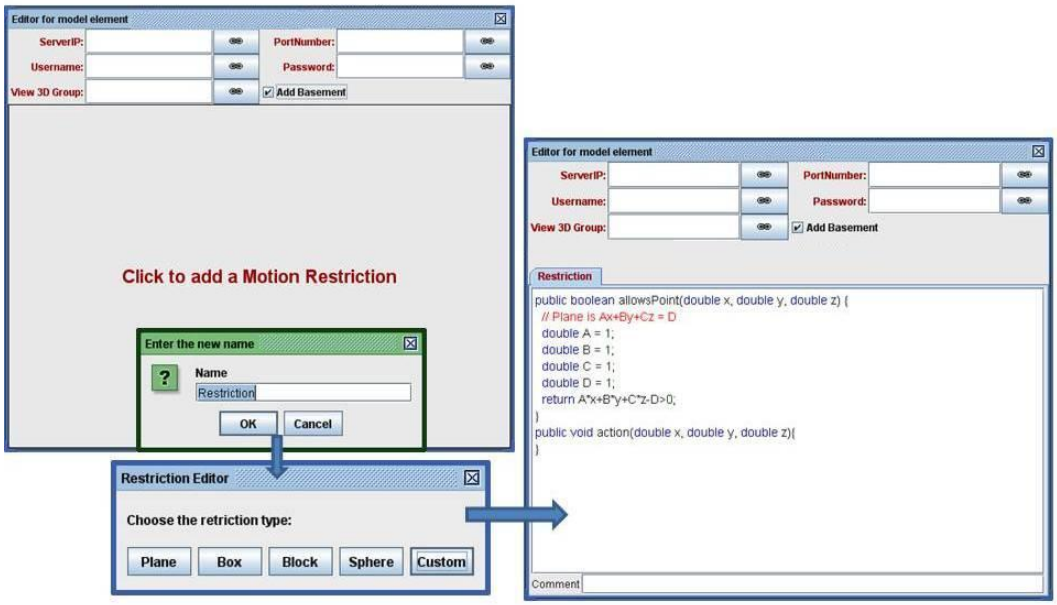



Figura 5.7: Definición de una restricción en EJS

## 5.2. Procedimiento para diseñar una simulación robótica en EJS

En general, a la hora de crear una simulación robótica en EJS se debe seguir básicamente el procedimiento que se describe a continuación.

1) *Configuración de la vista:*

Como paso inicial, el usuario utiliza EJS para crear la interfaz gráfica o vista de la simulación apoyándose en el editor que presenta para ello. Este simple editor contiene una serie de elementos predefinidos basados en los elementos de la libreria OSP 3D (ver Subsección 4.3.5) que facilitan al usuario la construcción de dicha interfaz.

Estos elementos se añaden a la vista simplemente con la acción de "arrastrar - soltar" sin necesidad de conocer los detalles concretos de implementación de cada uno de los elementos. Además, cada elemento proporciona su propio editor para poder configurar sus propiedades fácilmente.

En la Figura 5.8 se muestra un ejemplo de la construcción de la interfaz gráfica en el caso del diseño de un laboratorio virtual sencillo. En este caso, se han incorporado dos elementos para poder visualizar el modelo del robot. El primero de ellos es un elemento de tipo *DrawingPanel3D*, el cual proporciona el entorno 3D donde se representará el modelo del robot y su espacio de trabajo. Sobre este elemento, se ha añadido otro elemento de tipo *Group3D*

para visualizar el modelo del robot, definido como *robotGroup*. En este punto, se deben incluir tantos elementos de tipo *Group3D* como robots o componentes robóticas se quieran añadir al laboratorio. Por ejemplo, en este caso se ha añadido otro de estos elementos, *Environment*, para la configuración del entorno del robot.

La interfaz puede y normalmente incluye otros elementos para el control de la simulación (como pueden ser botones de acción, barras deslizadoras, campos para introducir valores, etc.) y paneles básicos para agruparlos. Estos elementos de control de la vista serán conectados con variables definidas posteriormente en el modelo y que permitirán una interacción entre la vista y el modelo de la simulación. Por ejemplo, si se considera un número n de barras deslizadoras (las cuales permiten visualizar y modificar un valor numérico de manera gráfica dentro de un rango establecido), estas serán conectadas con un conjunto n de variables numéricas $q_i$ que ayudará al usuario a interaccionar con las diferentes articulaciones del robot.

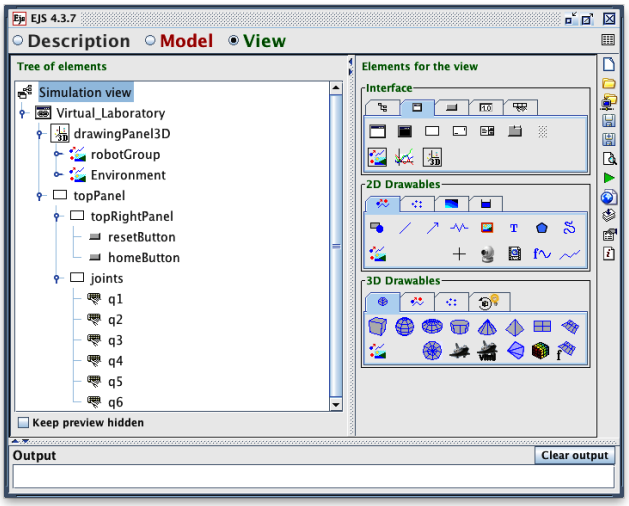



Figura 5.8: Configuración de la vista de EJS.

II) *Definición del modelo:*

El segundo paso para crear una simulación es la definición de su modelo. El modelo en EJS está dividido en seis paneles: *Variables*, *Initialization*, *Evolution*, *Fixed relations*, *Custom* y *Elements* (ver Figura 5.2). Partiendo de estos paneles, la configuración del modelo se llevaría a cabo de la siguiente manera:

*a*) *Definición de los elementos del laboratorio:* En primer lugar, se deben añadir todos los elementos robóticos que forman parte de la simulación que se está diseñando siguiendo el procedimiento descrito en la Subsección 5.1.1.

Una vez añadidos, los modelos 3D de estos elementos serán mostrados en la vista si el

usuario proporciona en el parámetro *"View 3D Group"* (que aparece en el editor del elemento) el nombre del correspondiente *Group3D*, previamente definido en la vista.

Además de la configuración de los parámetros de los elementos, se deberán establecer las restricciones necesarias para cada uno de los elementos siguiendo el procedimiento descrito en la Subsección 5.1.3.

*b*) *Definición de las variables del laboratorio:* En segundo lugar, en el panel *Variables* se define el conjunto de todas las variables necesarias para el desarrollo de la simulación, como por ejemplo, las variables $q_i$ mencionadas anteriormente en la configuración de la vista.

*c*) *Configuración de la lógica del modelo:* En tercer lugar, una vez añadidos los elementos robóticos y definidas las variables necesarias, es el momento de rellenar la lógica para la simulación usando el resto de paneles que ofrece el Modelo de EJS: *Initialization*, *Evolution*, *Fixed relations* y *Custom*. En cualquiera de estos paneles, el autor puede usar sentencias de Java y cualquier constructor o métodos definidos en las correspondientes APIs de los elementos robóticos de la simulación.

III) *Ejecución de la simulación:*

Finalmente, con un simple click en el botón *"Play"* se crea todo el código auxiliar, se compila y se ejecuta la simulación. Se crea el modelo, incluyendo la instanciación de los objetos robóticos, y la vista del laboratorio virtual aparecerá automáticamente en la pantalla. Los elementos de la vista diseñados para el control en las operaciones serán ejecutados en la parte lógica del modelo, y la visualización del robot mostrará los cambios de estado.

En la siguiente sección se facilita un primer ejemplo completo de una simulación robótica diseñada siguiendo este procedimiento y donde se verá el detalle de cómo implementar cada uno de estos pasos.

## 5.3. Ejemplo básico de una simulación robótica diseñada en EJS

En esta sección se va a llevar a cabo el diseño de la misma simulación dada en la Sección 4.6, pero en este caso desarrollada en la herramienta EJS. El objetivo de definir la misma simulación es simplemente para poder observar las ventajas que ofrece la integración de la nueva librería de Java en el entorno de EJS a la hora de diseñar estos laboratorios robóticos.

Siguiendo el procedimiento descrito en la Sección 5.2, en primer lugar se configura la vista de la simulación (ver en la Figura 5.9). En este caso se ha incluido un elemento "drawingPanel3D" donde se han definido dos elementos de tipo "Group3D", uno para representar el modelo del robot, el "ScaraGroup", y otro para representar las paredes y el suelo del laboratorio, el "Enviroment", cuya

finalidad es aportar una apariencia más realista a la simulación. Dentro del elemento ScaraGroup se ha incluido un elemento "Box3D" para representar de forma visual la restricción que se va a establecer para el movimiento del robot. Por otro lado, se ha incluido un panel de control, "topPanel" con dos botones de acción uno con la función "Play/Pause" y otro con la función "Reset".

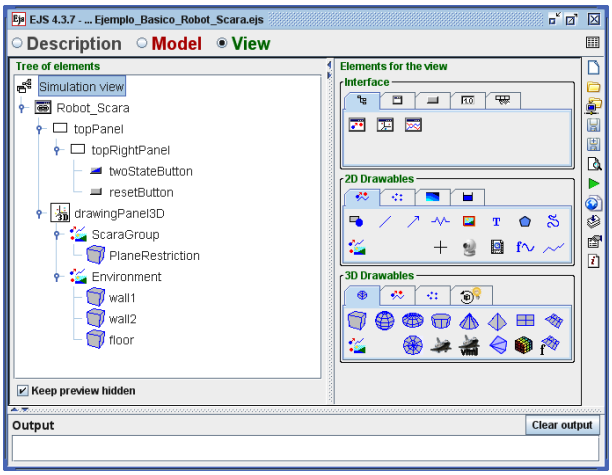


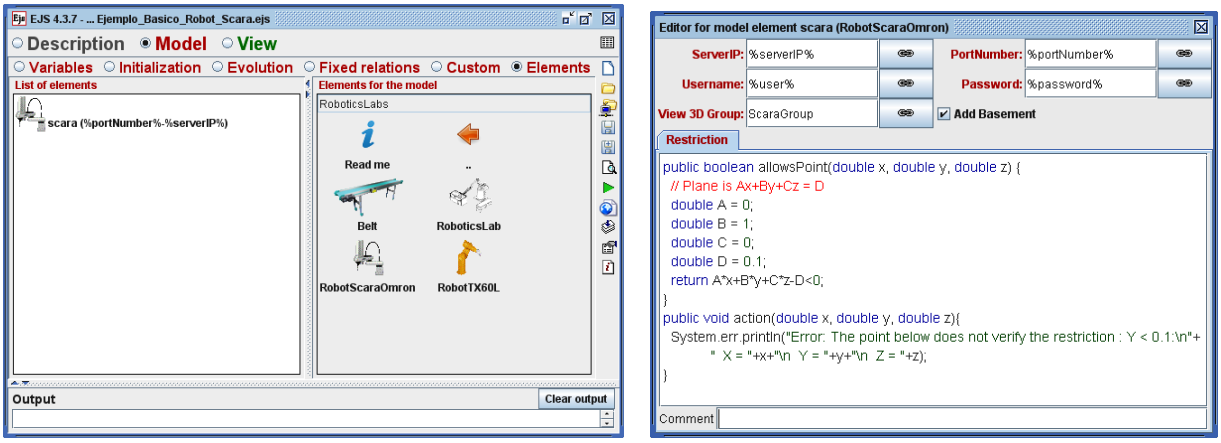

Figura 5.9: Ejemplo básico EJS - Configuración del panel "View"

En segundo lugar, se añade en la "List of Elements" del modelo el elemento robótico que integra esta simulación, un robot Scara (ver Figura 5.10(a)), y se configuran sus propiedades particulares. En este caso, se va a añadir una base al robot, seleccionando el campo correspondiente "Add Basement" y se proporcionan los parámetros correspondientes para poder establecer la conexión con el robot real. Además, se define una restricción de tipo "Plane" en Y = 0.1 tal y como se observa en la Figura 5.10(b).

Tras añadir y configurar el robot, el siguiente paso es definir todas las variables necesarias para esta simulación en el panel "Variables" (ver Figura 5.11). Las variables correspondientes a los parámetros de conexión con el robot real se han omitido por razones de seguridad. Además de estos parámetros, se han definido los puntos para la trayectoria y la duración de la misma.

A continuación, se termina la configuración de la lógica del modelo. En el panel "Initialization" (ver Figura 5.12(a)) se establece la conexión con el robot real, se mueve el robot a su posición "home" y se establece una trayectoria lineal entre los dos puntos definidos previamente en el panel de "Variables". En el panel "Evolution" se lleva a cabo la programación de la evolución del movimiento del robot a través de la trayectoria dada tal y como se puede observar en la Figura 5.12(b).

Finalmente, una vez configurados todos los paneles, sólo quedaría ejecutar la simulación obteniendo los resultados dados en la Figura 5.13(a). Durante la ejecución de la trayectoria, el robot viola la restricción previamente establecida, tal y como se puede observar en la Figura 5.13(b).




(a) Lista de Elementos definidos (b) Configuración del editor del robot Scara

Figura 5.10: Ejemplo básico EJS - Configuración del panel “Model - Elements”

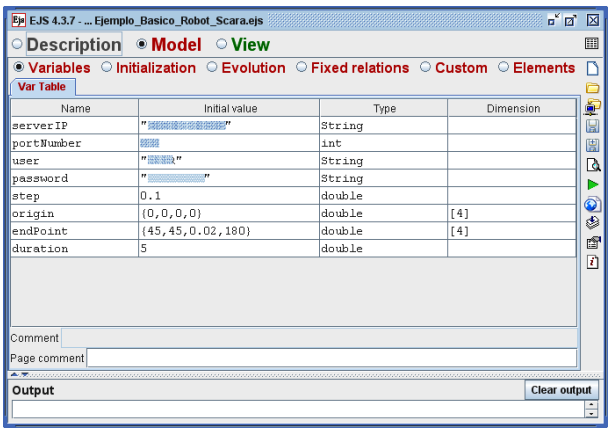
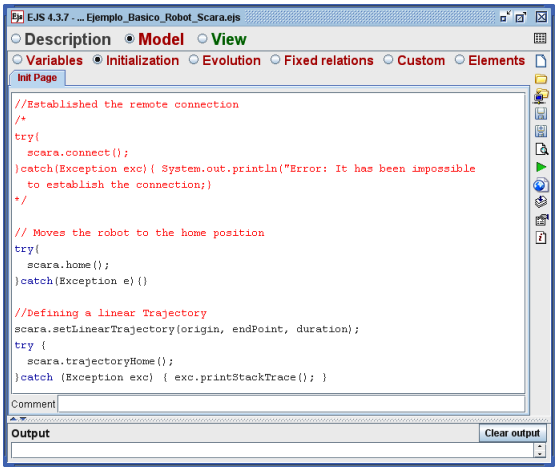


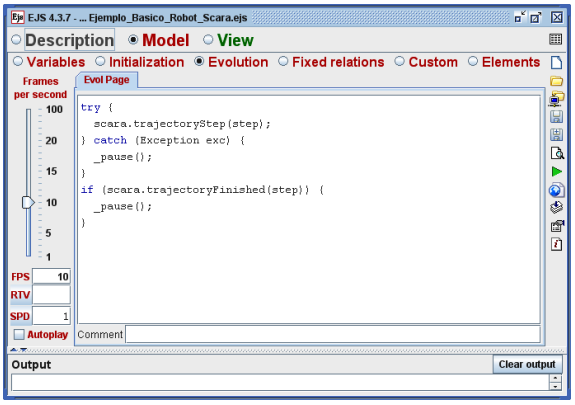

Figura 5.11: Ejemplo básico EJS - Configuración del panel “Model - Variables”

Automáticamente, el robot detiene su movimiento y como consecuencia un mensaje de error con los datos exactos del punto que no satisface la restricción dada aparece en la consola (ver Figura 5.13(c)).

En definitiva, con esta simulación inicial se puede concluir que la integración de la nueva librería robótica en el software EJS ofrece muchas ventajas al usuario ya que el diseño de los laboratorios se lleva a cabo de una forma muy metódica, sin requerir altos conocimientos ni de Java ni de Robótica y sin implementar demasiadas líneas de código, lo cual permite al usuario ahorrar tiempo y esfuerzo.

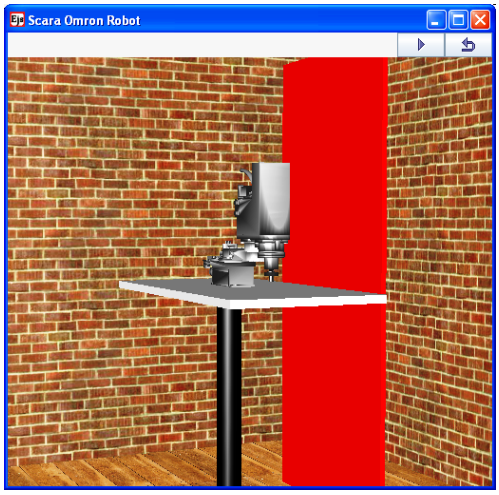



(a) Configuración del panel “Model - Initialization”

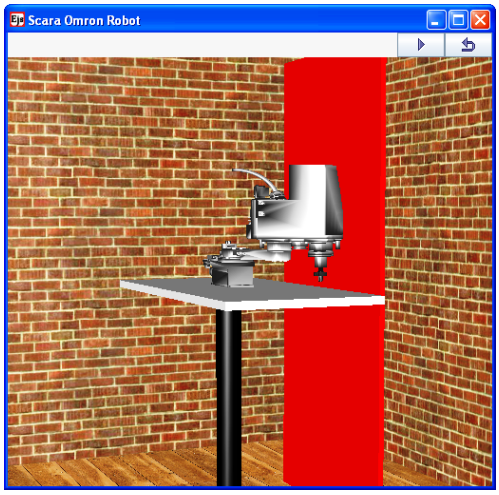



(b) Configuración del panel “Model - Evolution”

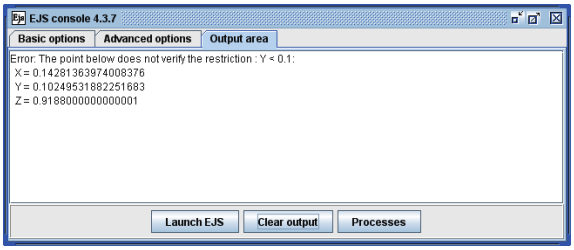
Figura 5.12: Ejemplo básico EJS - Configuración de los paneles “Model - Initialization” & “Model - Evolution”




(a) Posición Inicial




(b) Ejecución




(c) Error: Violación de la restricción

Figura 5.13: Ejemplo básico EJS - Visualización de un laboratorio virtual integrado por un robot Scara

# Capítulo 6

# Creación de laboratorios virtuales y remotos de Robótica

En este capítulo se facilitan diferentes ejemplos de laboratorios robóticos virtuales y remotos en entornos estáticos y dinámicos haciendo uso de la nueva libería Java que en esta tesis se presenta y de su integración en el software de simulación EJS.

Los usuarios pueden crear fácilmente objetos que representen a uno o varios robots, definir trayectorias y restricciones de movimiento, detectar posibles colisiones con él mismo o con el entorno que lo rodee, visualizar o no en un espacio tridimensional las operaciones establecidas para estos robots (*laboratorios virtuales*), u operar con el robot real estableciendo una conexión remota con el equipo real (*laboratorios remotos*). Finalmente, se pueden combinar ambos modos de trabajo para definir un laboratorio robótico virtual y remoto al mismo tiempo.

## 6.1. Laboratorios robóticos virtuales y remotos en Java

En esta sección se van a presentar diferentes ejemplos de laboratorios robóticos virtuales y remotos diseñados con la nueva librería de Java. Con estos ejemplos se pretende mostrar todas las ventajas y principales características que posee dicha librería y que se han ido presentando en el transcurso de esta tesis.

Según el tipo de elementos que se utilicen en el diseño de un laboratorio robótico, éste puede ser clasificado como:

I) *Laboratorio robótico elemental:*

Se considera laboratorio robótico elemental aquel que ha sido diseñado a partir de los elementos previamente integrados en la nueva librería, dos robots manipuladores de diferentes GDL y una cinta transportadora (detallados en la Sección 4.5).

Mediante el uso de los métodos y propiedades de cada uno de estos elementos el usuario puede crear un laboratorio robótico virtual y/o remoto sencillo, compuesto por uno o varios elementos que actúan de forma independiente, es decir, los movimientos entre los elementos del laboratorio se llevan a cabo independientemente, sin ninguna coordinación entre ellos. De esta forma, se tendría en cuenta la posible autocolisión de un robot con él mismo pero sería imposible de detectar la colisión entre él y el resto de elementos del laboratorio.

Un ejemplo sencillo de este tipo de laboratorios sería el que se detalló en la Sección 4.6.

II) *Laboratorio robótico complejo:*

Se considera laboratorio robótico complejo aquel que además de los elementos utilizados en el caso de un laboratorio robótico elemental hace uso del elemento *RoboticsLab* (ver Sección 4.2). Este elemento permite combinar y relacionar de forma eficaz varios robots o componentes robóticas dentro de un mismo laboratorio tal y como se representa en el esquema dado en la Figura 6.1.

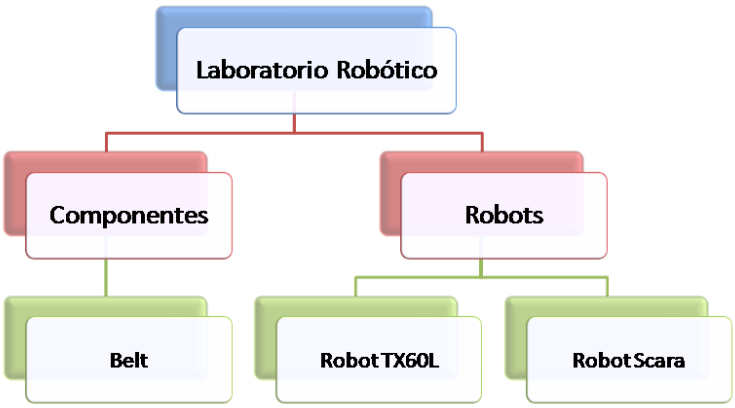



Figura 6.1: Relación entre los diferentes elementos integrados en la librería

Entonces, el uso del elemento *RoboticsLab* en el laboratorio ofrece al usuario las siguientes posibilidades:

- Permite establecer restricciones generales que todos los elementos deben verificar independientemente de las suyas propias.

- El movimiento de los elementos puede llevarse a cabo de dos maneras distintas. Por un lado, de manera análoga al caso de un laboratorio elemental, el movimiento de los robots puede realizarse de forma independiente usando directamente los métodos *moveToQ(double [] q)*, *moveToC(double [] point)* y *trajectoryStep(double deltaT)* (dados en las Tablas 4.2 y 4.4). En este caso, el robot comprobará que verifica sus propias restricciones y que no autocolisiona con él mismo. Por otro lado, el usuario puede apoyarse en el elemento *RoboticsLab* para mover el robot. Este elemento ofrece unos métodos análogos a los anteriores: *moveRobot(AbstractRobot robot, double [] q)*, *moveRobotC(AbstractRobot robot, double [] point)* y *trajectoryStep(AbstractRobot robot, double step)* (dados en la Tabla 4.1 en los que además de verificar sus propias restricciones y la posible autocolisión se comprueba que este elemento verifica también las restricciones generales establecidas en el laboratorio y que al moverse no existe colisión con ningún otro elemento del laboratorio.
- Facilita la configuración del entorno virtual 3D compuesto por todos los elementos del laboratorio.

En definitiva, si el usuario quiere diseñar un laboratorio con un único elemento o con varios que actúan independientemente no sería necesario hacer uso del elemento *RoboticsLab* y bastaría con utilizar directamente los métodos que tienen integrados cada uno de los elementos, tal y como se mostró en el ejemplo de la Sección 4.6. Sin embargo, para simulaciones más complejas compuestas por más de un elemento que trabajan en proximidad y en las que se quiere considerar la coordinación de todos estos elementos, sería necesario incorporar al laboratorio un elemento *RoboticsLab* y emplear sus métodos tal y como se mostrará en el resto de la sección.

### 6.1.1. Ejemplo 1: Diseño de un laboratorio virtual en Java compuesto por 3 robots

En este ejemplo se presenta el diseño de un laboratorio virtual compuesto por dos robots Scaras, un robot TX60L y una cinta transportadora, en un entorno dinámico.

En primer lugar, se define el elemento RoboticsLab al que se le irán añadiendo el resto de elementos que componen dicho laboratorio haciendo uso de los métodos dados en la Tabla 4.1, tal y como se muestra en el Código 6.1.

Código 6.1: Laboratorio Virtual - Definición de los elementos

```
1    //STEP 1: Defining the elements of the robotics lab
2      //A RoboticsLab class
3    final RoboticsLab roboticsLab = new RoboticsLab();
4      // Adding a Belt element to the lab
5    final Belt belt = new Belt(true);
6    double[] beltPosition = { 0, 0, 0 };
7    roboticsLab.addComponent(belt, beltPosition, true);
8
```

```
9       // Adding two Scara Omron robots to the lab
10    final AbstractRobot scara = new RobotScaraOmronWithComponents(true);
11    final Group scaraGroup = new Group();
12    scaraGroup.setXYZ(-0.45, -belt.getLength()/2, 0);
13    scaraGroup.setTransformation(Matrix3DTransformation.rotationZ(Math.PI/2));
14    roboticsLab.addRobot(scara, scaraGroup, true);
15
16    final AbstractRobot scara2 = new RobotScaraOmronWithComponents(true);
17    final Group scaraGroup2 = new Group();
18    scaraGroup2.setXYZ(0.45, -belt.getLength()/2, 0);
19    scaraGroup2.setTransformation(Matrix3DTransformation.rotationZ(Math.PI/2));
20    roboticsLab.addRobot(scara2, scaraGroup2, true);
21
22      // Adding a TX60L robot to the lab
23    final AbstractRobot tx60l = new RobotTX60LWithComponents(true, true);
24    final Group tx60lGroup= new Group();
25    tx60lGroup.setXYZ( -0.3, 0.45, 0);
26    roboticsLab.addRobot(tx60l, tx60lGroup, true);
```

En segundo lugar, siguiendo las sentencias dadas en el Código 6.2 se lleva a cabo la configuración del entorno virtual 3D del laboratorio compuesto por los elementos definidos en el paso anterior (ver Líneas 2-12). Además, en este paso se añaden otros dos objetos de tipo *ElementBox* (unas pequeñas cajas) sobre las mesas de los robots Scaras (ver Líneas 15-25).

Código 6.2: Laboratorio Virtual - Visualización 3D del laboratorio virtual

```
1     //STEP 2: Creating a virtual display of the virtual laboratory
2     final DrawingPanel3D mPanel3D = roboticsLab.getRoboticsPanel3D();
3     mPanel3D.getVisualizationHints().setDecorationType(VisualizationHints.DECORATION_CUBE);
4     mPanel3D.setPreferredMinMax(-1.5, 1.5, -1.5, 1.5, 0.0, 2.5);
5     mPanel3D.getCamera().adjust();
6     JFrame mFrame = new JFrame(``Virtual Robotics Lab with three robots'');
7     mFrame.getContentPane().setPreferredSize(new Dimension(600, 600));
8     mFrame.getContentPane().add(mPanel3D.getComponent(), BorderLayout.CENTER);
9     mFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10    mFrame.pack();
11    mFrame.setVisible(true);
12    mPanel3D.update();
13
14    //STEP 3: Put a box on the table of Scara robots
15    final ElementBox box1 = new ElementBox();
16    box1.setXYZ(-0.45 - 0.22, -belt.getLength()/4, belt.getHigh() + 0.02);
17    box1.setSizeXYZ(0.04, 0.04, 0.04);
18    box1.getStyle().setFillColor(Color.RED);
19    mPanel3D.addElement(box1);
20
21    final ElementBox box2 = new ElementBox();
22    box2.setXYZ(0.45 + 0.22, -belt.getLength()/4, belt.getHigh() + 0.02);
23    box2.setSizeXYZ(0.04, 0.04, 0.04);
24    box2.getStyle().setFillColor(Color.RED);
25    mPanel3D.addElement(box2);
```

Una vez definidos los elementos del laboratorio y configurado el entorno 3D para su visualización,

en el resto de código se lleva a cabo la definición y ejecución de las diferentes tareas que se han diseñado para cada uno de los elementos del laboratorio (ver Código 6.3).

Inicialmente, se conecta la cinta transportadora (Línea 4) y se asignan diferentes trayectorias a cada uno de los robots. Los robots Scaras deben coger las cajas que están situadas sobre sus bases (Líneas 7 - 26) y situarlas sobre la cinta transportadora (Líneas 29 - 47). Cabe destacar que, si ambos robots se mueven simultáneamente hacia la cinta, entonces se detectará una posible colisión entre ellos y se mostrará un mensaje de error en la consola. Una vez que los objetos han sido depositados sobre la cinta, estos se mueven sobre ella y es en este momento cuando el robot TX60L los coge (Líneas 50 - 66), se los lleva a su posición home (Líneas 69 - 75) y se desconecta la cinta transportadora (Línea 76).

Código 6.3: Laboratorio Virtual - Ejecución de las tareas

```
1   // STEP 4: Running the simulation
2   new Thread() {
3     public void run() {
4       belt.connect();
5       double step = 0.1;
6       // Task 1: Attach the objects
7       final double[] homeScara = scara.getHome();
8       final double[] endPoint = { 30, 0, 0, 0 };
9       final double duration = 4.0;
10      scara.setLinearTrajectory(homeScara, endPoint, duration);
11
12      final double[] homeScara2 = scara2.getHome();
13      final double[] endPoint2 = { -30, 0, 0, 0 };
14      scara2.setLinearTrajectory(homeScara2, endPoint2, duration);
15      try {
16        scara.trajectoryHome();
17        scara2.trajectoryHome();
18        while (!scara.trajectoryFinished(step) || !scara2.trajectoryFinished(step)) {
19          if(!scara.trajectoryFinished(step))roboticsLab.trajectoryStep(scara, step);
20          if(!scara2.trajectoryFinished(step))roboticsLab.trajectoryStep(scara2, step);
21          Thread.sleep((long) (1000 * step));
22          mPanel3D.update();
23        }
24        scara.attachObject(box1);
25        scara2.attachObject(box2);
26        mPanel3D.update();
27
28        // Task 2: Scara robots go to the belt
29        double[] pointBelt = new double[] { -90, 0, 0, 0 };
30        scara.setLinearTrajectory(endPoint, pointBelt, duration - 0.35);
31        scara.trajectoryHome();
32        double[] pointBelt2 = new double[] { 75, 0, 0, 0 };
33        scara2.setLinearTrajectory(endPoint2, pointBelt2, duration);
34        scara2.trajectoryHome();
35        while (!scara.trajectoryFinished(step) || !scara2.trajectoryFinished(step)) {
36          if(!scara.trajectoryFinished(step))roboticsLab.trajectoryStep(scara, step);
37          if(!scara2.trajectoryFinished(step))roboticsLab.trajectoryStep(scara2, step);
38          Thread.sleep((long) (1000 * step));
```

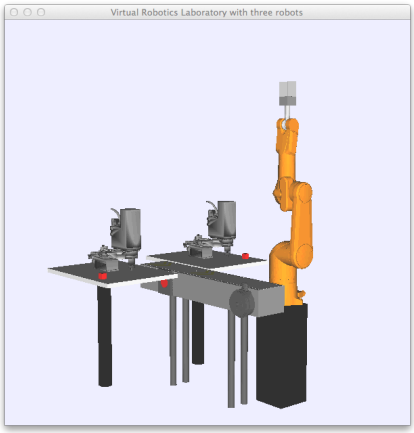
```
39            if(scara.trajectoryFinished(step)&& scara2.trajectoryFinished(step)){
40              scara.detachObject(box1);
41              belt.attachObject(box1);
42              scara2.detachObject(box2);
43              box2.setX(box2.getX() - box2.getSizeX() / 2);
44              belt.attachObject(box2);
45            }
46            mPanel3D.update();
47          }
48
49          // Task 3: TX60L robot picks up the boxes
50          double[] pointFinal = { 0, -3, 136, 0, 47, 0 };
51          double t = ((0.48 - ((box1.getX() + box2.getX()) / 2.))/belt.getLinealDistance());
52          tx60l.setLinearTrajectory(tx60l.getHome(), pointFinal, t);
53          tx60l.trajectoryHome();
54          while (!tx60l.trajectoryFinished(step)) {
55            roboticsLab.trajectoryStep(tx60l, step);
56            Thread.sleep((long) (1000 * step));
57            mPanel3D.update();
58          }
59          belt.detachObject(box1);
60          belt.detachObject(box2);
61          Element object = roboticsLab.encapsuleObject(box1, box2, 'X');
62          object.getStyle().setFillColor(Color.RED);
63          object.setSizeXYZ(object.getSizeY(), object.getSizeX(), object.getSizeZ());
64          tx60l.attachObject(object);
65          tx60l.closeTool();
66          mPanel3D.update();
67
68          //Task 4: TX60L robot returns to home position
69          tx60l.setLinearTrajectory(pointFinal, tx60l.getHome(), duration);
70          tx60l.trajectoryHome();
71          while (!tx60l.trajectoryFinished(step)) {
72            roboticsLab.trajectoryStep(tx60l, step);
73            Thread.sleep((long) (1000 * step));
74             mPanel3D.update();
75          }
76          belt.disconnect();
77        } catch (Exception exc) {exc.printStackTrace();}
78      }
79    }.start();
```

En la Figura 6.2 se muestra el resultado de este laboratorio robótico virtual y la ejecución de cada una de las tareas establecidas.

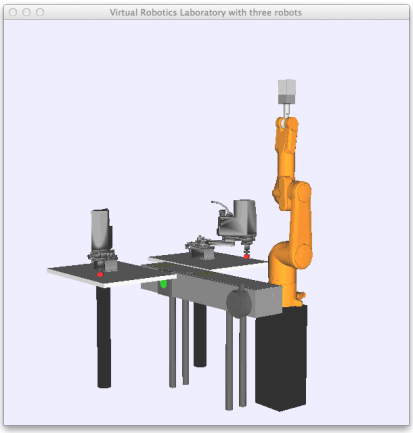
### 6.1.2. Ejemplo 2: Diseño de un laboratorio virtual y remoto en Java

En este ejemplo se presenta el diseño de un laboratorio tanto virtual como remoto, formado por un robot TX60L y una cinta transportadora, dentro de un entorno estático.

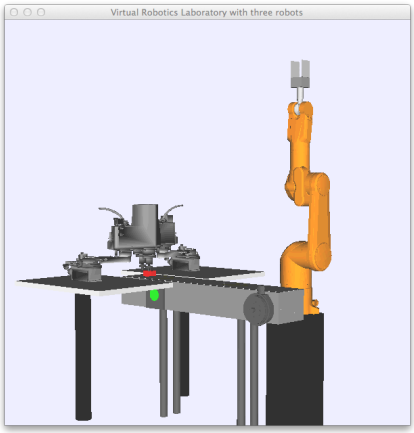
De manera análoga al ejemplo anterior, en primer lugar, en el Código 6.4 se puede observar cómo se define el elemento *RoboticsLab* (ver Línea 3) al que seguidamente se le asignan los elementos que



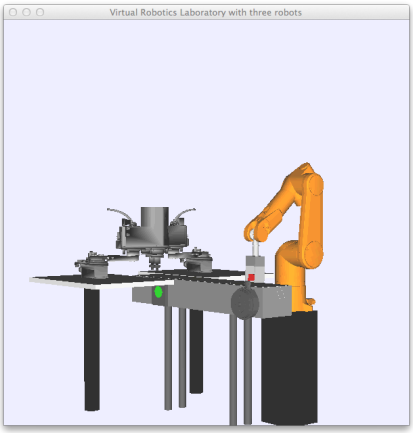

(a) Paso 1: Los robos están en sus posiciones inciales




(b) Paso 2: Los robots Scaras cogen sus objetos y se conecta la cinta




(c) Paso 3: Los robots Scaras depositan los objetos en la cinta




(d) Paso 4: El robot TX60L coge los objetos de la cinta

Figura 6.2: Laboratorio virtual en Java compuesto por tres robots

integran dicho laboratorio: una cinta transportadora y un robot TX60L (ver Líneas 6-13).

Código 6.4: Laboratorio Virtual y Remoto - Definición de los elementos

```
1   // STEP 1: Defining the elements of the robotics lab
2     // A robotics Lab class
3   final RoboticsLab roboticsLab = new RoboticsLab();
4
5     // Adding a Belt element to the lab
6   final Belt belt = new Belt(true);
7   double[] beltPosition = { 0.33, 0, 0 };
8   roboticsLab.addComponent(belt, beltPosition, true);
9
10    // Adding a TX60L robot to the lab
11  final AbstractRobot tx60l = new RobotTX60LWithComponents(true, true);
```

```
12    final Group tx60lGroup = new Group();
13    roboticsLab.addRobot(tx60l, tx60lGroup, true);
```

Definidos los elementos del laboratorio se establecen dos restricciones para el robot TX60L, una sería el plano $Y < 0.6$ y otra el plano $Y > -0.2$ tal y como se detalla en el Código 6.5:

Código 6.5: Laboratorio Virtual y Remoto - Definición de las restricciones

```
1     // STEP 2: Implementing restrictions for TX60L robot
2     tx60l.addRestriction(new PlaneRestriction(COORDINATE.Y, 0.60, false) {
3       public void action(double x, double y, double z) {
4         JOptionPane.showMessageDialog(null, ``The point below does not verify the
5         first restriction: Y < 0.6:\n''+`` X = ''+x+``\n Y = ''+y+``\n Z = ''+z,
6         ``Error with the TX60L robot: '',JOptionPane.ERROR_MESSAGE);}});
7
8     tx60l.addRestriction(new PlaneRestriction(COORDINATE.Y, -0.2, true) {
9       public void action(double x, double y, double z) {
10        JOptionPane.showMessageDialog(null, ``The point below does not verify the
11        second restriction : Y > -0.2:\n''+`` X = ''+x+``\n Y= ''+y+``\n  Z= ''+z,
12        ``Error with the TX60L robot:'',JOptionPane.ERROR_MESSAGE);}});
```

En tercer lugar, se lleva a cabo la configuración del laboratorio remoto. Para ello, basta con establecer la conexión con el robot real según se muestra en el Código 6.6. Los datos de conexión (IP, número de puerto, usuario y contraseña) han sido omitidos por razones de seguridad.

Código 6.6: Laboratorio Virtual y Remoto - Conexión con el robot real

```
1     // STEP 3: Connecting with a real robot: hostIP, portNumber, userName, password
2     try {
3       tx60l.connect( ``serverIP'', ``portNumber'', ``user'', ``password'');
4     } catch (Exception e) {}
```

A continuación, se completa el diseño del laboratorio virtual con la configuración de la visualización del entorno 3D siguiendo el Código 6.7. En este código además de añadir la visualización del laboratorio robótico integrado por el robot TX60L y la cinta transportadora (ver Líneas 2-13), se incorpora la visualización de las dos restricciones establecidas para el robot (ver Líneas 16-26) y se añade un objeto de tipo *ElementBox* que posteriormente se depositará sobre la cinta transportadora (ver Líneas 29-32).

Código 6.7: Laboratorio Virtual y Remoto - Visualización 3D del laboratorio virtual

```
1     // STEP 4: Creating a virtual display of the virtual laboratory
2     final DrawingPanel3D mPanel3D = roboticsLab.getRoboticsPanel3D();
3     mPanel3D.getVisualizationHints().setDecorationType(
4       VisualizationHints.DECORATION_CUBE);
5     mPanel3D.setPreferredMinMax(-1.5, 1.5, -1.5, 1.5, 0.0, 2.5);
6     mPanel3D.getCamera().adjust();
7     JFrame mFrame = new JFrame(``Virtual and Remote Robotics Laboratory'');
8     mFrame.getContentPane().setPreferredSize(new Dimension(600, 600));
9     mFrame.getContentPane().add(mPanel3D.getComponent(),BorderLayout.CENTER);
10    mFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11    mFrame.pack();
```

```
12    mFrame.setVisible(true);
13    mPanel3D.update();
14
15    // Visualizing the restrictions planes
16    ElementBox planeRestriction1 = new ElementBox();
17    planeRestriction1.setXYZ(0.2, 0.6, 1.2);
18    planeRestriction1.setSizeXYZ(0.8, 0.02, 1);
19    planeRestriction1.getStyle().setFillColor(Color.BLUE);
20    mPanel3D.addElement(planeRestriction1);
21
22    ElementBox planeRestriction2 = new ElementBox();
23    planeRestriction2.setXYZ(0.2, -0.25, 1.2);
24    planeRestriction2.setSizeXYZ(0.8, 0.02, 1);
25    planeRestriction2.getStyle().setFillColor(Color.BLUE);
26    mPanel3D.addElement(planeRestriction2);
27
28    // Defining a box
29    final ElementBox box = new ElementBox();
30    box.setXYZ(0.025, 0.47, 0.94);
31    box.setSizeXYZ(0.05, 0.05, 0.12);
32    box.getStyle().setFillColor(new Color(255, 0, 128));
```

Llegados a este punto, es importante destacar que, una vez establecida la conexión con el robot real y configurado el laboratorio virtual, todas las tareas que se establezcan a partir de este momento serán ejecutadas por ambos robots simultáneamente (ver Codigo 6.8). En este caso, se han establecido tres trayectorias diferentes: ir a la posición "home", ir donde está el objeto de la cinta para cogerlo y transladarlo a otra posición de la cinta. Si algún punto de estas trayectorias no verifica alguna de las restricciones establecidas, un panel con un mensaje de error aparecerá en la pantalla, y ambos robots volverán a la posición "home".

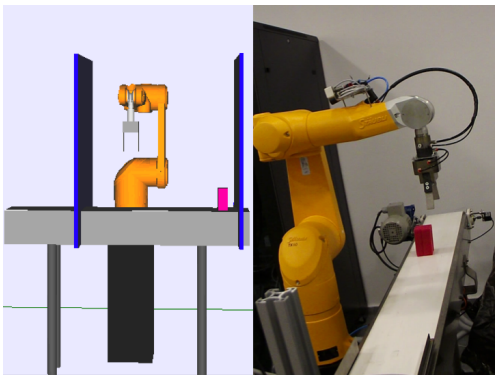
Con el objetivo de minimizar esfuerzo y evitar repetir las mismas sentencias de código para cada una de las trayectorias, se ha definido un método auxiliar para mover el robot a través de cualquier trayectoria establecida (ver líneas 45-51).

Código 6.8: Laboratorio Virtual y Remoto - Ejecución de las tareas

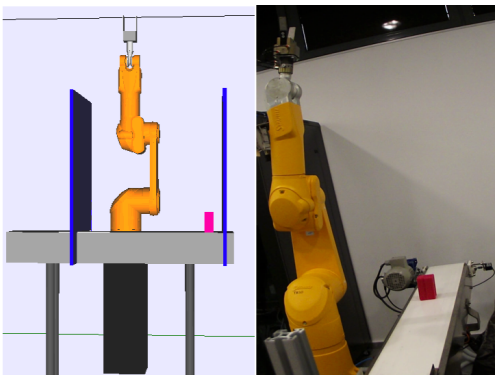
```
1   // STEP 5: Running the simulation as a thread
2  new Thread() {
3    public void run() {
4      double[] initialPoint = { 0, 0, 90, 0, 90, 0 };
5      double[] home = tx60l.getHome();
6      double[] p1 = { 50, 25, 90, 0, 64, 50 }; // pick up the box
7      double[] p2 = { -43.97, 7.67, 113.3, 0, 59.03, -43.97 }; // leave the box
8      double duration = 5.0; // seconds
9      double step = 0.1;
10     belt.attachObject(box); // Put the box on the belt
11     try {
12       // First task: Going to home position and open the tool
13       tx60l.setLinearTrajectory(initialPoint, home, duration);
14       followTrajectory(roboticsLab, tx60l, step, mPanel3D);
15       tx60l.openTool();
16       // Second task: Go towards the box
```

```
17          tx60l.setLinearTrajectory(home, p1, duration);
18          followTrajectory(roboticsLab, tx60l, step, mPanel3D);
19
20          // Third task: Pick up the box
21          belt.detachObject(box);
22          tx60l.attachObject(box);
23          tx60l.closeTool();
24          mPanel3D.update();
25
26          // Fourth task: Moving the box
27          tx60l.setLinearTrajectory(p1, p2, duration);
28          followTrajectory(roboticsLab, tx60l, step, mPanel3D);
29
30          // Final task: leave box
31          tx60l.openTool();
32          tx60l.detachObject(box);
33          belt.attachObject(box);
34          mPanel3D.update();
35
36      } catch (Exception exc) {
37          try { // Move quickly to home
38            tx60l.setLinearTrajectory(tx60l.getCurrentQ(), tx60l.getHome(), 2);
39            followTrajectory(roboticsLab, tx60l, step, mPanel3D);
40          } catch (Exception exc2) {exc.printStackTrace();}
41        }
42      }.start();
43  } // end of main()
44
45  static private void followTrajectory(RoboticsLab roboticsLab, AbstractRobot robot, double
        step, DrawingPanel3D panel3D) throws Exception {
46      robot.trajectoryHome();
47      while (!robot.trajectoryFinished(step)) {
48        roboticsLab.trajectoryStep(robot, step);
49        Thread.sleep((long) (1000 * step));
50        panel3D.update();
51      }
```

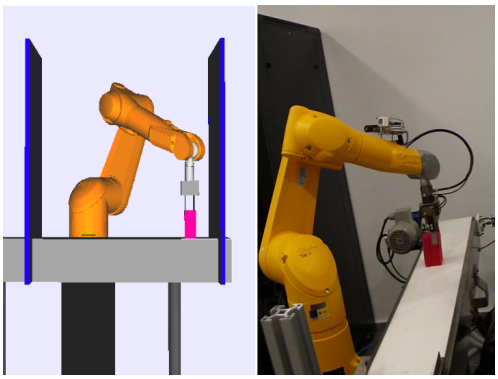
Para concluir con el ejemplo de este laboratorio virtual y remoto, en la Figura 6.3 se muestra el resultado de cada una de las fases del diseño llevado a cabo. En la Figura 6.3(a) se muestra el estado inicial de ambos robots antes de iniciar sus tareas, en la Figura 6.3(b) se observa cómo ambos robots (virtual y remoto) han ido a su posición home. Seguidamente, van a coger el objeto depositado en la cinta (ver Figura 6.3(a)) y lo mueven hacia otra posición. En el trascurso de esta última tarea (ver Figura 6.3(d)), se observa cómo una de las restricciones ha sido violada, los robots se han detenido antes de colisionar y, automáticamente, se ha obtenido un mensaje de error diciéndonos las coordenadas exactas del punto que no satisface una de las restricciones establecidas, ya que $Y = -0{,}206 \ngeq -0{,}2$. Además, en la parte del laboratorio virtual se puede ver fácilmente como el robot chocaría con la pared que representa dicha restricción.



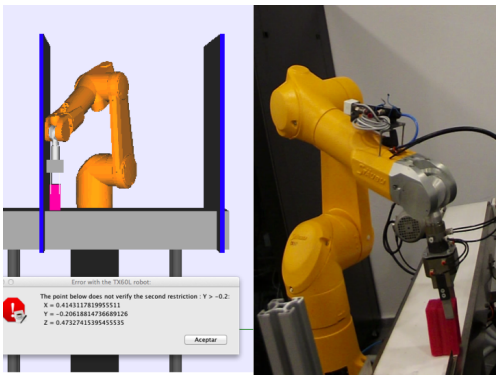
(a) Paso 1: Posición inicial



(b) Paso 2:Los robots se mueven a la posición "home"



(c) Paso 3: Los robots cogen el objeto de la cinta




(d) Paso 4: Los robots mueven dicho objeto, evitando una restricción

Figura 6.3: Laboratorio virtual y remoto en Java para un robot TX60L

## 6.2. Laboratorios robóticos virtuales y remotos en EJS

En esta sección se van a detallar distintos ejemplos de laboratorios robóticos virtuales y remotos, pero en este caso diseñados haciendo uso de la integración de la nueva librería en el software de simulación EJS.

De manera análoga a cómo ocurre en el caso de utilizar directamente la librería (ver Sección 6.1), los usuarios de EJS también tienen dos opciones a la hora de crear sus laboratorios. Por un lado, pueden diseñar laboratorios simples compuestos por uno o varios robots que están claramente separados y que no presentan interferencias entre ellos, tal y como se detalló en el ejemplo básico dado en la Sección 5.3. Por otro lado, pueden diseñar otros laboratorios más complejos en los que sus elementos deben operar en colaboración y proximidad. Para estos casos, el conjunto de elementos del modelo de EJS proporciona el elemento *RoboticsLab* (descrito en la Sección 5.1), diseñado principalmente para coordinar el movimiento de un conjunto de robots dentro de un mismo entorno.

Este elemento es añadido a la simulación siguiendo el mismo procedimiento que el empleado para el resto de elementos y que se detalló en la Subsección 5.1.1. Una vez incluido, el autor lo instruye para coordinar los diferentes elementos que integran el laboratorio mediante el uso de sus métodos, dados en la Tabla 4.1. Estos métodos internamente comprueban que los elementos verifican todas las restricciones establecidas y que no se producen colisiones entre los distintos elementos del laboratorio.

En las siguientes subsecciones se facilitan ejemplos de laboratorios robóticos complejos diseñados haciendo uso de este nuevo elemento *RoboticsLab*.

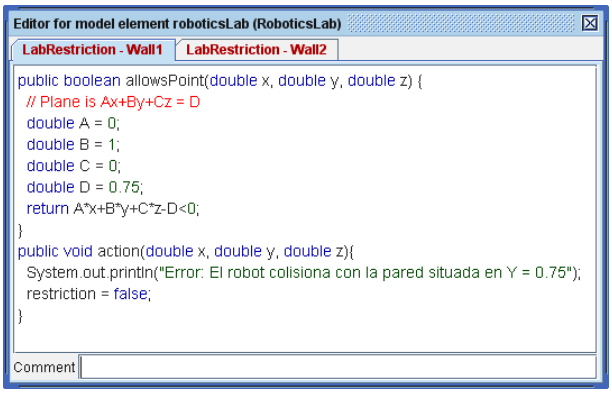
### 6.2.1. Ejemplo 1: Diseño de un laboratorio virtual en EJS compuesto por 2 robots

En este ejemplo se facilita un laboratorio robótico virtual compuesto por dos robots: un robot Scara y un robot TX60L. El objetivo fundamental de esta simulación es permitir al usuario comprobar de una forma sencilla las diferentes funcionalidades de los elementos robóticos integrados en EJS. Para ello, se han establecido diferentes restricciones generales en el laboratorio y en la vista del laboratorio se han incorporado diferentes elementos que permiten mover cada una de las articulaciones de los robots o establecer diferentes tipos de trayectorias. De esta manera, el usuario puede analizar diferentes situaciones en un mismo entorno, como por ejemplo, verificar si los robots se mueven sin colisionar entre ellos, si un robot autocolisiona con él mismo o con el otro robot, si alguno de los robots no satisface alguna de las restricciones, etc.

A continuación, siguiendo el procedimiento dado en la Sección 5.2 se describe cómo se ha diseñado este laboratorio. En este caso concreto, la configuración del panel de la *Vista* se va a detallar en el último paso puesto que es en el que más tiempo se ha invertido.

En primer lugar, se añaden a la *"List of Elements"* del Modelo todos los elementos robóticos que componen este laboratorio: un elemento RoboticsLab, un robot TX60L y un robot Scara, siguiendo el mismo procedimiento que se describió en la Subsección 5.1.1. Una vez incluidos en esta lista, se configura el elemento *RoboticsLab* y se establecen dos restricciones de tipo *Plane*, una en $Y = 0{,}75$ y otra en $X = -0{,}75$, siguiendo el procedimiento dado en la Subsección 5.1.3, y cuya definición se muestra en la Figura 6.4.

En segundo lugar, se definen todas las variables necesarias para el diseño del laboratorio. En este caso, las variables han sido definidas en tres páginas diferentes: en la página *Motion* se han definido las variables $q_i$ para las seis articulaciones del robot TX60L y las $j_i$ para las cuatro articulaciones del Scara, tal y como se puede observar en la Figura 6.5(a). De forma análoga, en la página *PathPlanning* se han definido las variables necesarias para establecer diferentes tipos de trayectorias, es decir, las variables correspondientes a las coordenadas de las articulaciones de los puntos intermedios y finales de estas trayectorias, así como la duración y el delta de tiempo empleado en las mismas. Finalmente, en la página *Interface* se han incluido algunas variables auxiliares

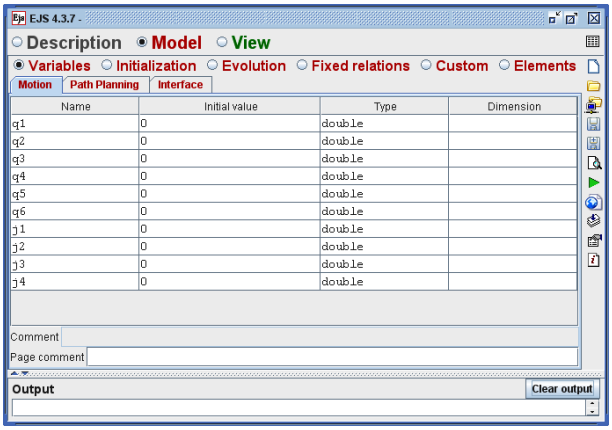



(a) Restricción en Y = 0.75 (b) Restricción en X = -0.75

Figura 6.4: Laboratorio virtual en EJS - Definición de restricciones del laboratorio

empleadas en la configuración de los paneles de la vista.

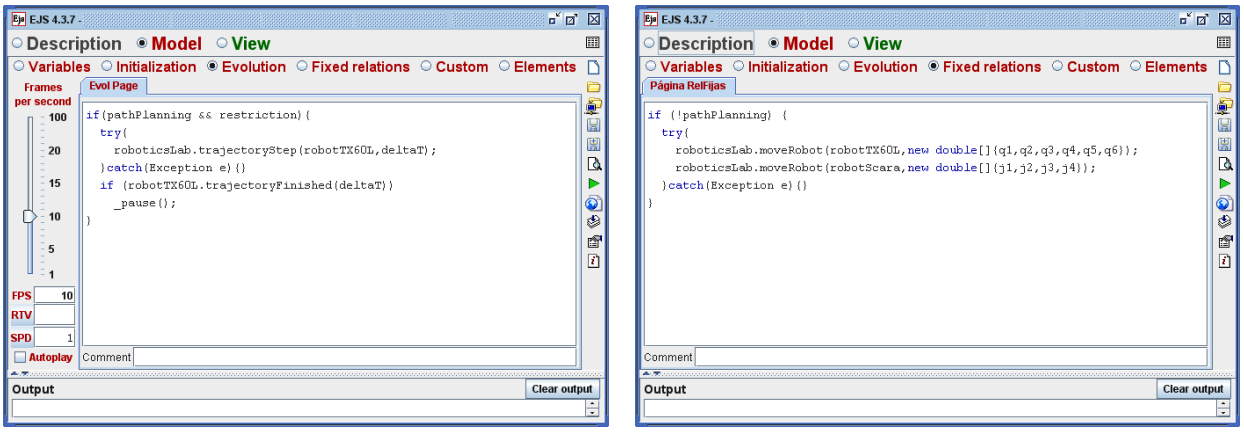
En tercer lugar, en la página de *Model* - *Initialization* se añaden los robots al elemento *roboticsLab* tal y como se muestra en la Figura 6.5(b).




(a) Variables del modelo (b) Inicialización del modelo

Figura 6.5: Laboratorio virtual en EJS - Definición de las variables e inicialización del modelo

Para finalizar el diseño del Modelo sólo nos queda configurar los paneles *Evolution* y *Fixed relations* dados en la Figura 6.6. En el panel *Evolution* se han escrito las sentencias de código necesarias para que el robot TX60L se mueva a través de cualquier trajectoria establecida en la ejecución de la simulación siempre y cuando éste verifique las restricciones del laboratorio y no colisione con el otro robot Scara (Figura 6.6(a)). En el panel *Fixed relations* se han establecido las relaciones necesarias para poder mover cualquiera de los dos robots del laboratorio cuando los valores de sus articulaciones cambien por la interacción del usuario durante la ejecución de la simulación (Figura 6.6(b)).

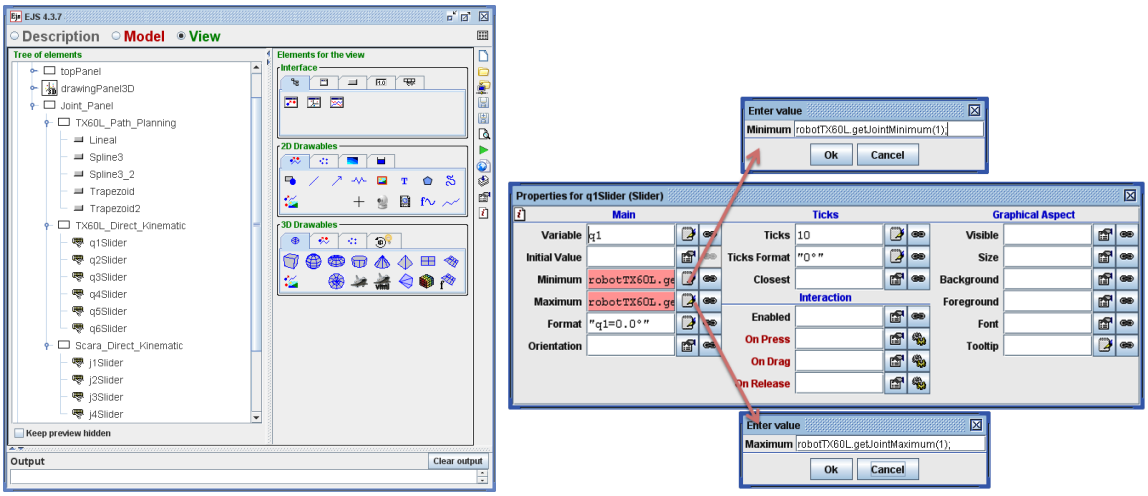



(a) Evolución del modelo (b) Relaciones fijas del modelo

Figura 6.6: Laboratorio virtual en EJS - Definición de la evolución del modelo y relaciones fijas

Concluida la configuración del Modelo estamos en condiciones de diseñar la vista de este laboratorio virtual. A continuación se describen con detalle los diferentes elementos que han sido integrados, no sólo para la visualización de los robots y el entorno tridimensional, sino también para el control de las articulaciones, generación de trayectorias y otras utilidades.

En la Figura 6.7(a) podemos observar tres paneles diferentes: uno para la generación de trayectorias, en el que se han definido diferentes elementos de tipo *Button* (uno para cada tipo de trayectoria que el usuario puede seleccionar durante la ejecución de la simulación), y otros dos paneles en los que se han incluido diferentes elementos de tipo *Sliders* (barras deslizadoras) para el control independiente de cada una de las articulaciones de los dos robots del laboratorio.



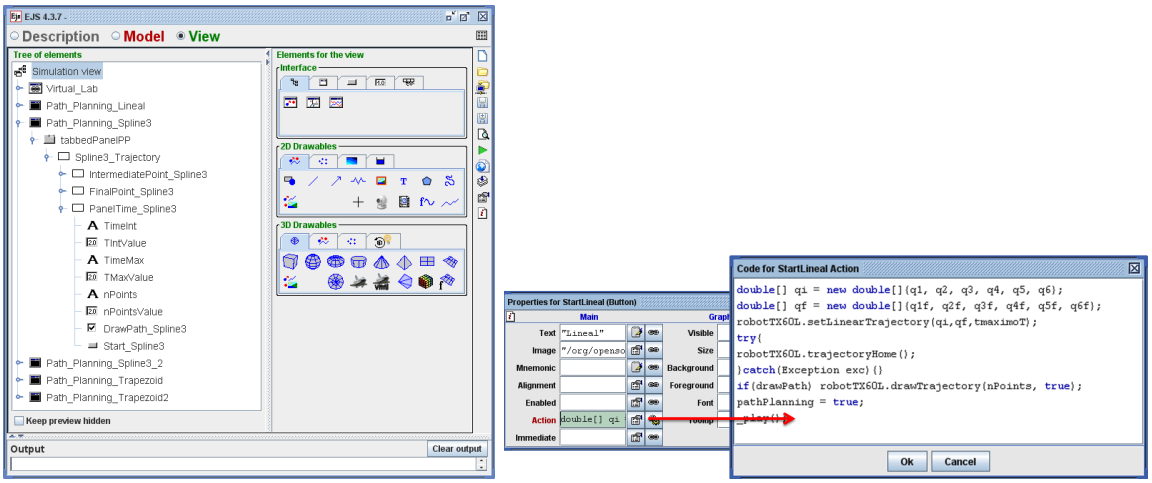

(a) Paneles para el posicionamiento del robot (b) Configuración de los valores de las articulaciones

Figura 6.7: Laboratorio virtual en EJS - Configuración de la vista del laboratorio I

Todos estos elementos de tipo *Sliders* han sido configurados de manera análoga a la configuración mostrada en la Figura 6.7(b) para el caso del elemento *q1Slider*. En dicha configuración se puede observar el uso de los métodos correspondientes para obtener los valores extremos de las articulaciones de un robot (implementados en la API del elemento) y cómo se puede relacionar el valor de este elemento de la vista con una variable del modelo previamente definida.

Por otro lado, en la Figura 6.8(a) se observa la definición de cinco ventanas emergentes, una para cada una de las trayectorias implementadas, que se abrirán automáticamente cuando el usuario seleccione el tipo de trayectoria que desea definir haciendo click en su botón correspondiente, implementado en el panel superior "TX60L_Path_Planning".

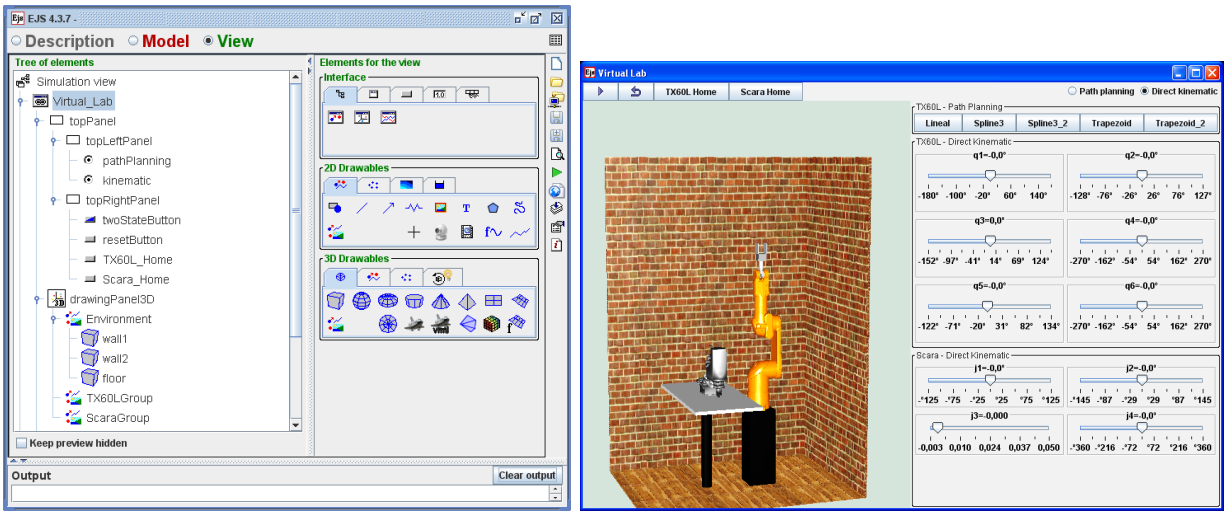
La configuración de estas ventanas tiene una estructura similar para todas ellas y sólo difieren en los parámetros necesarios para establecer cada una de las trayectorias. En la misma Figura 6.8(a) se pueden observar los parámetros necesarios para establecer una trayectoria Spline con un único punto intermedio, mientras que en la Figura 6.8(b) se muestra cómo se definiría este tipo de trayectoria para el robot TX60L.




(a) Paneles para planificación de trayectorias (b) Configuración de la trayectoria "Spline"

Figura 6.8: Laboratorio virtual en EJS - Configuración de la vista del laboratorio II

En la Figura 6.9(a) se facilita el resto de elementos necesarios para completar la vista del laboratorio. Por un lado, en el panel *topPanel* se incluyen diferentes acciones para la interacción con el laboratorio: un botón "Play/Pause" para parar y ejecutar la simulación, un botón "Reset" para resetear el estado actual del laboratorio, dos botones "TX60L_Home" y "Scara_Home" para mover los robots a sus posiciones por defecto y finalmente otros dos botones, "pathPlanning" y "kinematic", para seleccionar el modo en el que el usuario quiere trabajar, bien si desea establecer trayectorias o si por el contrario quiere mover él mismo las articulaciones de cada uno de los robots. Por otro lado, en *"drawingPanel3D"* se incluyen los elementos necesarios para la visualización 3D de los robots y del entorno, de forma análoga a como se hizo en el ejemplo dado en la Sección 5.3.




(a) Paneles para configuración del laboratorio (b) Visualización del laboratorio virtual

Figura 6.9: Laboratorio virtual en EJS - Configuración de la vista del laboratorio III

Finalmente, sólo nos queda ejecutar la simulación para observar el resultado de este laboratorio virtual (ver Figura 6.9(b)) e interaccionar con ella para estudiar con detalle cada una de las propiedades que ofrece al usuario. A continuación se analizan diferentes casos que ilustran cada una de las características implementadas:

- Caso A - Estudio de las restricciones del laboratorio:

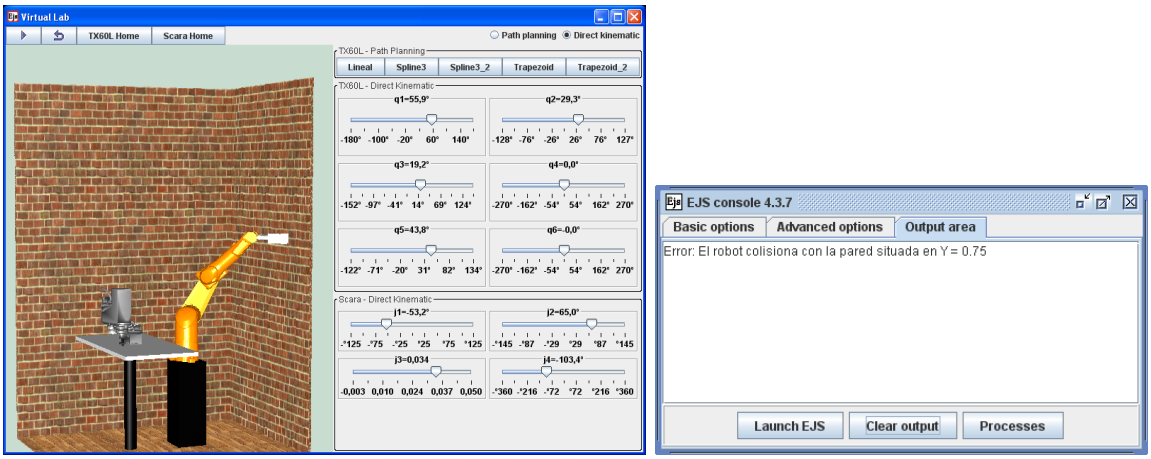
  Seleccionando la opción "Direct Kinematics" se pueden mover las diferentes articulaciones de cada uno de los robots dentro de sus límites establecidos tal y como se observa en la Figura 6.10(a). Si alguna de estas configuraciones choca con alguna de las paredes del laboratorio definidas como restricciones generales del elemento *RoboticsLab* obtendremos automáticamente un mensaje de error en la consola tal y como se muestra en la Figura 6.10(b).

- Caso B - Análisis de la detección de autocolisiones:

  De forma análoga al caso anterior se pueden obtener otras configuraciones para las articulaciones de los robots (ver Figura 6.11(a)) con el fin de que permitan estudiar si para alguna de estas configuraciones existe una posible autocolisión entre dos de las articulaciones del robot. En caso de que ocurra se obtendría también un mensaje de error tal y como se muestra en la Figura 6.11(b).

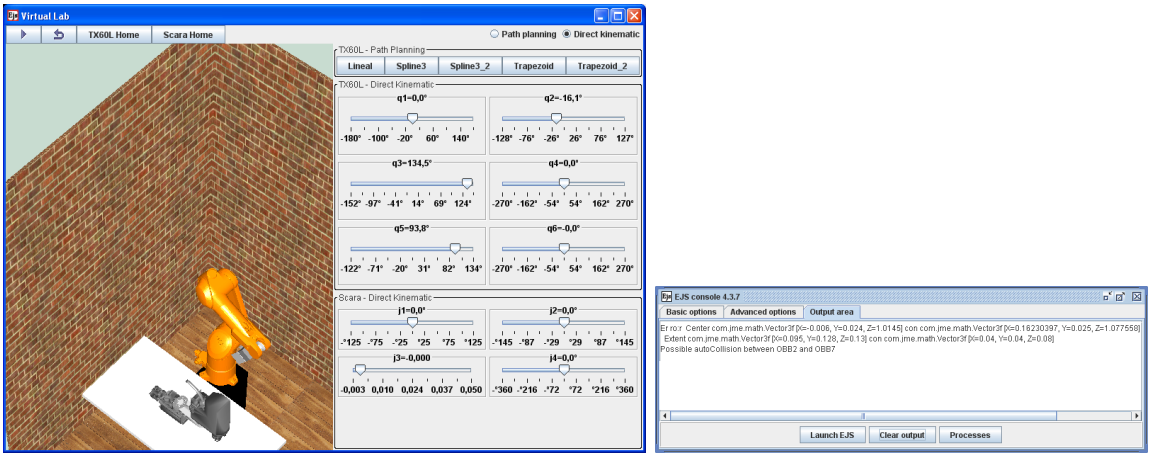
- Caso C - Definición de trayectorias y detección de colisiones:

  En este último caso se ha seleccionado la configuración "Paht Planning" para poder definir por ejemplo una trayectoria de tipo Trapezoidal. Haciendo click en el botón "Trapezoid" automáticamente aparecerá una ventana emergente para llevar a cabo su configuración tal




(a) Comprobación de restricciones del laboratorio (b) Error - Violación de restricción

Figura 6.10: Laboratorio virtual en EJS - Simulación del laboratorio I

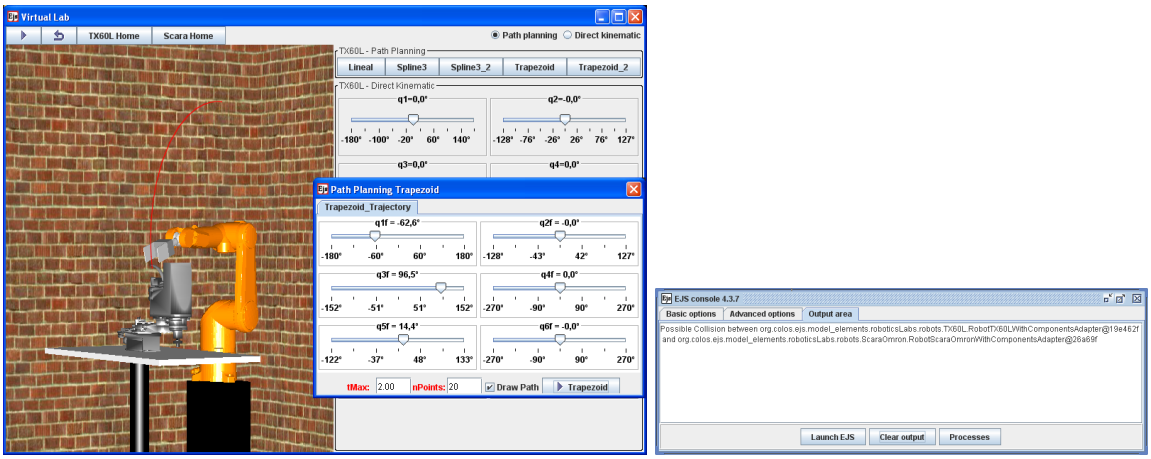



(a) Comprobación de detección de autocolisiones (b) Error - Posible autocolisión

Figura 6.11: Laboratorio virtual en EJS - Simulación del laboratorio II

y como se muestra en la Figura 6.12(a). En esta ventana se deben facilitar el punto final de la trayectoria (el inicial es el estado actual de las variables $q_i$), el tiempo máximo para completar dicha trayectoria y el número de puntos que deseamos dibujar, en el caso en el que se seleccione la opción "Draw Path".

Una vez facilitados todos los parámetros, sólo faltaría hacer click en el botón"Play Trapezoid" que aparece en esa misma ventana y automáticamente se observa cómo se dibuja esta nueva trayectoria en el laboratorio mientras se mueve el robot a través de ella. En este caso, la trayectoria no se puede completar, ya que se detecta una posible colisión entre los dos robots. Entonces, el movimiento del robot es detenido automáticamente y se obtiene un mensaje de error por consola (ver Figura 6.12(b)).




(a) Generación y visualización de trayectorias (b) Error - Detección de colisión

Figura 6.12: Laboratorio virtual en EJS - Simulación del laboratorio III

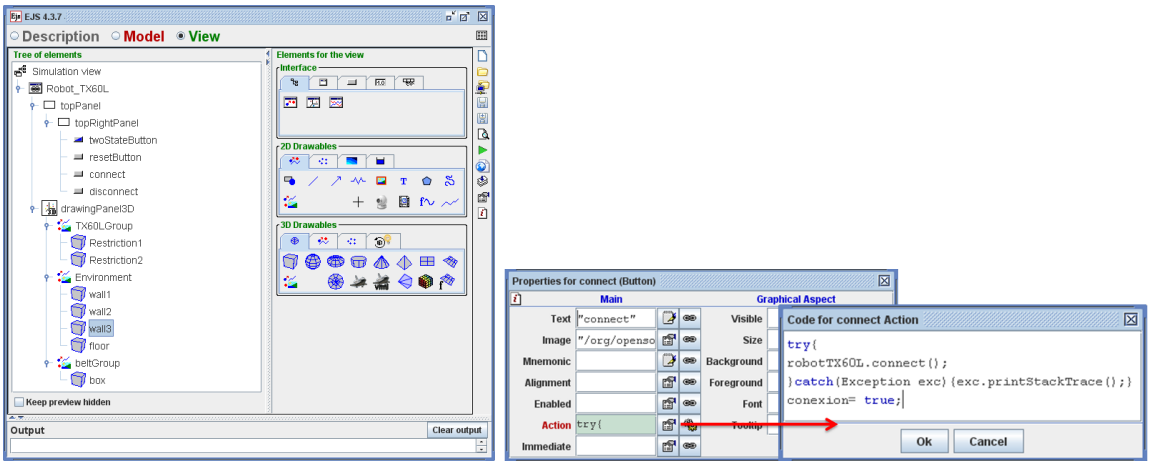
### 6.2.2. Ejemplo 2: Diseño de un laboratorio virtual y remoto en EJS

En este segundo ejemplo se va a desarrollar el mismo laboratorio virtual y remoto que el dado en la Subsección 6.1.2, pero en este caso diseñado en el entorno EJS.

De forma similar a como se ha hecho en los otros ejemplos desarrollados en EJS y siguiendo el procedimiento descrito en la Sección 5.2, se comienza con la configuración del panel de la vista. En el caso de este laboratorio se trata de una configuración muy sencilla, tal y como se puede observar en la Figura 6.13(a). Por un lado, se ha integrado un panel superior compuesto por cuatro botones para definir las diferentes acciones que el usuario puede llevar a cabo durante la ejecución de la simulación: play/pause, reset, establecer conexión con el robot real e interrumpir dicha conexión. A modo de guía en la Figura 6.13(b) se facilita la configuración del botón connect (los otros botones se configuran de manera similar). Por otro lado, el resto del panel de la vista contiene los grupos 3D necesarios para visualizar todos los elementos que integran el laboratorio así como el entorno del mismo.

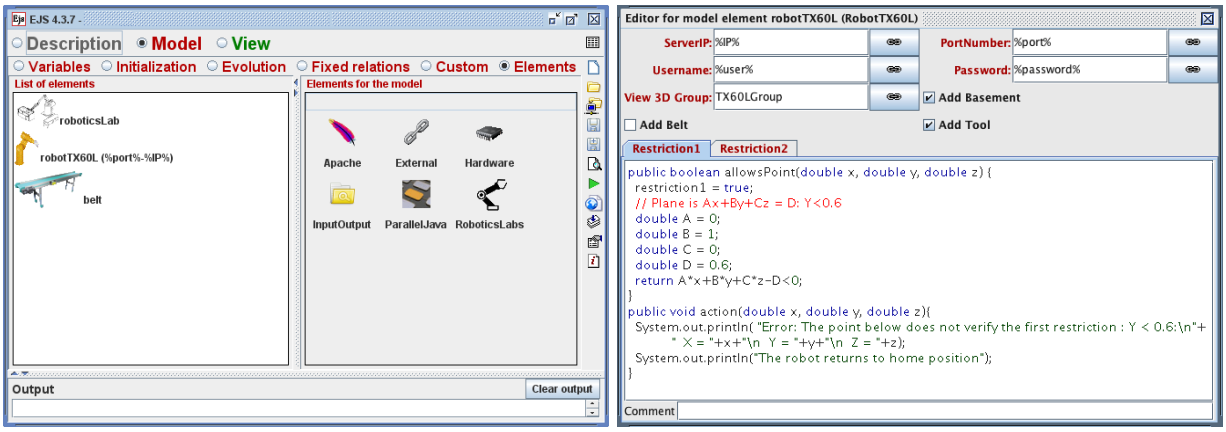
En segundo lugar, se define la lista de elementos que integran el laboratorio: un elemento roboticsLab, un robot TX60L y una cinta transportadora (ver Figura 6.14(a)) y se configura el robot TX60L tal y como se muestra en la Figura 6.14(b). En dicha configuración se facilitan los parámetros necesarios para establecer la conexión con el robot real y se definen dos restricciones de tipo “Plane”, una para $Y < 0.6$ y otra para $Y > -0.2$.

A continuación, se definen las variables del modelo (ver Figura 6.15(a)) y se implementa la inicialización de este laboratorio como se muestra en la Figura 6.15(b). En esta inicialización se añade el robot y la cinta transportadora al elemento “roboticsLab”, se inicializan algunas variables definidas previamente en el panel *Variables*, se establece la conexión con el robot real, se mueven ambos robots a la posición inicial dada y se define una trayectoria lineal.




(a) Lista de elementos de la vista (b) Configuración del botón “connect”

Figura 6.13: Laboratorio virtual y remoto en EJS - Configuración de la vista

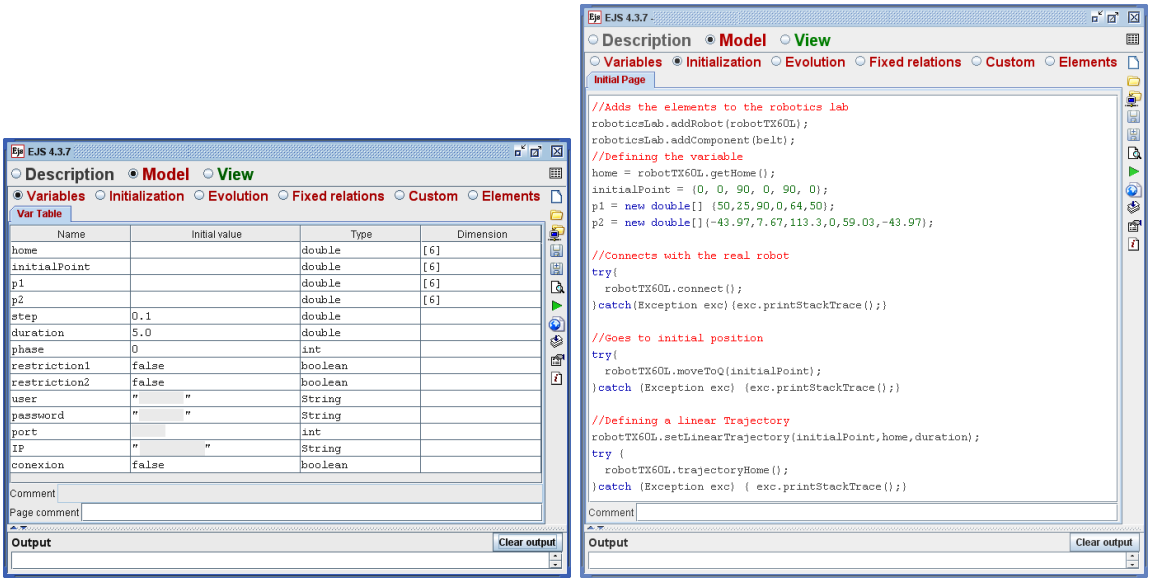



(a) Lista de elementos (b) Configuración del robot TX60L

Figura 6.14: Laboratorio virtual y remoto en EJS - Configuración de los elementos

Una vez que se ha establecido la conexión, cualquier operación afectará automáticamente tanto al robot virtual como al real. En este caso, se han definido tres trayectorias diferentes: ir a la posición “home” del robot, recoger el objeto depositado sobre la cinta y trasladarlo a otra posición (Figura 6.16). Si algún punto de las trayectoria no satisface alguna de las restricciones dadas, un mensaje de error aparecerá en la consola y el robot volverá a la posición home.

Finalmente, en la Figura 6.17 se muestran las diferentes fases de la ejecución de este laboratorio virtual y remoto en EJS. Se puede observar que los resultados obtenidos son idénticos a los del ejemplo desarrollado directamente en Java (ver Figura 6.3) dado en la Subsección 6.1.2.




(a) Variables del modelo (b) Inicialización del modelo

Figura 6.15: Laboratorio virtual y remoto en EJS - Variables e Inicialización del modelo

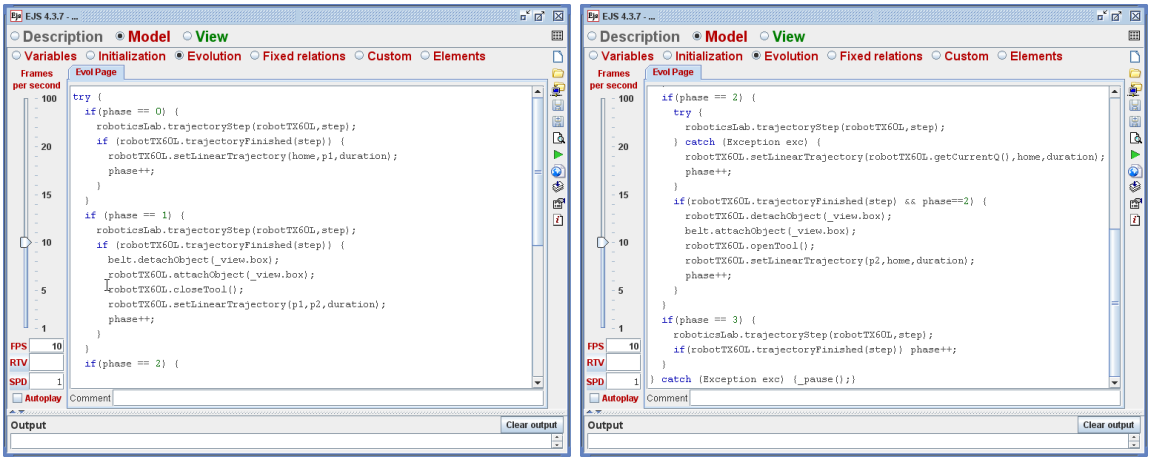
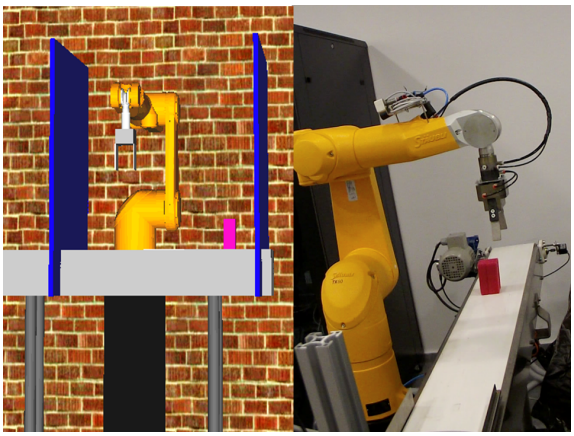
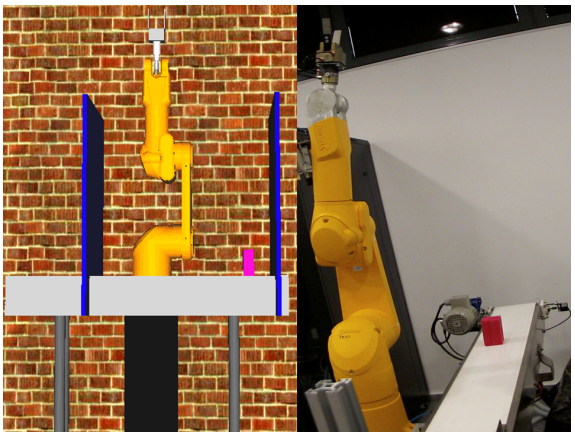


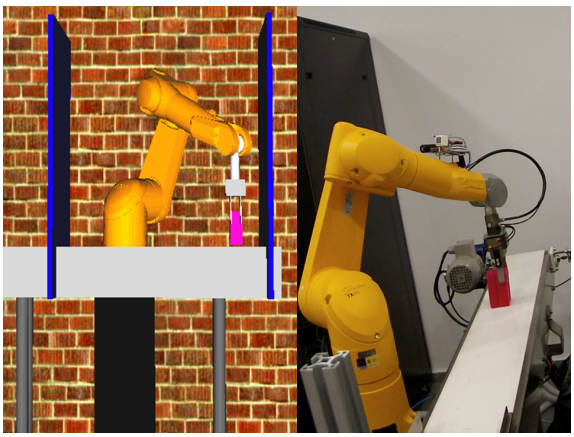

Figura 6.16: Laboratorio virtual y remoto en EJS - Evolución del modelo



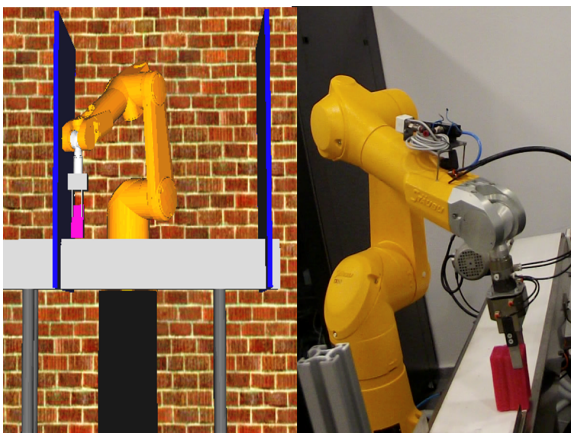
(a) Paso 1: Posición inicial



(b) Paso 2: Los robots se mueven a la posición "home"



(c) Paso 3: Los robots cogen el objeto de la cinta



(d) Paso 4: Los robots mueven dicho objeto

Figura 6.17: Laboratorio virtual y remoto en EJS para un robot TX60L - Simulación

# Parte III

# Conclusiones

# Capítulo 7

# Conclusiones y Trabajos futuros

En este capítulo se presentan las conclusiones del trabajo desarrollado, las aportaciones científicas conseguidas, las posibles líneas de trabajos futuros y, finalmente, las publicaciones obtenidas.

## 7.1. Conclusiones

En esta tesis doctoral se aporta un nuevo framework en Java para la coordinación de robots manipuladores en entornos estáticos y dinámicos mediante el diseño de laboratorios virtuales y remotos. La herramienta es de código abierto, compatible con todos los sistemas operativos que soporten Java, y muy sencilla de utilizar, es decir, presenta una curva de aprendizaje suave.

Dicho entorno ha sido desarrollado para ser aplicado en diversos ámbitos:

- En educación, al ser una herramienta sencilla permite a los estudiantes analizar y estudiar conceptos básicos de robótica industrial.
- En investigación, ya que facilita a los investigadores la representación de distintas situaciones complejas para su posterior análisis y estudio.
- En ingeniería, pues resulta una herramienta útil para el prototipado de plantas industriales.

En primer lugar, y como principal contribución, se puede destacar la modelización, conceptualización y diseño de un nuevo HAL común para todos los robots manipuladores. Este HAL facilita una API universal que contiene todas las operaciones básicas de este tipo de robots: posicionamiento, resolución de sus cinemáticas directa e inversa, planificación de trayectorias, definición de

restricciones físicas y de movimiento, detección de posibles autocolisiones y detección de colisiones con el propio entorno que lo rodea. Además, este HAL incluye la conceptualización de otros componentes robóticos que pueden resultar básicos en el diseño de los laboratorios robóticos (sensores, actuadores, cintas transportadoras, ...).

En segundo lugar, incluye la visualización 3D de todos sus elementos, lo cual permite crear laboratorios virtuales con un alto grado de realismo e interactividad.

En tercer lugar, se han implementado diferentes protocolos de comunicación en la librería para facilitar al usuario el diseño de laboratorios remotos. Estas implementaciones permiten establecer la conexión remota con el robot real sin necesidad de que el usuario conozca los detalles técnicos del protocolo requerido. Al mismo tiempo, facilita la comunicación con el robot real traduciendo las sentencias establecidas por el usuario al lenguaje de programación propio del robot.

En cuarto lugar, dicha API ha sido implementada para dos robots manipuladores particulares de diferentes GDL: el robot TX60L de Stäubli y el robot Scara de Omron, y para un componente robótico: una cinta transportadora. Con estos tres elementos la librería permite al usuario diseñar diferentes laboratorios robóticos tanto virtuales como remotos de distinta complejidad. Además, otra de las ventajes que presenta esta librería es que esta lista de elementos puede ser fácilmente ampliada con otros robots o componentes robóticos, ya que dichas implementaciones se han realizado de forma sencilla y fácilmente reutilizable.

En quinto lugar, como otra de las principales aportaciones de la tesis, los elementos de esta librería han sido embebidos en el software de simulación EJS como elementos del modelo. Mediante esta integración, tanto los programadores con alto nivel de Java como aquellos que no lo posean están en disposición de diseñar laboratorios virtuales y remotos sin necesidad de escribir demasiado código y sin la necesidad de consultar ningún manual de robótica para estudiar los detalles de los robots (tales como sus particularidades geométricas, el lenguaje propio de programación del robot, protocolos de comunicación soportados,...). El entorno EJS, enriquecido con estos nuevos elementos del modelo, proporciona una herramienta poderosa en el campo de la educación en ingeniería porque permite enseñar conceptos robóticos de una manera fácil y sencilla.

Independientemente de si los laboratorios robóticos son diseñados haciendo uso directamente de la librería o apoyándose en el software EJS, cabe destacar que este nuevo framework facilita al usuario coordinar el movimiento de todos los elementos del laboratorio, evitando además las posibles colisiones que puedan existir entre ellos tanto en entornos estáticos como dinámicos.

Finalmente, a lo largo de esta tesis doctoral se han proporcionado ejemplos de diferentes laboratorios robóticos virtuales y remotos de distinta complejidad con el fin de mostrar todas las capacidades que posee este nuevo framework.

## 7.2. Trabajos Futuros

Del trabajo realizado en esta tesis doctoral y de sus resultados obtenidos se pueden considerar las siguientes líneas de trabajo futuro:

- *Extensión de los elementos de la librería:*

  La nueva librería presentada en esta tesis proporcionaba a los usuarios una serie de elementos previamente integrados. En concreto, dos robots manipuladores de distintos GDL y una cinta transportadora.

  Una primera línea de trabajo que se podría plantear como continuación de esta tesis sería la extensión de esta lista de elementos con la integración de nuevos robots manipuladores y otros componentes robóticos, tales como sensores, actuadores, ...

- *Simulación con realidad aumentada:*

  Una segunda línea de trabajo posible sería el desarrollo de las simulaciones robóticas con realidad aumentada. En este momento, la nueva librería facilita el diseño de laboratorios robóticos virtuales y/o remotos en los que el usuario es capaz de controlar los robots simulados, los reales o ambos al mismo tiempo.

  Llegados a este punto, un siguiente paso sería poder diseñar simulaciones robóticas con realidad aumentada. En esta situación se podría tener, por ejemplo, un robot simulado y otro real que interaccionan entre ellos y llevan a cabo tareas en coordinación. Otro ejemplo de aplicación sería para poder comparar el comportamiento de un robot virtual con su equivalente real.

- *Modelos de comunicación entre robots:*

  Dentro del área de la coordinación de los robots, otro trabajo futuro destacable sería la implementación de diferentes modelos de comunicación entre robots, como por ejemplo, la implementación de redes de Petri. Esta implementación permitiría al usuario definir redes de Petri para llevar a cabo la coordinación de todos los robots de su laboratorio de una manera cómoda y sencilla.

- *Migración de la librería desarrollada en Java a JavaScript:*

  Finalmente, otra línea de trabajo futuro sería la migración de la nueva librería desarrollada en Java a JavaScript.

  Una de las herramientas de software que todos los dispositivos de computación, incluidas las plataformas móviles, seguramente tienen instalada es un navegador World Wide Web (WWW). Por lo tanto, distribuir y ejecutar programas asumiendo el único apoyo de un navegador web es una apuesta segura.

Por supuesto que para la administración de los laboratorios, los usuarios pueden instalarse programas o platafomas concretos. Pero este requisito aumenta la probabilidad de problemas de instalación, necesidad de pagar licencias o plataformas no soportadas. Por esta razón, gradualmente se está adoptando el navegador web como una distribución preferida y la plataforma de tiempo de ejecución.

En particular, Java, quizás el primer lenguaje de programación que se ha integrado en los navegadores web y que se ha utilizado como plataforma de ejecución para todas las simulaciones durante muchos años, ha perdido recientemente el favor del gran público (debido a tantos problemas de seguridad) y Google ha decidido interrumpir el soporte para applets Java en su navegador Chrome [136].

Por este motivo, la migración de la nueva librería a JavaScript resolvería estos problemas de los applets de Java en los servidores Web.

JavaScript es el lenguaje interpretado más utilizado, principalmente en la construcción de páginas Web, con una sintaxis muy semejante a Java y a C. Pero, al contrario que Java, no se trata de un lenguaje orientado a objetos propiamente dicho, sino que éste está basado en prototipos, ya que las nuevas clases se generan clonando las clases base (prototipos) y extendiendo su funcionalidad [137].

## 7.3. Publicaciones

Los resultados y aportaciones obtenidas a lo largo de esta tesis doctoral se han presentado y expuesto en los siguientes congresos internacionales:

- *"High-Level Control of Robots in EJS"*. Almudena Ruiz, Francisco Esquembre, Humberto Martínez. *18th International Conference on Multimedia in Physics Teaching and Learning Workshop (MPTL'18)*. Madrid (Spain). September 2013.

- *"High-Level Control of Robots in Java"*. Almudena Ruiz, Francisco Esquembre, Humberto Martínez. *Med-Souk: I International conference for young researchers in the Mediterranean.* Murcia (Spain). October 2013. Publicado en: *WAF 2014, Book of proceedings* . ISBN: 978-84-9773-681-7, pp.188-194.

- *"Creation of Robotics Simulations in a simple way with Easy Java Simulations"*. Almudena Ruiz, Humberto Martínez, Francisco Esquembre. *XV Workshop of Physical Agents (WAF 2014)*. León (Spain). June 2014.

Además de las contribuciones a congresos internacionales, también se han enviado dos artículos a diferentes revistas de carácter internacional, cuya publicación no ha sido hasta el momento aceptada.

# Parte IV

# Apéndices

# Apéndice A

# Implementación de los elementos de la nueva librería

En este Apéndice se facilitan los detalles concretos de las implementaciones de los elementos que conforman la librería. El objetivo de facilitar estos detalles es que puedan servir como base en el momento en que un usario decida extender dicha librería con nuevos robots o componentes robóticos.

Tal y como se ha detallado a lo largo de esta tesis, para incorporar nuevos elementos es necesario extender la clase *AbstractRobot* o *AbstractComponents* según sea un nuevo robot o un nuevo componente. A continuación, en la Sección A.1 y Sección A.2 se muestran los detalles de las implementaciones de los robots TX60L y Scara, mientras que en la Sección A.3 se encuentran los detalles de la integración de una cinta transportadora.

## A.1. Detalles implementación robot TX60L

La clase *AbstractRobot* detallada en la Sección 4.3 es una clase abstracta que declara una serie de métodos y propiedades comunes a todos los robots manipuladores (independientemente de sus características propias) que el usuario debe implementar en el caso de querer incorporar nuevos robots a la librería. Dichos métodos fueron resumidos en la Figura 4.9.

En el resto de sección se adjuntan los detalles concretos de la implementación de estos métodos en el caso del robot TX60L de Stäubli.

### A.1.1. Información física del robot TX60L

En primer lugar, a la hora de definir un nuevo robot es necesario proporcionar todos los datos correspondientes a la información física del robot: GDL, dimensiones de sus ejes, valores límites de las articulaciones, máximos valores permitidos para la velocidad y aceleración, ... Estas variables serán básicas para poder implementar los diferentes métodos de las clase AbstractRobot.

En el Código A.1 se muestra la definición de los parámetros del robot TX60L así como la implementación de los métodos correspondientes que proporcionan la información física del robot: GDL, la longitud de las coordenadas cartesianas, y la posición "home" que tiene el robot por defecto.

Código A.1: Implementación de la información física del robot TX60L

```
1  ...
2  // Parámetros del robot TX60L
3   static final private  int sDOF = 6;
4   static final private  int sCARTESIAN_ARRAYS_LENGTH = 6;
5   static final private  int [] sDHq = new int[] { 0, 1, 2, 3, 4, 5};
6   static final private final double[] sHOME = { 0, 0, 0, 0, 0, 0 };
7   static final private double[] sJOINT_MINIMA = {-180, -127.5, -142.5, -270, -122.5, -270};
8   static final private double[] sJOINT_MAXIMA = {+180, +127.5, +142.5, +270, 133.5, +270};
9   static final private double[] sSPEED_LIMITS = {435, 385, 500, 995, 1065, 1445};
10  static final private double[] sACCEL_LIMITS = {1373, 1373, 3096, 2802, 1707, 8167};
11  static final protected double sLengthOfBase = 0.375;
12  static final protected double sWidthOfShoulder = 0.217;
13  static final protected double sWidthOfElbow = 0.197;
14  static final protected double sLengthOfForearm = 0.450;
15  static final protected double sLengthOfWrist = 0.070;
16  static final protected double sLengthOfArm = 0.400;
17  static final protected double a3 = 0.0;
18  protected double sLengthOfTool = 0.0;
19
20  public int getDOF() { return sDOF; }
21
22  public int getCartesianArraysLength() { return sCARTESIAN_ARRAYS_LENGTH; }
23
24  public double[] getHome() {
25    double[] home = new double[sDOF];
26    System.arraycopy(sHOME,0,home,0,sDOF);
27    return home;
28  }
29  ...
```

### A.1.2. Posicionamiento del robot TX60L

Independientemente de la resolución de la cinemática directa e indirecta, la clase *AbstractRobot* es capaz de posicionar el robot dada una posición tanto en coordenadas articulares como en cartesianas (ver métodos de la Tabla 4.2). Sin embargo, si el usuario quiere posicionar el robot real

una vez establecida la conexión remota o manejar la herramienta del robot necesita implementar los métodos *moveRealRobot()*, *openTool()* y *closeTool()* dados en el Código A.2.

En el método *moveRealRobot()* se llevan a cabo las sentencias necesarias para comunicarle al robot real la nueva posición donde queremos que se mueva (ver Líneas 3-5), en *openTool()* se abriría tanto la herramienta del robot real (Línea 9) como del virtual (Líneas 10-13) y en *closeTool()* se cerrarían ambas (Líneas 17-21).

Código A.2: Implementación del posicionamiento del robot TX60L

```
1  ...
2  protected void moveRealRobot() throws Exception {
3    String data = "2\n" + mCurrentQ[0] + "\n" + mCurrentQ[1] + "\n" + mCurrentQ[2] + "\n"
4      + mCurrentQ[3] + "\n" + mCurrentQ[4] + "\n" + mCurrentQ[5] + "\n";
5    writeSocket(data);
6  }
7  public void openTool() throws Exception {
8    mOpenTool=true;
9    if (mIsConnected) writeSocket("6\n");
10   if (mHasViewGroup) {
11     gripper1.setXYZ(0,-0.04, 0.14);
12     gripper2.setXYZ(0, 0.04, 0.14);
13   }
14 }
15 public void closeTool() throws Exception {
16   mOpenTool=false;
17   if (mIsConnected) writeSocket("7\n");
18   if (mHasViewGroup) {
19     gripper1.setXYZ(0,-0.02, 0.14);
20     gripper2.setXYZ(0, 0.02, 0.14);
21   }
22 }
23 ...
```

### A.1.3. Cinemática Directa del robot TX60L

La implementación de la cinemática directa se ha llevado a cabo mediante el algoritmo de DH, basado en una multiplicación secuencial de transformaciones homogéneas que describen la relación entre los valores de las articulaciones y la localización espacial del extremo del robot (ver Subsección 3.1.1.1). Este algoritmo ha sido implementado sobre la base de la clase *AbstractRobot*, de forma que los usuarios pueden completar su definición fácilmente proporcionando los parámetros de DH del robot, dados en la expresión 4.1, e implementando el método *calculateDH(double[] q)* tal y como se muestra en el Código A.3.

Cabe destacar que la clase *Matrix* que se utiliza en la definición de estos métodos es una clase auxiliar del paquete *utils.Math.Matrix* que se ha definido para representar el elemento matemático de una matriz y sus operaciones elementales, básicas en el desarrollo de este algoritmo. Por otro lado, el método *FKinematics(double q, int number)*, utilizado para el cálculo auxiliar del algoritmo

de DH, es común para todos los robots de la librería ya que ha sido implementado en la clase *AbstractRobot* y no depende del robot en concreto.

Código A.3: Implementación de la cinemática directa del robot TX60L

```
1  ...
2  // Constantes comunes auxiliares para cálculos
3  static protected final double sTO_RADIANS = Math.PI / 180.0;
4  static protected final double PI2 = Math.PI / 2.0;
5  //Variables auxiliares necesarias para cálculo de cinemáticas
6  protected double Px, Py, Pz;
7  protected double[] O = new double[3];
8  private double[] A = new double[3];
9  static private final int [] sDHq = new int[] {0,1,2,3,4,5};
10
11 protected int getDHQ(int index) { return sDHq[index]; }
12
13 public double[] getDHParameters() {
14   return new double[] { 0, -AbstractRobot.PI2, AbstractRobot.PI2, 0 , 0, 0,
15     sLengthOfBase, sWidthOfShoulder, -sWidthOfElbow, sLengthOfForearm, 0, sLengthOfWrist +
           mLengthOfTool,
16     0, sLengthOfArm, 0, 0, 0, 0,
17     -AbstractRobot.PI2, 0.0, AbstractRobot.PI2, -AbstractRobot.PI2, AbstractRobot.PI2, 0
18   };
19 }
20 protected double[] calculateDH(double[] q){
21   Matrix A01 = FKinematics(q[0] * sTO_RADIANS, 1);
22   Matrix A02 = FKinematics(q[1] * sTO_RADIANS - AbstractRobot.PI2, 2);
23   Matrix A03 = FKinematics(q[2] * sTO_RADIANS + AbstractRobot.PI2, 3);
24   Matrix A04 = FKinematics(q[3] * sTO_RADIANS, 4);
25   Matrix A05 = FKinematics(q[4] * sTO_RADIANS, 5);
26   Matrix A06 = FKinematics(q[5] * sTO_RADIANS, 6);
27   O[0] = A06.get(0,1);
28   O[1] = A06.get(1,1);
29   O[2] = A06.get(2,1);
30   A[0] = A06.get(0,2);
31   A[1] = A06.get(1,2);
32   A[2] = A06.get(2,2);
33   Px = A06.get(0,3);
34   Py = A06.get(1,3);
35   Pz = A06.get(2,3);
36   return A06.getVectorColumn(3,0,2);
37 }
38 ...
```

### A.1.4. Cinemática Inversa del robot TX60L

El algoritmo empleado para la implementación de la cinemática inversa del TX60L, dado en [106], establece una serie de ecuaciones a partir de la tabla de parámetros de DH y de la cinemática directa (detallada en la subsección anterior). El objetivo es obtener una serie de ecuaciones cerradas y no un algoritmo numérico que en muchas ocasiones no permite obtener buenos resultados.

El código correspondiente a dicha implementación se ha omitido debido a su extensión, pero se encuentra disponible junto al código completo de toda la librería en la siguiente dirección web: `http://www.um.es/fem/publications/2017/Robotics`.

### A.1.5. Restricciones físicas del robot TX60L

Una vez que el usuario ha proporcionado los límites entre los que se puede mover cada una de las articulaciones así como los extremos máximos para la velocidad y aceleración (ver Tabla 4.9), se implementan los métodos que nos proporcionan las denominadas restricciones físicas, específicas de cada robot, tal y como se detallan en el Código A.4. En el método *setSpeedRealRobot(int speed)* se ejecuta la acción correspondiente para modificar la velocidad del robot real (ver Línea 19).

Código A.4: Implementación de las restricciones físicas del robot TX60L

```
1
2  public double getJointMinimum(int joint) {
3    if (joint <= 0 || joint > sDOF) return Double.NaN;
4    return sJOINT_MINIMA[joint -1];
5  }
6  public double getJointMaximum(int joint) {
7    if (joint <= 0 || joint > sDOF) return Double.NaN;
8    return sJOINT_MAXIMA[joint -1];
9  }
10 public double getJointSpeedMaximum(int joint) {
11   if (joint <= 0 || joint > sDOF) return Double.NaN;
12   return sSPEED_LIMITS[joint - 1];
13 }
14 public double getJointAccelerationMaximum(int joint) {
15   if (joint <= 0 || joint > sDOF) return Double.NaN;
16   return sACCEL_LIMITS[joint - 1];
17 }
18 protected boolean setSpeedRealRobot(int speed) throws Exception {
19   writeSocket("8\n" + speed + "\n");
20   return true;
21 }
```

### A.1.6. Visualización 3D del modelo del robot TX60L

A la hora de querer visualizar el robot en un entorno tridimensional, el usuario debe implementar el método abstracto *createViewGroup()*, usando elementos gráficos de la librería OSP3D definida en la Subsección 4.3.5. Además, para esta implementación el usuario puede apoyarse en objetos gráficos de tipo OBJ o VRML que facilitan una visualización más realista del modelo del robot. La implementación completa de este método para un robot de muchos GDL puede ser algo tediosa. En el Código A.5 se muestra la implementación de las dos primeras articulaciones del robot TX60L (el resto se definen de forma análoga).

Al mismo tiempo que se define el objeto gráfico, se define un elemento *OrientedBoundingBox* en la misma posición y con las mismas dimensiones para cada una de las articulaciones que conforman el robot y que serán la base para el cálculo de las autocolisiones y detección de colisiones. En la Líneas 30-51 se puede observar la definición de este elemento para la base del robot TX60L.

Código A.5: Implementación de la visualización 3D del robot TX60L

```
1  ...
2  static private final String sBASE = ''org/colos/robots/obj_files/TX60L/'';
3  private AxisRotationWrapper mShoulderRotationZ, mArmRotationY, mElbowRotationY,
       mForearmRotationZ, mWristRotationY, mFlangeRotationZ;
4  protected boolean mHasViewGroup = false;
5  private Group mainGroup;
6
7  protected Group createViewGroup() {
8    if (mainGroup!=null) return mainGroup;
9    mainGroup = new Group();
10   Color orangeColor = new Color(255, 128, 0);
11   mHasViewGroup = true;
12   mainGroup.setTransformation(Matrix3DTransformation.rotationZ(Math.PI));
13   Group robotGroup = new Group();
14   robotGroup.setName(''robotGroup'');
15   robotGroup.setXYZ(0, 0, 0);
16   mainGroup.addElement(robotGroup);
17
18   ElementObject baseObj = new ElementObject();
19   baseObj.setXYZ(0.05, 0, 0.0925);
20   baseObj.setSizeXYZ(0.176, 0.155, 0.176);
21   baseObj.setObjectFile(sBASE+''base.obj'');
22   baseObj.getStyle().setFillColor(orangeColor);
23   Matrix3DTransformation tr = Matrix3DTransformation.rotationZ(Math.PI);
24   tr.multiply(Matrix3DTransformation.rotationX(3 * PI2));
25   baseObj.setTransformation(tr);
26   robotGroup.addElement(baseObj);
27   robotGroup.addElement(baseObj);
28
29   //OrientedBoundingBox0
30   ElementBox OBB0 = new ElementBox();
31   OBB0.setXYZ(0, 0, 0.0925);
32   OBB0.setSizeXYZ(0.238, 0.238, 0.176);
33   OBB0.getStyle().setDrawingFill(false);
34   OBB0.setVisible(false);
35   robotGroup.addElement(OBB0);
36   {
37     OrientedBoundingBox obb0 = new OrientedBoundingBox();
38     obb0.setExtent(new Vector3f((float)(OBB0.getSizeX()/2), (float)(OBB0.getSizeY()/2), (
           float)(OBB0.getSizeZ()/2)));
39     obbTable.put(obb0, OBB0);
40     obbList.add(obb0);
41     Element[] triedro = new Element[3];
42     for (int i=0; i<3; i++) {
43       trihedron[i] = new ElementArrow();
44       trihedron[i].getStyle().setLineColor(Color.RED);
45       trihedron[i].getStyle().setLineWidth(4);
```

```
46        }
47        trihedron[0].getStyle().setLineColor(Color.RED);
48        trihedron[1].getStyle().setLineColor(Color.YELLOW);
49        trihedron[2].getStyle().setLineColor(Color.BLUE);
50        obbTrihedronTable.put(obb0, trihedron);
51      }
52      Group shoulderGroup = new Group();
53      shoulderGroup.setName(''shoulderGroup'');
54      shoulderGroup.setXYZ(0.006, 0, 0.3145);
55      shoulderGroup.setTransformation(Matrix3DTransformation.rotationZ(Math.PI));
56      robotGroup.addElement(shoulderGroup);
57
58      ElementObject shoulderObj = new ElementObject();
59      shoulderObj.setXYZ(0, 0.024, 0);
60      shoulderObj.setSizeXYZ(0.176 * 0.84, 0.256 * 0.59, 0.13);
61      shoulderObj.setObjectFile(sBASE+''shoulder.obj'');
62      shoulderObj.getStyle().setFillColor(orangeColor);
63      shoulderObj.setTransformation(Matrix3DTransformation.rotationX(Math.PI));
64      shoulderGroup.addElement(shoulderObj);
65      mShoulderRotationZ = new AxisRotationWrapper(new ZAxisRotation(), shoulderGroup);
66      shoulderGroup.addSecondaryTransformation(mShoulderRotationZ);
67
68      //OrientedBoundingBox1
69      ElementBox OBB1 = new ElementBox();
70      OBB1.setXYZ(0, 0.024, 0);
71      OBB1.setSizeXYZ(0.19, 0.256, 0.26);
72      OBB1.getStyle().setFillColor(Color.blue);
73      OBB1.getStyle().setDrawingFill(false);
74      OBB1.setVisible(false);
75      shoulderGroup.addElement(OBB1);
76      {
77        OrientedBoundingBox obb1 = new OrientedBoundingBox();
78        obb1.setExtent(new Vector3f((float)(OBB1.getSizeX()/2), (float)(OBB1.getSizeY()/2), (
              float)(OBB1.getSizeZ()/2)));
79        obbTable.put(obb1, OBB1);
80        obbList.add(obb1);
81        Element[] trihedron = new Element[3];
82        for (int i=0; i<3; i++) {
83          trihedron[i] = new ElementArrow();
84          trihedron[i].getStyle().setLineColor(Color.RED);
85          trihedron[i].getStyle().setLineWidth(4);
86        }
87        trihedron[0].getStyle().setLineColor(Color.RED);
88        trihedron[1].getStyle().setLineColor(Color.YELLOW);
89        trihedron[2].getStyle().setLineColor(Color.BLUE);
90        obbTrihedronTable.put(obb1, trihedron);
91      }
92      //Resto de articulaciones ...
93      return mainGroup;
94    }
95    ...
```

Una vez implementada la visualización 3D del modelo del robot el usuario debe implementar los siguientes métodos: *updateView()* donde se actualizarían las posiciones de los diferentes com-

ponentes del modelo tridimensional en el caso del que el robot se mueva a una nueva posición, *attachObject(Element object)* donde se añade un objeto a este modelo predefinido en el caso de que el robot coja algún objeto en el transcurso de su simulación y el método *detachObject(Element object)* para eliminar dicho objeto del modelo en el momento en el que el robot ya no lo tiene. Los detalles de estas implementaciones se pueden ver en el Código A.6.

Código A.6: Actualización e interacción con el modelo del robot TX60L

```
1  protected void updateView() {
2    if(mHasViewGroup){
3      mShoulderRotationZ.mTr.setAngle(mCurrentQ[0] * sTO_RADIANS);
4      mArmRotationY.mTr.setAngle(mCurrentQ[1] * sTO_RADIANS);
5      mElbowRotationY.mTr.setAngle(mCurrentQ[2] * sTO_RADIANS);
6      mForearmRotationZ.mTr.setAngle(mCurrentQ[3] * sTO_RADIANS);
7      mWristRotationY.mTr.setAngle(mCurrentQ[4] * sTO_RADIANS);
8      mFlangeRotationZ.mTr.setAngle(mCurrentQ[5] * sTO_RADIANS);
9     }
10 }
11 public void attachObject(Element object) {
12   if (object == null) System.out.println(``Error: It is necessary to add a element'');
13   else {
14     if(mHasViewGroup){
15       if(mTool){
16         object.setXYZ(0, 0, 0.14 + object.getSizeZ()/2);
17         ElementBox OBBbox = new ElementBox();
18         OBBbox.setXYZ(object.getX(), object.getY(), object.getZ());
19         OBBbox.setSizeXYZ(object.getSizeX(),object.getSizeY(),object.getSizeZ());
20         OBBbox.getStyle().setFillColor(object.getStyle().getFillColor());
21         toolGroup.addElement(OBBbox);
22         OrientedBoundingBox obbObject = new OrientedBoundingBox();
23         obbObject.setExtent(new Vector3f((float)(OBBbox.getSizeX()/2),
24           (float)(OBBbox.getSizeY()/2), (float)(OBBbox.getSizeZ()/2)));
25         obbTable.put(obbObject, OBBbox);
26         obbList.add(obbObject);
27         Element[] triedro = new Element[3];
28         for (int i=0; i<3; i++) {
29           triedro[i] = new ElementArrow();
30           triedro[i].getStyle().setLineColor(Color.RED);
31           triedro[i].getStyle().setLineWidth(4);
32         }
33         triedro[0].getStyle().setLineColor(Color.RED);
34         triedro[1].getStyle().setLineColor(Color.YELLOW);
35         triedro[2].getStyle().setLineColor(Color.BLUE);
36         obbTriedroTable.put(obbObject, triedro);
37       }
38       else super.attachObject(object);
39     }
40   }
41 }
42 public void detachObject(Element object){
43   if(mHasViewGroup){
44     if(mTool){
45       toolGroup.removeElement(object);
46       obbTable.remove(obbObject);
```

```
47          obbList.remove(obbObject);
48          obbTriedroTable.remove(obbObject);
49          double[] pos = toolGroup.toSpaceFrame(new double[]{0,0,0});
50          object.setXYZ(pos[0]-object.getSizeX(), pos[1], pos[2]);
51        }
52        else{
53          flangeGroup.removeElement(object);
54          obbTable.remove(obbObject);
55          obbList.remove(obbObject);
56          obbTriedroTable.remove(obbObject);
57          double[] pos = flangeGroup.toSpaceFrame(new double[]{0,0,0});
58          object.setXYZ(pos[0]-object.getSizeX(), pos[1], pos[2]);
59        }
60      }
61    }
```

### A.1.7. Comunicación del robot TX60L

En el momento de establecer una comunicación con el robot real, como se ha indicado a lo largo de esta tesis, hay que tener en cuenta dos factores importantes: el protocolo de comunicación y el lenguaje propio de programación del robot.

Una vez que estos factores han sido analizados, la implementación de los métodos *connect()* y *disconnect()* que proporciona la API para establecer la conexión con el robot real son sencillos de implementar (ver Código A.7) y hacen uso de la clase *SocketClient*, la aplicación cliente del protocolo de comunicación establecido (ver Figura 4.8) y que será detallada en la Subsección A.1.7.2).

Código A.7: Implementación de la comunicación con el robot TX60L

```
1  protected boolean mIsConnected;
2  public void connect(String hostIP, int portNumber, String user, String password) throws
       Exception {
3    openSocket(hostIP, portNumber, user, password);
4    mIsConnected = true;
5  }
6  public void disconnect() throws Exception {
7    closeSocket();
8    mIsConnected = false;
9  }
```

Establecida la conexión, se han implementado una serie de métodos privados que se encargan de enviar los comandos correspondientes al robot real. Por ejemplo, para mover el robot a una posición articular dada, el usuario simplemente debe usar el método correspondiente *moveToQ(double [] q)*, dado en la Tabla 4.2. Este método internamente se encarga de llamar al método *moveToRealRobot()*(dado en la Subsección A.1.2) que envía al robot real el comando necesario para que se mueva a la posición dada.

Al mismo tiempo, hay una aplicación servidor ejecutándose en el controlador del robot, detallada en la Subsección A.1.7.3, que ha sido instalada para aceptar y procesar estos comandos. Esta

aplicación traduce los comandos que recibe de la la librería al lenguaje de programación propio del robot TX60L descrito en la Subsección A.1.7.1. De esta manera, el robot real realiza las acciones que el usuario desea, sin necesidad de que el usuario aprenda el lenguaje de programación del robot.

#### A.1.7.1. VAL3: Lenguaje propio de programación del robot TX60L

VAL3 es un lenguaje de programación diseñado para controlar los robots de Stäubli en todo tipo de aplicaciones.

El lenguaje VAL3 combina las características básicas de un lenguaje informático de alto nivel en tiempo real estándar y de las funcionalidades específicas del control de una célula de robot industrial:

- Herramientas de control del robot
- Herramientas de modelización geométrica
- Herramientas de control de entradas salidas

En [138] se facilita un manual de referencia donde se explica las nociones indispensables para la programación de un robot, y detalla las instrucciones del lenguaje VAL3, clasificadas según las siguientes categorías:

- Elementos del lenguaje: aplicaciones, programas, bibliotecas, tipos de datos, constantes, variables (variables globales, variables locales y parámetros) y las tareas.
- Tipos sencillos: pequeñas instrucciones, tipo bool (valores posibles true o false), tipo num (representa un valor numérico con aproximadamente 14 cifras significativas), tipo de campo de bits (medio para almacenar e intercambiar bajo forma compacta una serie de bits), tipo String (variables de tipo cadena de caracteres para almacenar textos), tipo dio (usado para vincular una variable VAL3 a un entrada/salida digital del sistema), tipo aio (usado para enlazar una variable VAL3 a una entrada/salida numérica del sistema), tipo sio (permite vincular una variable VAL3 a una entrada/salida serie del sistema o a una conexión mediante socket Ethernet).
- Interfaz de usuario: Las instrucciones de la interfaz de usuario permiten la visualización de mensajes en un página del mando manual reservada a la aplicación y la adquisión de las presiones-teclas en el teclado del mando.
- Tareas: Se caracterizan por tener un nombre único, una prioridad o un periodo, un programa, un estado y la próxima instrucción a ejecutar.

- Bibliotecas: Son aplicaciones VAL3 con variables o programas que pueden volver a ser utilizados por otra aplicación u otras bibliotecas VAL3.
- Control del robot: Contiene diferentes instrucciones que dan acceso al estado de las diferentes partes del robot.
- Posición del brazo: Existen diferentes tipos de datos VAL3 que permiten configurar las posiciones del brazo del robot. Se definen dos posiciones en VAL3: posiciones joint (tipo joint) que proporciona la posición angular de cada eje angular y la posición lineal para cada eje lineal, y puntos cartesianos (tipo point) que proporciona la posición cartesiana del punto central de la herramienta en el extremo del brazo relativo a un plano de referencia.
- Control de los movimientos: Permite controlar la trayectoria para el robot, anticiparse a los movimientos y controlar la velocidad dentro de sus capacidades. Por ejemplo, el método *void movej(joint jPosition, tool tHerramienta, mdesc mDesc)* registra el movimiento hacia la posición *jPosition* especificando la herramienta que lleva el robot para esta tarea (variable *tHerramienta*) y el tipo de movimiento deseado (variable *mDesc*).

#### A.1.7.2. Aplicación Cliente del robot TX60L

La aplicación cliente del robot TX60L denominada *SocketClient* se basa en la clase estándar *java.net.Socket* para establecer la comunicación con el robot real y está constituida principalmente por tres métodos: *openSocket* para establecer la conexión, *closeSocket* para cerrar la conexión, y el *writeSocket* usado para enviar las diferentes sentencias al robot real una vez establecida la conexión.

Código A.8: Definición de la aplicación Cliente para la comunicación con el robot real TX60L

```
1  import java.io.DataOutputStream;
2  import java.net.Socket;
3
4  public class SocketClient {
5    private Socket clientSocket;
6    private DataOutputStream outToServer;
7    private String mIP;
8    private int mportNumber;
9
10   public Socket openSocket(String hostIP, int portNumber, String user, String password)
         throws Exception {
11     mIP = hostIP;
12     mportNumber = portNumber;
13     clientSocket = new Socket(mIP, mportNumber);
14     outToServer = new DataOutputStream(clientSocket.getOutputStream());
15     return clientSocket;
16   }
17
18   public void closeSocket() throws Exception {
19     clientSocket.close();
```

```
20      System.out.println(``The connection is completed'');
21    }
22
23    public void writeSocket(String values) throws Exception {
24      outToServer.writeBytes(values);
25    }
26  }
```

Una vez establecida la aplicación cliente, la librería se apoya en ella para poder interaccionar con el robot real sin necesidad de que el usuario conozca las características de su lenguaje. Por un lado, la librería ofrece al usuario métodos para que él pueda interaccionar directamente con el robot, por ejemplo: openTool() y closeTool(), detallados en el Código A.2. Por otro lado, si se ha establecido conexión con el robot real, todos los métodos de la API están implementados de manera que internamente se ejecuta la orden correspondiente en el robot real sin que el usuario tenga que preocuparse por ello.

#### A.1.7.3. Aplicación Servidor del robot TX60L

Finalmente, se ha definido una aplicación en el controlador del robot (ver Código A.9). Esta aplicación ha sido desarrollada en lenguaje VAL3 y su principal función es recibir sentencias y parámetros, traducirlos al lenguaje VAL3 y hacer que el robot los ejecute. Por ejemplo, algunas de estas funciones pueden ser las siguientes: si se recibe el parámetro "nCod = 0" significa que el robot real debe ir a su posición home (ver Líneas 25 y 26) o si "nCod = 1" el robot real se moverá a la posición cartesiana enviada (ver Líneas 28 - 49).

Código A.9: Aplicación servidor ejecutada en el controlador del robot TX60L

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <programList xmlns="ProgramNameSpace">
3    <program name="start" public="false">
4      <description />
5      <paramSection />
6      <localSection />
7      <source>
8        <code>begin
9  cls()
10 userPage()
11 putln(``Running...'')
12 sioLink(mySocket,io:portEther)
13 clearBuffer(mySocket)
14 //Lanzador del buffering
15 taskCreate "buffering",10,buffering()
16 // Va a posicion inicial la PRIMERA vez
17 call goHome()
18 while (true)
19   putln(``Esperando punto'')
20   // Lectura codigo de movimiento
21   call read_buffer(nCod)
22   put(``nCode: '')
```

```
23      putln(nCod)
24      //Reset del movimiento: LLevamos el robot a la posición home
25      if (nCod==0)
26        call goHome()
27        // Recibe un punto desde Ejs y lo asigna a pMove
28      elseIf (nCod==1)
29        call read_buffer(nCoordX)
30        call read_buffer(nCoordY)
31        call read_buffer(nCoordZ)
32        call read_buffer(nRx)
33        call read_buffer(nRy)
34        call read_buffer(nRz)
35        pMove.trsf.x=nCoordX
36        pMove.trsf.y=nCoordY
37        pMove.trsf.z=nCoordZ
38        pMove.trsf.rx=nRx
39        pMove.trsf.ry=nRy
40        pMove.trsf.rz=nRz
41        putln(``Point: '')
42        putln(pMove.trsf.x)
43        putln(pMove.trsf.y)
44        putln(pMove.trsf.z)
45        putln(pMove.trsf.rx)
46        putln(pMove.trsf.ry)
47        putln(pMove.trsf.rz)
48        movel(pMove,tTool0,mNomSpeed)
49        waitEndMove()
50        // moveJ al punto "PosJ" enviado
51      elseIf (nCod==2)
52        call read_buffer(nJ1)
53        call read_buffer(nJ2)
54        call read_buffer(nJ3)
55        call read_buffer(nJ4)
56        call read_buffer(nJ5)
57        call read_buffer(nJ6)
58        jMove.j1=nJ1
59        jMove.j2=nJ2
60        jMove.j3=nJ3
61        jMove.j4=nJ4
62        jMove.j5=nJ5
63        jMove.j6=nJ6
64        putln(``moveJ to point: '')
65        putln(jMove.j1)
66        putln(jMove.j2)
67        putln(jMove.j3)
68        putln(jMove.j4)
69        putln(jMove.j5)
70        putln(jMove.j6)
71        movej(jMove,tTool0,mNomSpeed)
72        waitEndMove()
73        // moveJ al punto "PosC" enviado
74      elseIf (nCod==3)
75        call read_buffer(nCoordX)
76        call read_buffer(nCoordY)
```

```
77        call read_buffer(nCoordZ)
78        call read_buffer(nRx)
79        call read_buffer(nRy)
80        call read_buffer(nRz)
81        pMove.trsf.x=nCoordX
82        pMove.trsf.y=nCoordY
83        pMove.trsf.z=nCoordZ
84        pMove.trsf.rx=nRx
85        pMove.trsf.ry=nRy
86        pMove.trsf.rz=nRz
87        putln(''moveJ to point: '')
88        putln(pMove.trsf.x)
89        putln(pMove.trsf.y)
90        putln(pMove.trsf.z)
91        putln(pMove.trsf.rx)
92        putln(pMove.trsf.ry)
93        putln(pMove.trsf.rz)
94        movej(pMove,tTool0,mNomSpeed)
95        waitEndMove()
96        // moveL al punto enviado
97      elseIf (nCod==4)
98        call read_buffer(nCoordX)
99        call read_buffer(nCoordY)
100       call read_buffer(nCoordZ)
101       call read_buffer(nRx)
102       call read_buffer(nRy)
103       call read_buffer(nRz)
104       pMove.trsf.x=nCoordX
105       pMove.trsf.y=nCoordY
106       pMove.trsf.z=nCoordZ
107       pMove.trsf.rx=nRx
108       pMove.trsf.ry=nRy
109       pMove.trsf.rz=nRz
110       putln(''moveL to point: '')
111       putln(pMove.trsf.x)
112       putln(pMove.trsf.y)
113       putln(pMove.trsf.z)
114       putln(pMove.trsf.rx)
115       putln(pMove.trsf.ry)
116       putln(pMove.trsf.rz)
117       movel(pMove,tTool0,mNomSpeed)
118       waitEndMove()
119       // moveC al punto enviado
120     elseIf (nCod==5)
121       call read_buffer(nCoordX)
122       call read_buffer(nCoordY)
123       call read_buffer(nCoordZ)
124       call read_buffer(nRx)
125       call read_buffer(nRy)
126       call read_buffer(nRz)
127       pMoveInter.trsf.x=nCoordX
128       pMoveInter.trsf.y=nCoordY
129       pMoveInter.trsf.z=nCoordZ
130       pMoveInter.trsf.rx=nRx
```

```
131         pMoveInter.trsf.ry=nRy
132         pMoveInter.trsf.rz=nRz
133         call read_buffer(nCoordX)
134         call read_buffer(nCoordY)
135         call read_buffer(nCoordZ)
136         call read_buffer(nRx)
137         call read_buffer(nRy)
138         call read_buffer(nRz)
139         pMove.trsf.x=nCoordX
140         pMove.trsf.y=nCoordY
141         pMove.trsf.z=nCoordZ
142         pMove.trsf.rx=nRx
143         pMove.trsf.ry=nRy
144         pMove.trsf.rz=nRz
145         putln(``moveC to point: '')
146         putln(pMove.trsf.x)
147         putln(pMove.trsf.y)
148         putln(pMove.trsf.z)
149         putln(pMove.trsf.rx)
150         putln(pMove.trsf.ry)
151         putln(pMove.trsf.rz)
152         putln(``point Intermediate: '')
153         putln(pMoveInter.trsf.x)
154         putln(pMoveInter.trsf.y)
155         putln(pMoveInter.trsf.z)
156         putln(pMoveInter.trsf.rx)
157         putln(pMoveInter.trsf.ry)
158         putln(pMoveInter.trsf.rz)
159         movec(pMoveInter,pMove,tTool0,mNomSpeed)
160         waitEndMove()
161         //Abrir herramienta
162       elseIf (nCod==6)
163         open(tTool0)
164         //Cerrar herramienta
165       elseIf (nCod==7)
166         close(tTool0)
167         //Cambio de velocidad
168       elseIf (nCod==8)
169         call read_buffer(speed)
170         mNomSpeed.vel=speed
171         //Parar el movimiento
172       elseIf (nCod==9)
173         stopMove()
174       endIf
175     endWhile
176   end
177         </code>
178       </source>
179     </program>
180 </programList>
```

## A.2. Detalles implementación robot Scara

De manera análoga al procedimiento presentado en la sección anterior, en esta sección se van a presentar los detalles de la extensión de la clase *AbstractRobot* para el robot Scara de Omron, presentado en la Subsección 4.5.2.

Los métodos correspondientes a las implementaciones de la información física del robot, la cinemática directa, las restricciones físicas y la visualización 3D del modelo se llevarían a cabo de manera muy similar a como se hizo en el caso del TX60L, pero partiendo de los datos propios de este robot, es decir, considerando las dimensiones de este robot (ver Figura 4.15), sus parámetros de DH dados en la Expresión 4.2 y los valores máximos permitidos para sus articulaciones resumidos en la Tabla 4.10. De entre todos estos métodos, sólo existe diferencia en aquellos cuya implementación envía alguna sentencia al robot real ya que los protocolos de comunicación utilizados en el robot TX60L y en el Scara son diferentes. Estos métodos serían por ejemplo los dados en el Código A.10.

Código A.10: Implementación de algunos métodos para el control del robot Scara

```
1
2  protected void moveRealRobot() {
3    move("P", new PointScara(1000, mCurrentQ));
4  }
5
6  protected boolean setSpeedRealRobot(int speed) throws Exception  {
7    return TelnetClient.write(``@ SPEED '' + speed);
8  }
9
10 protected boolean setAccelRealRobot(int accel) throws Exception  {
11   return TelnetClient.write(``@ ACCEL '' + accel);
12 }
```

Por lo tanto, en esta sección sólo se van a facilitar los detalles de aquellos métodos cuya implementación se haya realizado de manera diferente a la del caso anterior y sean realmente relevantes.

### A.2.1. Cinemática Inversa del robot Scara

En el caso del robot Scara, la resolución de la cinemática inversa se ha llevado a cabo con la implementación del algoritmo basados en grupos de Assur dado en [107]. Partiendo de los modos de ensamblaje dados en la Figura 3.7, este método consiste en calcular la posición del punto C($x_c$ e $y_c$), su velocidad, aceleración y el ángulo formado por el eslabón AC conocidas las posiciones, velocidades y aceleraciones de los puntos A, B, las longitudes $l_{AC}$ y $l_{BC}$ y el modo de ensamblaje. Entonces, la solución de este algoritmo vendrá dada por las ecuaciones de la Figura A.1 y cuya implementación se detalla en el Código A.11.

Código A.11: Implementación de la cinemática inversa del robot Scara

```
1  public double[] inverseKinematics(double[] coordinates, double[] q, boolean shoulder,
       boolean elbow, boolean wrist) {
```

$$\textit{Entradas}: \ \textit{posición} \rightarrow [x \quad y] \ , \ \textit{velocidad} \rightarrow [\dot{x} \quad \dot{y}] \quad y \quad \textit{aceleración} \rightarrow [\ddot{x} \quad \ddot{y}]$$

$$\text{Si } x<0 \quad \left\{ \begin{array}{l} \text{Si } 0 \le y \rightarrow \theta = \arctan\left(\frac{y}{x}\right) + \pi \\ \\ \text{Si no} \rightarrow \theta = \arctan\left(\frac{y}{x}\right) - \pi \end{array} \right\} \text{ y } \left\{ \begin{array}{l} \dot{\theta} = \frac{1}{1+\left(\frac{y}{x}\right)^2} \cdot \frac{\dot{y}\cdot x - y\cdot \dot{x}}{x^2} \\ \\ \ddot{\theta} = \frac{\left(\ddot{y}\cdot x - y\cdot \ddot{x}\right)\cdot\left(x^2+y^2\right) - \left(\dot{y}\cdot x - y\cdot \dot{x}\right)\cdot 2\cdot\left(x\cdot \dot{x} + y\cdot \dot{y}\right)}{\left(x^2+y^2\right)^2} \end{array} \right.$$

$$\text{Si } x=0 \quad \left\{ \begin{array}{l} \text{Si } y<0 \rightarrow \theta = -\frac{\pi}{2} \ , \ \dot{\theta} = \frac{\dot{x}}{y} \text{ y } \ddot{\theta} = \frac{\ddot{x}\cdot y - \dot{x}\cdot \dot{y}}{y^2} \\ \\ \text{Si } y=0 \rightarrow \text{vector nulo} \\ \\ \text{Si } 0<y \rightarrow \theta = \frac{\pi}{2} \ , \ \dot{\theta} = -\frac{\dot{x}}{y} \text{ y } \ddot{\theta} = \frac{\dot{x}\cdot \dot{y} - \ddot{x}\cdot y}{y^2} \end{array} \right.$$

$$\text{Si } 0<x \rightarrow \theta = \arctan\left(\frac{y}{x}\right) \ , \ \dot{\theta} = \frac{1}{1+\left(\frac{y}{x}\right)^2} \cdot \frac{\dot{y}\cdot x - y\cdot \dot{x}}{x^2} \text{ y } \ddot{\theta} = \frac{\left(\ddot{y}\cdot x - y\cdot \ddot{x}\right)\cdot\left(x^2+y^2\right) - \left(\dot{y}\cdot x - y\cdot \dot{x}\right)\cdot 2\cdot\left(x\cdot \dot{x} + y\cdot \dot{y}\right)}{\left(x^2+y^2\right)^2}$$

Figura A.1: Ecuaciones para la resolución de la cinemática inversa mediante grupos de Assur

```
2   if (coordinates.length != sCARTESIAN_ARRAYS_LENGTH) return null;
3   if (q == null) q = new double[sDOF];
4
5   double l_OA = sLengthOfBase - sLengthOfForearm;
6   double l_AB = sWidthOfShoulder;
7   double l_BC = sWidthOfArm;
8   double theta, beta, phi, sbeta, cbeta, theta1, theta2;
9   double x = coordinates[0] - mP0x;
10  double y = coordinates[1] - mP0y;
11  double l3 = mP0z + mHighBasement +(sHighAxisZ - coordinates[2]);
12
13  // Solution of the second joint: q2
14  double a = x * x + y * y - l_AB * l_AB - l_BC * l_BC;
15  double b = 2 * l_AB * l_BC;
16  double c2 = a / b;
17  double s2 = Math.sqrt(1 - c2 * c2);
18  theta = Math.atan2(s2, c2);
19  if ((new Double(theta)).isNaN()) theta = 0.0;
20
21  // Solution of the first joint: q1
22  double sphi = y;
23  double cphi = x;
24  if (x < 0)
25    phi = Math.atan2(sphi, cphi);
26  else {
27    if (x == 0.0) {
28      if (y < 0)
29        phi = -Math.PI / 2.;
30      else {
31        if (y == 0.0)
32          phi = 0; // null vector
```

```
33            else
34              phi = Math.PI / 2.;
35          }
36        } else
37          phi = Math.atan2(sphi, cphi);
38      }
39      sbeta = l_BC * Math.sin(theta);
40      cbeta = l_AB + l_BC * Math.cos(theta);
41      beta = Math.atan2(sbeta, cbeta);
42      if ((new Double(beta)).isNaN()) beta = 0.0;
43
44      if (elbow) {
45        theta1 = (phi - beta);
46        theta2 = theta;
47      } else {
48        theta1 = (phi - beta);
49        theta2 = -theta;
50      }
51      q[0] = theta1 * sTO_DEGREES;
52      q[1] = theta2 * sTO_DEGREES;
53      q[2] = l3;
54      q[3] = coordinates[3];
55
56      if(!checkJointPosition(new double[]{q[0], q[1], q[2], q[3]})) return null;
57      return q;
58    }
```

### A.2.2. Comunicación del robot Scara

En el caso del robot Scara, la implementación de los métodos *connect()* y *disconnect()*, básicos para establecer la conexión con el robot real, se basa en la clase *TelnetClient*, aplicación cliente del protocolo de comunicación Telnet utilizado en la conexión con el robot Scara (ver Subsección A.2.2.2).

Código A.12: Implementación de la comunicación con el robot Scara

```
1
2   public void connect(String hostIP, int portNumber, String user, String password) throws
        Exception{
3     if(openTelnet(hostIP, portNumber, user, password)) {
4       mIsConnected = true;
5       servo(true);
6     }
7     else System.out.println(``Eror: The connection is failed'');
8   }
9
10  public void disconnect() throws Exception{
11    closeTelnet();
12    servo(false);
13    mIsConnected = false;
14  }
```

#### A.2.2.1. Lenguaje propio de programación del robot Scara

El lenguaje del robot Scara de Omron es similar al BASIC (código de instrucción simbólica para principiantes) y hace que los complejos movimientos del robot sean fáciles de programar.

En [139] se encuentran disponibles todos los manuales donde se explica cómo escribir programas de control de robots con el lenguaje Omron, cómo definir todos sus comandos específicos, cómo establecer la conexión con el controlador, ejemplos reales sobre cómo utilizar sus comandos ...

Los controladores de los robots de Omron facilitan un software de soporte, *SCARA Studio*, que incluye una GUI (interfaz gráfica de usuario) de fácil uso. Con Scara Studio se pueden hacer, entre otras cosas, las siguientes:

- Hacer la edición sin conexión de todos los datos utilizados en los controladores de robot.
- Operar y monitorizar robots conectados a sus controladores.
- Hacer la edición en línea de todos los datos utilizados por los controladores de los robots.
- Hacer copias de seguridad y restauración de datos del controlador del robot.
- Establecer conexión con los controladores por vía Ethernet.
- Realizar la entrada de datos en formato de hoja de cálculo.
- Transferir datos entre el controlador en línea y un documento sin conexión mediante arrastrar y soltar.
- Ejecutar comandos en línea usando una ventana de terminal.

Los programas desarrollados con este lenguaje tienen un formato básico, donde cada línea debe contener un comando. Las líneas en blanco (líneas sin comando) provocan un error cuando se compila el programa.

Centrándonos en la conexión Telnet que presenta este controlador, sus comandos de comunicación se clasifican en dos tipos. Un tipo son los comandos que instruyen a la unidad Ethernet para procesar la tarea de comando. El otro tipo de comandos son de control del robot.Estos comandos de control del robot se subdividen en las siguientes 5 categorías: funcionamiento de las teclas, utilidades, manejo de datos, lenguaje del robot y códigos de control.

Los comandos de comunicación para el control del robot presentan el siguiente formato:

$$@\ [] < online\ commands > [< \_command\ option >] < termination\ code >$$

donde "@" indica el inicio del código, "_" representa un espacio en blanco, " $< online commands >''$ y " $< \_command\ option >''$ sería para especificar el comando que se desea ejecutar (ver listado completo en el manual de usuario) y " $< termination\ code >''$ indica el código CR o CRLF. Los elementos entre [] pueden ser omitidos. Los comandos de control del robot comienzan con el código de inicio "@" y se ejecutan cuando se envía a la lista del controlador una instrucción con la última línea con el código de terminación, CR o CRLF. Como excepciones, los códigos de control no requieren un código de inicio ni un código de terminación.

Por ejemplo, si el usuario quiere mover el robot de forma lineal a un punto P1 al 50 % de su velocidad, la expresión de dicho comando sería:

$$@\,MOVE\,L,\ P1,\ VEL = 50$$

#### A.2.2.2. Aplicación Cliente del robot Scara

Partiendo de que el robot Scara se basa en el protocolo Telnet para establecer la comunicación remota con su controlador, en el Código A.13 se facilitan los detalles de la definición de su aplicación cliente *TelnetClient*.

Código A.13: Definición de la clase TelnetClient para la comunicación con el robot real Scara

```
1  import java.net.*;
2  import java.io.*;
3
4  public class TelnetClient{
5    private static final int BUFFERSIZE = 64;
6    private String mHostIP, mUser, mPassword;
7    private int mPortNumber = 23;
8    private Socket mSocket;
9    private InputStream mIn;
10   private OutputStream mOut;
11
12   public TelnetClient (String hostIP, int portNumber, String user, String password){
13     this.mHostIP = hostIP;
14     this.mPortNumber = portNumber;
15     this.mUser = user;
16     this.mPassword = password;
17   }
18
19   public boolean createTelnetConnection (){
20     try {
21       mSocket = new Socket(mHostIP, mPortNumber);
22       mIn = mSocket.getInputStream();
23       mOut = mSocket.getOutputStream();
24       @SuppressWarnings(''unused'')
25       int recvMsgSize = 0;
26       byte [] byteBuffer = new byte[BUFFERSIZE];
27       recvMsgSize = mIn.read(byteBuffer, 0, 3);
28       for (int i=0; i<BUFFERSIZE; i++){byteBuffer[i] = 0;}
```

```
29          recvMsgSize = mIn.read(byteBuffer, 0, 7);
30          for (int i=0; i<BUFFERSIZE; i++){byteBuffer[i] = 0;}
31          mOut.write((mUser+ ''\r\n'').getBytes());
32          mOut.flush();
33          recvMsgSize = mIn.read(byteBuffer, 0, (mUser + ''\r\n'').length());
34          for (int i=0; i<BUFFERSIZE; i++){byteBuffer[i] = 0;}
35          recvMsgSize = mIn.read(byteBuffer, 0, 10);
36          for (int i=0; i<BUFFERSIZE; i++){byteBuffer[i] = 0;}
37          mOut.write((mPassword + ''\r\n'').getBytes());
38          mOut.flush();
39          recvMsgSize = mIn.read(byteBuffer, 0, (mPassword + ''\r\n'').length());
40          for (int i=0; i<BUFFERSIZE; i++){byteBuffer[i] = 0;}
41          byteBuffer = new byte[BUFFERSIZE];
42          String line = '''';
43          while (mIn.read(byteBuffer, 0, 1) > 0){
44            if (byteBuffer[0] == 0x0a) break;
45            line = line + new String(byteBuffer);
46          }
47        } catch (UnknownHostException e) {
48            System.out.println(''Host''+mHostIP+''on port ''+mPortNumber+''not found'');
49            e.printStackTrace();
50            return  false;
51        } catch (IOException e) {
52           System.out.println(''Error found creating socket or getting input&output stream'');
53            try {
54              mSocket.close();
55            } catch (IOException e1) {e1.printStackTrace();}
56            e.printStackTrace();
57            return false;
58        }
59        return true;
60      }
61
62      public boolean closeTelnetConnection (){
63        try {
64          mOut.write((''LOGOUT\r\n'').getBytes());
65          mOut.flush();
66          mSocket.close();
67        } catch (IOException e) {
68            System.out.println(''Error found while closing.'');
69            e.printStackTrace();
70            return false;
71        }
72        return true;
73      }
74
75      public String read (){
76        byte [] byteBuffer = new byte[1];
77        String line ='''';
78        try {
79          while (mIn.read(byteBuffer, 0, 1) > 0){
80            if (byteBuffer[0] == 0x0a) break;
81            line = line + new String(byteBuffer);
82          }
```

```
83      } catch (IOException e) {
84          System.out.println(``Error found while reading.'');
85          e.printStackTrace();
86          return null;
87      }
88      return line;
89    }
90
91    public boolean write (String command){
92      try {
93        byte [] byteBuffer = new byte[BUFFERSIZE];
94        mOut.write((command + ``\r\n'').getBytes());
95        mOut.flush();
96        if (mIn.read(byteBuffer, 0, (command + ``\r\n'').length()) !=
97          (command + ``\r\n'').length()) return false;
98        else return true;
99      } catch (IOException e) {
100         System.out.println(``Error found while writing.'');
101         e.printStackTrace();
102         return false;
103     }
104   }
105 }
```

En la Tabla A.1 se resumen los métodos que el usuario tiene disponibles para controlar el movimiento del robot real una vez establecida la conexión, es decir, una vez que se ha definido el socket del protocolo Telnet. Básicamente estos métodos consisten en el envío de puntos u órdenes de movimiento a un punto dado con un tipo de movimiento específico. Además, también se facilitan los tres métodos privados que la librería utiliza internamente en la implementación de los métodos *connect()* y *disconnect()*.

Se puede observar que a la hora de enviar o recibir un punto desde el controlador del robot real, éste debe tener un formato específico: el nombre debe venir dado de la forma "Pn" donde n es un número entero y sus coordenadas de tipo "double". Para la definición de dicho elemento se ha integrado en la librería una clase auxiliar *PointScara* cuyos constructores y métodos se facilitan en la Tabla A.2.

## A.3. Detalles implementación del componente Belt

La clase *AbstractComponent* detallada en la Sección 4.4 es una clase abstracta que declara, pero no implementa, una serie de métodos y propiedades comunes a todos los componentes robóticos y que el usuario debe implementar en el caso de querer incorporar nuevos componentes a la librería. Dichos métodos fueron resumidos en la Tabla 4.8.

En esta sección se adjuntan los detalles de la implementación de estos métodos y de otras propiedades propias del elemento Belt, con el fin de que sirva de guía para nuevas implementaciones.

Tabla A.1: Métodos disponibles para la interacción con el robot real Scara

| **Métodos para el control del robot real** |
|---|
| **public boolean sendPoint(PointScara point)**<br>*Envía un punto al robot real de tipo "PointScara"* |
| **public boolean sendPoint(String name, double[] coord)**<br>*Envía un punto al robot real facilitando el nombre, dado de la forma P + un número, por ejemplo P3 y las coordenadas del mismo* |
| **public boolean move(String type, String namePoint)**<br>*Ejecuta el movimiento de tipo PTP, P, L o C al punto indicado* |
| **public boolean move(String type, PointScara p)**<br>*Ejecuta el movimiento de tipo PTP, P, L o C al punto indicado* |
| **public boolean move(String type, String namePoint, String option, int optValue)**<br>*Ejecuta el movimiento de tipo PTP, P, L o C al punto indicado pudiendo especificarse un valor para la velocidad "V" dado entre (1 - 750mm/sec ambos inclusive) o speed "S" (de 1 a 100 ambos inclusive)* |
| **public boolean move(String type, PointScara p, String option, int optValue)**<br>*Ejecuta el movimiento de tipo PTP, P, L o C al punto indicado pudiendo especificarse un valor para la velocidad "V" dado entre (1 - 750mm/sec ambos inclusive) o speed "S" (de 1 a 100 ambos inclusive)* |
| **public boolean delay(int delay)**<br>*Ejecuta un delay en el trascurso de las tareas* |
| **public PointScara readPoint(String point)**<br>*Lee un punto desde el controlador del robot* |
| **Métodos privados usados internamente** |
| **private boolean openTelnet(String hostIP, int portNumber, String user, String password)**<br>*Establece una conexión Telnet para poder comunicarse con el robot real* |
| **private boolean closeTelnet()**<br>*Cierra la conexión Telnet previamente establecida* |
| **private boolean servo(boolean on)**<br>*Enciende o apaga los servos del robot real dependiendo de si el parámetro es "true" o "false"* |

Tabla A.2: Métodos y constructores de la clase PointScara

| **Constructores de la clase PointScara** |
|---|
| **public PointScara(String name, double[] coord)**<br>*Constructor del elemento PointScara en el que se facilita el nombre y sus coordenadas* |
| **public PointScara(int pointNumber, int[] coord)**<br>*Constructor del elemento PointScara en el que se facilita el número del punto, por ejemplo, si se facilita un 1, entonces el nombre de este punto será P1 y sus coordenadas de tipo "int"* |
| **public PointScara(int pointNumber, double[] coord)**<br>*Constructor del elemento PointScara en el que se facilita el número del punto, por ejemplo, si se facilita un 1, entonces el nombre de este punto será P1 y sus coordenadas de tipo "double"* |
| **Método de la clase PointScara** |
| **public static PointScara createPoint(String line)**<br>*Método para la creación de un punto de tipo "PointScara" leido desde una cadena de String* |

### A.3.1. Información física del componente Belt

En el caso de las componentes robóticas, la información física común a todos los elementos es muy básica, simplemente es necesario proporcionar las dimensiones: largo, ancho y alto. En el Código A.14 se muestra la implementación detallada de los métodos correspondientes para la cinta transportadora.

Código A.14: Implementación de la información física del componente Belt

```
1  ...
2  static private double sLength = 1.3; // metros
3  static private double sWidth = 0.24;// metros
4  static private double sHigh = 0.88;// metros
5  double mLength = sLength;
6  double mWidth = sWidth;
7  double mHigh = sHigh;
8
9  public double getLength(){return mLength;}
10
11 public double getWidth(){return mWidth;}
12
13 public double getHigh(){return mHigh;}
14 ...
```

### A.3.2. Visualización 3D del componente Belt

De forma análoga a como se ha implementado la visualización de los robots se implementaría la visualización 3D de los componentes robóticos. En el Código A.15 se facilita el detalle de la implementación para la cinta transportadora, cuya definición está basada en elementos básicos de la libreria OSP3D. Por ejemplo, para representar las patas de la cinta transportadora se emplea el elemento *ElementCylinder* y para la visualización del encoder de la cinta se ha incorporado un elemento externo de tipo *Object* (.obj) (incluido en la Línea 104). Además, para obtener una visualización lo más realista posible y poder observar el movimiento de la cinta, se han representado grupos de flechas amarillas sobre la cinta blanca que se irán moviendo cuando la cinta esté conectada (ver líneas 66 - 96).

Código A.15: Implementación de la visualización 3D del componente Belt

```
1  private Group mainGroup;
2  protected boolean mHasViewGroup = false;
3  protected boolean mEncoder;
4
5  protected Group createViewGroup() {
6    if (mainGroup!=null) return mainGroup;
7    mainGroup = new Group();
8    mHasViewGroup = true;
9    mbeltGroup.setName(''beltGroup'');
10   mbeltGroup.setTransformation(Matrix3DTransformation.rotationZ(Math.PI / 2.));
```

```
11    mainGroup.addElement(mbeltGroup);
12    ElementBox basement = new ElementBox();
13    basement.setXYZ(0, 0, mHigh - 0.09);
14    basement.setSizeXYZ(mLength, mWidth, 0.18);
15    basement.getStyle().setFillColor(Color.GRAY);
16    mbeltGroup.addElement(basement);
17
18    ElementBox basement2 = new ElementBox();
19    basement2.setXYZ(0, 0, mHigh - 0.089);
20    basement2.setSizeXYZ(mLength, mWidth - 0.04, 0.18);
21    basement2.getStyle().setFillColor(Color.WHITE);
22    mbeltGroup.addElement(basement2);
23
24    ElementCylinder column1 = new ElementCylinder();
25    column1.setXYZ(mLength / 4., mWidth / 4., (mHigh - 0.18) / 2.);
26    column1.setSizeXYZ(0.05, 0.05, mHigh - 0.18);
27    column1.getStyle().setFillColor(Color.DARK_GRAY);
28    mbeltGroup.addElement(column1);
29
30    ElementCylinder column2 = new ElementCylinder();
31    column2.setXYZ(mLength / 4., -mWidth / 4., (mHigh - 0.18) / 2.);
32    column2.setSizeXYZ(0.05, 0.05, mHigh - 0.18);
33    column2.getStyle().setFillColor(Color.DARK_GRAY);
34    mbeltGroup.addElement(column2);
35
36    ElementCylinder column3 = new ElementCylinder();
37    column3.setXYZ(-mLength / 4., mWidth / 4., (mHigh - 0.18) / 2.);
38    column3.setSizeXYZ(0.05, 0.05, mHigh - 0.18);
39    column3.getStyle().setFillColor(Color.DARK_GRAY);
40    mbeltGroup.addElement(column3);
41
42    ElementCylinder column4 = new ElementCylinder();
43    column4.setXYZ(-mLength / 4., -mWidth / 4., (mHigh - 0.18) / 2.);
44    column4.setSizeXYZ(0.05, 0.05, mHigh - 0.18);
45    column4.getStyle().setFillColor(Color.DARK_GRAY);
46    mbeltGroup.addElement(column4);
47
48    // Control Panel
49    ElementBox control = new ElementBox();
50    control.setXYZ(-mLength / 4., -mWidth / 2. - 0.01, mHigh - 0.09);
51    control.setSizeXYZ(0.02, 0.15, 0.18);
52    control.getStyle().setFillColor(Color.DARK_GRAY);
53    control.setTransformation(Matrix3DTransformation.rotationZ(Math.PI / 2.));
54    mbeltGroup.addElement(control);
55    led = new ElementCylinder();
56    led.setXYZ(-mLength / 4., -mWidth / 2. - 0.02, mHigh - 0.09);
57    led.setSizeXYZ(0.08, 0.08, 0.005);
58    if (mIsConnected)
59      led.getStyle().setFillColor(Color.GREEN);
60    else
61      led.getStyle().setFillColor(Color.RED);
62    led.setTransformation(Matrix3DTransformation.rotationX(Math.PI / 2.));
63    mbeltGroup.addElement(led);
64
```

```
65    // Movement
66    arrowGroup1 = new Group();
67    arrowGroup1.setXYZ(0, 0, mHigh);
68    mbeltGroup.addElement(arrowGroup1);
69    ElementTetrahedron f1 = new ElementTetrahedron();
70    f1.setXYZ(0, 0, 0.003);
71    f1.setSizeXYZ(0.001, mWidth - 0.05, mWidth / 2.);
72    f1.getStyle().setFillColor(Color.YELLOW);
73    f1.setTransformation(Matrix3DTransformation.rotationY(Math.PI / 2.));
74    arrowGroup1.addElement(f1);
75    ElementTetrahedron f2 = new ElementTetrahedron();
76    f2.setXYZ(-0.03, 0, 0.004);
77    f2.setSizeXYZ(0.001, mWidth - 0.05, mWidth / 2.);
78    f2.getStyle().setFillColor(Color.WHITE);
79    f2.setTransformation(Matrix3DTransformation.rotationY(Math.PI / 2.));
80    arrowGroup1.addElement(f2);
81    ...
82    arrowGroup2 = new Group();
83    arrowGroup2.setXYZ(-mLength / 2. + 0.15, 0, mHigh);
84    mbeltGroup.addElement(arrowGroup2);
85    ElementTetrahedron fb1 = new ElementTetrahedron();
86    fb1.setXYZ(0, 0, 0.003);
87    fb1.setSizeXYZ(0.001, mWidth - 0.05, mWidth / 2.);
88    fb1.getStyle().setFillColor(Color.YELLOW);
89    fb1.setTransformation(Matrix3DTransformation.rotationY(Math.PI / 2.));
90    arrowGroup2.addElement(fb1);
91    ElementTetrahedron fb2 = new ElementTetrahedron();
92    fb2.setXYZ(-0.03, 0, 0.004);
93    fb2.setSizeXYZ(0.001, mWidth - 0.05, mWidth / 2.);
94    fb2.getStyle().setFillColor(Color.WHITE);
95    fb2.setTransformation(Matrix3DTransformation.rotationY(Math.PI / 2.));
96    arrowGroup2.addElement(fb2);
97    ...
98
99    // Encoder
100   if (mEncoder) {
101     ElementObject encoderObj = new ElementObject();
102     encoderObj.setXYZ(mLength/4. + 0.25, -mWidth/2. - 0.05, mHigh - 0.08);
103     encoderObj.setSizeXYZ(0.05, 0.13, 0.13);
104     encoderObj.setObjectFile(''org/colos/roboticsLabs/components/obj_files/encoder.obj'');
105     encoderObj.getStyle().setFillColor(Color.DARK_GRAY);
106     encoderObj.setTransformation(Matrix3DTransformation.rotationZ(Math.PI / 2.));
107     mbeltGroup.addElement(encoderObj);
108   }
109 return mainGroup;
110 }
111
112 protected void updateView() {
113   if (mIsConnected)
114     led.getStyle().setFillColor(Color.GREEN);
115   else
116     led.getStyle().setFillColor(Color.RED);
117 }
```

### A.3.3. Otras propiedades del componente Belt

Finalmente, en esta subsección se facilita el detalle de la implementación del resto de métodos propios que componen la clase Belt y que modelan el resto de funcionalidades propias de la cinta transportadora, principalmente métodos para la configuración del movimiento de la cinta y de los objetos depositados sobre ella.

Código A.16: Configuración del movimiento del componente Belt

```
1  // Definición de las variables requeridas
2  static private double sDiameterRoller = 0.09; // unit in meters
3  static private double sMotorFrequency = 50; // unit in Hz
4  static private int sVariatorFrequency = 40; // unit in Hz
5  static private double sRev = 1370;
6  static private double sReductionFactor = 40;
7  static private double sEncoderPulses = 2000;
8
9  int mFrequency = sVariatorFrequency;
10 double mRevMotor;
11 double mRPM;
12 double mLinearSpeed; // linear distance per seconds
13 double mPulses = 0;
14 double timeBelt = 0.0;
15 double distanceBelt = 0.0;
16
17 Thread mEncoderThread = null;
18 Thread mMotionThread = null;
19
20 // Implementación de los métodos
21 public void setVariatorFrequency(int variatorFrequency) {
22   mFrequency = variatorFrequency;
23   initializeVariables();
24 }
25
26 public int getVariatorFrequency() {return mFrequency;}
27
28 public double getNumberOfPulses() {return mPulses;}
29
30 public double getLinearSpeed() {return mLinearSpeed;}
31
32 public void attachObject(Element object) {
33   if (object == null) System.out.println(''Error: It is necessary to add a element'');
34   else {
35     mObjectsSet.add(object);
36     if(mHasViewGroup){
37       object.setXYZ(object.getY(), object.getX()-object.getSizeX() ,sHigh + object.
           getSizeZ()/2);
38       mbeltGroup.addElement(object);  //Adds object to the view
39     }
40   }
41 }
42
43 public void detachObject (Element object){
44   mObjectsSet.remove(object);
```

```
45    if(mHasViewGroup){
46      mbeltGroup.removeElement(object);
47      double [] pos = object.toSpaceFrame(new double[]{0,0,0});
48      object.setPosition(new double[]{pos[1], pos[0], pos[2]});
49    }
50  }
51
52  // Métodos privados y protected
53
54  private void initializeVariables() {
55    mRevMotor = (mFrequency * sRev) / (sMotorFrequency);
56    mRPM = mRevMotor / sReductionFactor;
57    mLinearSpeed = (mRPM * (sDiameterRoller * Math.PI)) / 60.; // (meters/seg)
58  }
59
60  private void moveBelt() {
61    double delta = mLinearSpeed / 10;  // 10 times per second
62    if (arrowGroup1.getX() < mLength / 2. - 0.1)
63      arrowGroup1.setX(arrowGroup1.getX() + delta);
64    else
65      arrowGroup1.setX(-mLength / 2. + 0.1);
66    if (arrowGroup2.getX() < mLength / 2. - 0.1)
67      arrowGroup2.setX(arrowGroup2.getX() + delta);
68    else
69      arrowGroup2.setX(-mLength / 2. + 0.1);
70    timeBelt = timeBelt + 0.1;
71    distanceBelt = distanceBelt + delta;
72    if (mEncoder)
73      mPulses = (distanceBelt * sEncoderPulses) / (sDiameterRoller * Math.PI);
74  }
75
76  private void moveObjects() {
77    double delta = mLinearSpeed / 10;  // 10 times per second
78    for (Element objects : mObjectsSet) {
79      if (objects.getX() <= mLength / 2.)
80        objects.setX(objects.getX() + delta);
81      else {
82        objects.setX(mLength / 2. + 0.05);
83        System.out.println(``Alarm: the object falls from the belt'');
84        if (objects.getZ() > 0)
85          objects.setZ(objects.getZ() - 2 * delta);
86        else
87          objects.setZ(0);
88        }
89    }
90  }
91
92  protected synchronized void playBelt() {
93    if (mMotionThread != null) return; // animation is running
94    mMotionThread = new Thread() {
95      public void run() {
96        while (mMotionThread == Thread.currentThread()) {
97          long currentTime = System.currentTimeMillis();
98          moveBelt();
```

```
99            if (!mObjectsSet.isEmpty())   moveObjects();
100           // adjust the sleep time to try and achieve a constant, animation rate
101           // some VMs will hang if sleep time is less than 10
102           long sleepTime = sDELAY - (System.currentTimeMillis() - currentTime);
103           if (sleepTime < 10) Thread.yield();
104           else {
105             try {
106               Thread.sleep(sleepTime);
107             } catch (InterruptedException ie) {}
108           }
109         }
110       }
111     };
112     mMotionThread.setPriority(Thread.MIN_PRIORITY);
113     mMotionThread.setDaemon(true);
114     mMotionThread.start(); // start the animation
115   }
116
117   protected synchronized void pauseBelt() {
118     if (mMotionThread == null) return; // animation thread is already dead
119       Thread tempThread = mMotionThread; // local reference
120       mMotionThread = null; // signal the animation to stop
121       if (Thread.currentThread() == tempThread) return; // cannot join with own thread so return
122         // another thread has called this method in order to stop the animation thread
123       try { // guard against an exception in applet mode
124         tempThread.interrupt(); // get out of a sleep state
125         tempThread.join(100); // wait up to 1 second for animation thread to stop
126       } catch (Exception e) {
127           System.out.println(``excetpion in stop animation''+e);
128       }
129   }
```

# Referencias


[1] R. Heradio, L. de la Torre, and S. Dormido, "Virtual and remote labs in control education: A survey," *Annual Reviews in Control*, vol. 42, pp. 1–10, 2016.

[2] A. Balestrino, A. Caiti, and E. Crisostomi, "From remote experiments to web-based learning objects: An advanced telelaboratory for robotics and control systems," *IEEE, Transactions on Industrial Electronics*, vol. 56, December 2009.

[3] R. Bordini, L. Braubach, M. Dastani, A. Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci, "A Survey of Programming Languages and Platforms for Multi-Agent Systems," *Informatica*, vol. 30, no. (1), pp. 33–44, 2006.

[4] C. A. Jara, *Aportaciones al aprendizaje constructivo y colaborativo en internet, aplicaciones a laboratorios virtuales y remotos en robótica industrial.* PhD thesis, Universidad de Alicante, Spain, 2010.

[5] C. A. Jara, F. Candelas, J. Pomares, P. Gil, and F. Torres, "Ejs+EjsRL: a free Java tool for advanced Robotics simulation and Computer Vision processing," in *Proceedings of 7th International Conference on Informatics in Control, Automation and Robotics*, pp. 153–160, 2010.

[6] C. Wolfgang, F. Esquembre, and L. Barbato, "Open Source Physics," *Science*, vol. 334, pp. 1077–1078, November 2011.

[7] *The Open Source Physics ComPADRE website. [Online]. Available: http://www.compadre.org/osp, 2017. Accessed: 2017-05-10.*

[8] *Easy Java Simulations wiki. [Online]. Available: http://www.um.es/fem/EjsWiki, 2017. Accessed: 2017-05-10.*

[9] *The Stäubli website. [Online]. Available: http://www.staubli.com, 2017. Accessed: 2017-05-10.*

[10] *The Omron website. [Online]. Available: http://www.omron.com, 2017. Accessed: 2017-05-10.*

[11] M. Ridao, *Generación automática de trayectorias libres de colisiones para múltiples robots manipuladores.* PhD thesis, Escuela Técnica Superior de Ingenieros Industriales de la Universidad de Sevilla, Octubre 1994.

[12] C. Bredin, "Compañeros de equipo," *Revista ABB, ISSN 1013-3135*, no. 1, pp. 26–29, 2005.

[13] G. Raush and S. R., "Coordinación de múltiples robots que ejecutan tareas con incertidumbre temporal.," *VI Congreso de la Asociación Española de Robótica y Automatización Tecnologías de la Producción*, 1999.

[14] S. Chiddarwar and N. Ramesh, "Conflict free coordinated path planning for multiple robots using a dynamic path modification sequence," *Robotics and Autonomous Systems*, vol. 59, pp. 508–518, April 2011.

[15] V. Panwar, N. Kumar, N. Sukavanam, and J.-H. Borm, "Adaptive neural controller for cooperative multiple robot manipulator system manipulating a single rigid object," *Applied Soft Computing*, vol. 12, pp. 216–227, 2012.

[16] K. Souccar and R. Grupen, "Distributed Motion Control for Multiple Robotic Manipulators," *International Conference on Robotics and Automation, Proceedings of the IEEE*, April 1996.

[17] I. Calvo, E. Zulueta, U. Gangoiti, and J. López, *Laboratorios remotos y virtuales.* No. 3, Ikastorratza, e-Revista de didáctica, 2008.

[18] X. Chen, G. Song, and Y. Zhang, "Virtual and Remote Laboratory Development: A Review," *12th Biennial International Conference on Engineering, Construction, and Operations in Challenging Environments; and Fourth NASA/ARO/ASCE Workshop on Granular Materials in Lunar and Martian Exploration*, 14-17 March 2010.

[19] A. Lorandi, G. Hermida, J. Hernández, and E. Ladrŕon de Guevara, "Los Laboratorios Virtuales y Laboratorios Remotos en la Enseñanza de la Ingeniería," *Revista Internacional de Educación en Ingeniería*, vol. 4, 2011.

[20] R. Heradio, L. de la Torre, D. Galan, F. J. Cabrerizo, E. Herrera-Viedma, and S. Dormido, "Virtual and remote labs in education: A bibliometric analysis," *Computers and Education*, vol. 98, pp. 14–38, July 2016.

[21] F. Von-Borstel and J. Gordillo, "Model-Based Development of Virtual Laboratories for Robotics Over the Internet.," *System, Man and Cybernetics, Part A: System and Humans, IEEE Transation on,*, vol. 40, May 2010.

[22] C. Gravier, J. Fayolle, B. Bayard, M. Ates, and J. Lardon, "State of the Art About Remote Laboratories Paradigms - Foundations of Ongoing Mutations," *International Journal of Online Engineering, International Association of Online Engineering*, vol. 4, no. 1, pp. 19–25, 2008.

[23] S. Dormido, "Control learning: present and future," *Annual Reviews in Control*, vol. 28, no. 1, pp. 115–136, 2004.

[24] Z. Nedic, J. Machotka, and A. Nafalski, "Remote laboratories versus virtual and real laboratories," *Frontiers in Education (FIE), 33rd Annual*, 5-8 November 2003.

[25] L. Rosado and J. Herreros, "Nuevas aportaciones didácticas de los laboratorios virtuales y remotos en la ensenñanza de la Física," *International Conference on Multimedia and ICT in Education*, 22-24 April 2009.

[26] F. Candelas, F. Torres, P. Gil, F. Ortiz, S. Puente, and J. Pomares, "Laboratorio virtual remoto para robótica y evaluación de su impacto en la docencia," *Revista Iberoamericana de Automatica e Informatica Industrial (RIAI)*, vol. 1, July 2004.

[27] F. Torres, F. Candelas, S. Puente, J. Pomares, P. Gil, and F. G. Ortiz, "Experiences with virtual environment and remote laboratory for teaching and learning robotics at the university of alicante," *International Journal of Engineering Science*, vol. 22, no. 4, pp. 766–776, 2006.

[28] F. Candelas, C. A. Jara, and F. Torres, "Flexible virtual and remote laboratory for teaching Robotics," *Current Developments in Technology-Assisted Education*, 2006.

[29] F. Arguello, "Diseño y prototipado rápido de un robot articular con 6 grados de libertad," Master's thesis, Escuela Politécnica Nacional, http://bibdigital.epn.edu.ec/handle/15000/16878, 2016.

[30] J. A. Alonso, *Sistemas de prototipado rápido.* Universidad de Vigo (España), http://webs.uvigo.es/disenoindustrial/docs/protorapid.pdf, 2001.

[31] D. Ordóñez, "Diseño y prototipado rápido de un robot stanford con 6 grados de libertad.," Master's thesis, Escuela Politécnica Nacional, http://bibdigital.epn.edu.ec/handle/15000/16707, Septiembre 2016.

[32] M. Fei, Z. Haiou, and W. Guilan, "Application of industrial robot in rapid prototype manufacturing technology," in *Industrial Mechatronics and Automation (ICIMA), 2nd International Conference on*, vol. 1, pp. 218–220, May 2010.

[33] C. Pozna, E. Horvath, and J. Kovacs, "Developing rapid prototype-capable applications for industrial mobile robot platforms," in *Intelligent Engineering Systems (INES), 18th International Conference on*, pp. 203–207, July 2014.

[34] Proceedings of ICETECT, *Analysis of Contemporary Robotics Simulators*, IEEE, 2011.

[35] A. Harris and J. Conrad, "Survey of Popular Robotics Simulators, Frameworks, and Toolkits," *Southeastcon, 2011 Proceedings of IEEE*, 2011.

[36] *Webots. [Online]. Available: https://www.cyberbotics.com/webots.php, 2017. Accessed: 2017-05-10.*

[37] O. Michel, "Webots: Professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, pp. 39–42, 2004.

[38] *Microsoft Robotics Developer Studio. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd939239.aspx, 2017. Accessed: 2017-05-10.*

[39] *Microsoft Robotics Developer Studio - Samples and Tutorials. [Online]. Available: https://msdn.microsoft.com/en-us/library/cc998487.aspx, 2017. Accessed: 2017-05-10.*

[40] *Matlab - The Language of Technical Computing [Online]. Available: https://mathworks.com/products/matlab.html, 2017. Accessed: 2017-05-10.*

[41] *Simulink - Simulation and Model-Based Design. [Online]. Available: https://mathworks.com/products/simulink.html, 2017. Accessed: 2017-05-10.*

[42] L. Zlajpah, "Simulation in robotics," *Mathematics and Computer in Simulation*, vol. 79, pp. 879–897, 2008.

[43] P. Corke, "A robotics toolbox for Matlab," *IEEE Robotics and Automation Magazine*, vol. 3, no. 1, pp. 24–32, 1996.

[44] *Robotics System Toolbox website. [Online]. Available: https://es.mathworks.com/products/robotics.html, 2017. Accessed: 2017-05-10.*

[45] S. Balakirsky, C. Scrapper, S. Carpin, and M. Lewis, "UsarSim: providing a framework for multirobot performance evaluation.," in *Proceedings of Performance Metrics for Intelligent Systems (PerMIS)*, August 21-23 2006.

[46] B. Balaguer, S. Balakirsky, S. Carpin, M. Lewis, and C. Scrapper, "Usarsim: a validated simulator for reserach in robot and automation.," *Workshop on Robot simulators: available software, scientific applications and future trends at IEEE/RSJ IROS*, 2008.

[47] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: a robot simulator for research and education," *Robotics and Automation, IEEE International Conference on*, May 2007.

[48] *USARSim wiki. [Online]. Available: https://sourceforge.net/projects/usarsim/, 2017. Accessed: 2017-05-10.*

[49] *Player website. [Online]. Available: http://playerstage.sourceforge.net/, 2017. Accessed: 2017-05-10.*

[50] *ROS - Wiki. [Online]. Available: http://wiki.ros.org/, 2017. Accessed: 2017-05-10.*

[51] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berguer, R. Wheeler, and A. Ng, "ROS: and open-source Robot Operating System.," in *In ICRA workshop on open source software*, vol. 3, p. 5.

[52] J. Boren and S. Cousins, "Exponential growth of ROS," *IEEE Robotics and Automation Magazine*, vol. 18, pp. 19–20, March 2011.

[53] S. Cousins, B. Gerkey, and K. Conley, "Sharing software with ROS," *IEEE Robotics and Automation Magazine*, vol. 17, pp. 12–14, June 2010.

[54] *ROS - Robots Using ROS. [Online]. Available: http://wiki.ros.org/Robots, 2017. Accessed: 2017-05-10.*

[55] *ROS - Sensors supported by ROS. [Online]. Available: http://wiki.ros.org/Sensors, 2017. Accessed: 2017-05-10.*

[56] *ROS - Integration with Other Libraries. [Online]. Available: http://www.ros.org/integration/, 2017. Accessed: 2017-05-10.*

[57] *ROS - Tutorials. [Online]. Available: http://wiki.ros.org/ROS/Tutorials, 2017. Accessed: 2017-05-10.*

[58] E. Rohmen, S. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework.," in *In Intelligent Robots and Systems (IROS), International Conference on*, pp. 1321–1326, IEEE, 2013.

[59] M. Freese, S. Singh, F. Ozaki, and N. Matsuhira, "Virtual Robot Experimentation Platform V-REP: A Versatile 3D Robot Simulator," *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, vol. 6472, pp. 51–62, 2010.

[60] *The V-REP website. [Online]. Available: http://www.coppeliarobotics.com, 2017. Accessed: 2017-05-10.*

[61] *V-REP User Manual. [Online]. Available: http://www.coppeliarobotics.com/helpFiles/index.html, 2017. Accessed: 2017-05-10.*

[62] *The JRoboOp website. [Online]. Available: http://digilander.libero.it/carmine.lia/jroboop/, 2017. Accessed: 2017-05-10.*

[63] *RosJava wiki. [Online]. Available: http://wiki.ros.org/rosjava, 2017. Accessed: 2017-05-10.*

[64] *RosJava Repository. [Online]. Available: https://github.com/rosjava, 2017. Accessed: 2017-05-10.*

[65] G. Braught, "dLife: A Java library for Multiplatform Robotics, AI and Vision in Undergraduate CS and Research," *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 2012.

[66] *The dLife website. [Online]. Available: http://users.dickinson.edu/ braught/dlife/dLife/Robotics.html, 2017. Accessed: 2017-05-10.*

[67] C. A. Jara, F. A. Candelas, P. Gil, F. Torres, F. Esquembre, and S. Dormido, "EJS + EjsRL: An interactive tool for industrial robots simulation, Computer Vision and remote operation," *Robotics and Autonomous Systems*, vol. 59, pp. 389–401, 2011.

[68] L. Muñoz-Gómez, M. Alencastre-Miranda, and R. Isaac, "Defining and executing practise sessions in a Robotics Virtual Laboratory," in *Proceedings of the Fourth Mexican International Conference on Computer Science (ENC'03)*, 2003.

[69] I. Draganjac, V. Sesar, S. Bogdan, and Z. Kovacic, "An Internet-based System for Remote Planning and Execution of SCARA Robot Trajectories," in *Industrial Electronics, IECON 2008, 34th Annual Conference of IEEE*, 2008.

[70] R. Kumar, P. Kalra, and N. Prakash, "A virtual RV-M1 robot system," *Robotics and Computer-Integrated Manufacturing*, vol. 27, pp. 994–1000, 2011.

[71] M. Korayem, M. Taherifar, S. Maddah, and H. Tovrajzadeh, "Design and programming a graphical user interface for the IcasBot robot using LabView," in *Control Instrumentation and Automation (ICCIA), 2nd International Conference on,*, 2011.

[72] G. Heredia and A. Ollero, "Diez años dando clase de Robótica con HEMERO," in *Actas ROBOT*, pp. 156–161, Noviembre 2011 2011.

[73] *Remote laboratory of Automatic Control - Robot website. [Online]. Available: http://lra.unileon.es/content/physicalsystems/robotic_cell, 2017. Accessed: 2017-05-10.*

[74] *OSP home. [Online]. Available: http://www.opensourcephysics.org/webdocs/about.cfm, 2017. Accessed: 2017-05-10.*

[75] *Open Source Physics. [Online]. Available: https://es.wikipedia.org/wiki/Open Source Physics, 2017. Accessed: 2017-05-10.*

[76] *OSP, Open Source Physics website. [Online]. Available: http://www.opensourcephysics.org/index.cfm, 2017. Accessed: 2017-05-10.*

[77] R. K. Morgan, K. T. Olivares, M. D. Dixson, A. D. Gavrin, M. C. Morrone, J. Lafuze, and A. S. Morrone, eds., *Quick Hits for Teaching with Technology: Successful strategies by award-winning teachers.* 2012.

[78] *OSP tools - Launcher website. [Online]. Available: http://www.opensourcephysics.org/webdocs/Tools.cfm?t=Launcher, 2017. Accessed: 2017-05-10.*

[79] *OSP tools - Tracker website. [Online]. Available: http://www.opensourcephysics.org/webdocs/Tools.cfm?t=Tracker, 2017. Accessed: 2017-05-10.*

[80] *OSP tools - Data Tool website. [Online]. Available: http://www.opensourcephysics.org/webdocs/Tools.cfm?t=Datatool, 2017. Accessed: 2017-05-10.*

[81] C. Wolfgang, *Open Source Physics. A Userś Guide with Examples.* Pearson; Addison-Wesley, 2007.

[82] *OSP User's Guide Example. [Online]. Available: http://www.opensourcephysics.org/items/detail.cfm?ID=7153&S=2, 2017. Accessed: 2017-05-10.*

[83] F. Esquembre, "Easy Java Simulations: A software tool to create scientific simulations in Java.," *Comput.Phys.Common*, vol. 156, pp. 199–204, January 2004.

[84] C. A. Jara, F. Candelas, and F. Torres, "Laboratorios virtuales y remotos basados en EJS para la enseñanza de robótica industrial," *Actas de las XXVIII Jornadas de Automática, Huelva*, 2007.

[85] F. Esquembre, *Creación de Simulaciones Interactivas en Java. Aplicación a la enseñanza de la Física.* Pearson; Prentice Hall, 2005.

[86] C. A. Jara, F. Esquembre, C. Wolfgang, F. A. Candelas, and F. Torres, "A new 3D visualization Java framework based on physics principles," *Computer Physics Communications*, vol. 183, pp. 231–244, 2012.

[87] E. Besada-Portas, J. A. Lopez-Orozco, L. de la Torre, and J. M. de la Cruz, "Remote Control Laboratory Using EJS Applets and TwinCAT Programmable Logic Controllers," *IEEE Transactions on Education*, vol. 56, May 2013.

[88] J. Guzmán, F. Rodríguez, and M. Berenguel, "Laboratorio virtual para la enseñanza de control climático de invernaderos," *Actas de la XXV Jornadas de Automática*, Ciudad Real, 2004.

[89] M. Guinaldo, B. Pérez-Lancho, and E. Sanz, "Laboratorio virtual para el aprendizaje del control térmico en edificios," *Actas de la V Jornadas de Enseñanza vía Internet/Web de la Ingeniería de Sistemas y Automática*, Zaragoza, 2007.

[90] J. Cornellà, A. Doria, and R. Costa, "Fichas interactivas para el aprendizaje de la teoría de sistemas lineales," *Actas de la V Jornadas de Enseñanza vía Internet/Web de la Ingeniería de Sistemas y Automática*, Zaragoza, 2007.

[91] N. Duro, R. Dormido, H. Vargas, S. Dormido-Canto, J. Sánchez, G. Farias, S. Dormido, and F. Esquembre, "An integrated Virtual and Remote Control Lab: The Three-Tank System as a Case Study," *IEEE Computing in Science and Engineering*, vol. 10, no. 4, pp. 50–59, 2008.

[92] C. A. Jara, F. Candelas, and F. Torres, "Creación de laboratorios virtuales de robótica mediante EjsRL," *Actas ROBOT*, pp. 136–143, 28-29 Noviembre 2011.

[93] A. Gil, A. Peidró, Ó. Reinoso, and J. M. Marín, "Implementation and assessment of a virtual laboratory of parallel robots developed for engineering students," *Education, IEEE Transactions on*, 2013.

[94] *Modelo-Vista-Controlador. [Online]. Available: https://es.wikipedia.org/wiki/Modelo-vista-controlador, 2017. Accessed: 2017-05-10.*

[95] Y. Fernández-Romero and Y. Díaz-González, "Patrón Modelo Vista Controlador," *Revista digital de las tecnologías de la informática y las comunicaciones TELEM@TICA*, vol. 11, pp. 47–57, enero-abril 2012.

[96] *Elementos de la Vista de EJS. [Online]. Available: http://www.um.es/fem/EjsWiki/Main/Elements, 2017. Accessed: 2017-05-10.*

[97] F. Torres, J. Pomares, P. Gil, S. Puente, and R. Aracil, *Robots y Sistemas Sensoriales*. Pearson Educación, S.A., 2002.

[98] *Wiki de Robótica - Clasificación de robots. [Online]. Available: http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/clasificacion-de-robots/, 2017. Accessed: 2017-05-10.*

[99] *Robots industriales. [Online]. Available: http://platea.pntic.mec.es/vgonzale/cyr_0204/ctrl_rob/robotica/industrial.htm, 2017. Accessed: 2017-05-10.*

[100] *Robotic Industries Association (RIA) website. [Online]. Available: http://www.robotics.org/, 2017. Accessed: 2017-05-10.*

[101] *International Organization for Standardization (ISO) website. [Online]. Available: https://www.iso.org/home.html, 2017. Accessed: 2017-05-10.*

[102] A. Barrientos, L. Peñín, C. Balaguer, and R. Aracil, *Fundamentos de Robótica 2ª. Edición*. McGraw-Hill/Interamericana de España, S.A.U, 2007.

[103] *International Federation of Robotics (IFR) website. [Online]. Available: https://ifr.org/, 2017. Accessed: 2017-05-10.*

[104] A. O. Baturone, *ROBÓTICA, Manipuladores y robots móviles.* Marcombo Boixareu Editores, 2001.

[105] *Wiki de Robótica - Metodología de Denavit-Hartenberg. [Online]. Available: http://wiki.robotica.webs.upv.es/wiki-de-robotica/cinematica/metodologia-de-denavit-hartenberg/, 2017. Accessed: 2017-05-10.*

[106] M. Wenz and H. Worn, "Solving the Inverse Kinematics Problem Symbolically by Means of knowledge-Based and Linear Algebra-Based Methods," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pp. 1346–1353, September 2007.

[107] A. Noriega and A. García, "Resolución del problema cinematico inverso de un robot Scara mediante grupos de Assur," in *XIX Congreso Nacional de Ingeniería Mecánica*, XIX Congreso Nacional de Ingeniería Mecánica.

[108] *Clasificación de los mecanismos planos. [Online]. Available: http://blog.utp.edu.co/adriamec/files/2012/04/LECCION3-Clasificacion.pdf, 2017. Accessed: 2017-05-10.*

[109] *Wikibooks - Robótica/Planificación de trayectorias. [Online]. Available: https://es.wikibooks.org/wiki/Robótica/Planificación_de_trayectorias, 2017. Accessed: 2017-05-10.*

[110] *Bounding volume. [Online]. Available: https://en.wikipedia.org/wiki/Bounding_volume, 2017. Accessed: 2017-05-10.*

[111] S. Gottschalk, *Collision Queries using Oriented Bounding Boxes.* PhD thesis, University of North Carolina at Chapel Hill, 2000.

[112] *WolframMathWorld - Convex Hulls. [Online]. Available: http://mathworld.wolfram.com/ConvexHull.html, 2017. Accessed: 2017-05-10.*

[113] *Separating Axis Theorem for Oriented Bounding Boxes. [Online]. Available: http://www.jkh.me/files/tutorials/Separating Axis Theorem for Oriented Bounding Boxes.pdf, 2017. Accessed: 2017-05-10.*

[114] *Wiki de Robótica - Sensores en Robótica. [Online]. Available: http://wiki.robotica.webs.upv.es/wiki-de-robotica/sensores/, 2017. Accessed: 2017-05-10.*

[115] A. Soriano, "Control distribuido y coordinación de robots mediante sistemas multiagente," Master's thesis, Instituto Universitario de Automática e Informática Industrial de la Universidad Politécnica de Valencia, 2012.

[116] *Wikibooks - Robótica/Componentes de los Robots. [Online]. Available: https://es.wikibooks.org/wiki/Robótica/Componentes_de_los_Robots, 2017. Accessed: 2017-05-10.*

[117] M. Sánchez, J. García, A. Gómez, and H. Martínez, "Generación de simuladores eficientes para procesos complejos basados en Arquitecturas Distribuidas," in *X Jornadas del Paralelismo*, La Manga del Mar Menor (Murcia), Septiembre 1999.

[118] *Wikipedia - Lenguajes de alto nivel. [Online]. Available: https://es.wikipedia.org/wiki/Lenguaje_de_alto_nivel, 2017. Accessed: 2017-05-10.*

[119] E. Caicedo, E. Bladimir, B. Calvache, J. Evelio, and J. Buitrago, "Laboratorio distribuido con acceso remoto para la enseñanza de la Robótica," *Revista en Educación en Ingeniería*, pp. 51–61, June 2009.

[120] D. Pla-Santamaría, *Localización de información específica en la web.* Universidad Politécnica de Valencia - Servicio de Publicación, 2005.

[121] *Introducción al protocolo Telnet. [Online]. Available: http://es.ccm.net/contents/283-protocolo-telnet, 2017. Accessed: 2017-05-10.*

[122] *Definición de TELNET. [Online]. Available: http://definicion.de/telnet/, 2017. Accessed: 2017-05-10.*

[123] S. Luján, *Programación de aplicaciones web: historia, principios básicos y clientes web.* Editorial Club Universitario, ECU, 2002.

[124] G. Raush, E. Todt, and S. R., "Clasificación y análisis de los métodos de coordinación de múltiples robots,"

[125] M. S. Suárez, *Las Redes de Petri: en la Automática y la Informática.* 1985.

[126] F. DiCesare, G. Harhalakis, J. Proth, M. Silva, and F. Vernadat, *Practice of Petri Nets in Manufacturing.* Springer, 1993.

[127] Q. Jin, Y. Sugasawa, and K. Seya, "Probabilistic behavior and reliability analysis for a multi-robot system by applying Petri net and Markov renewal process theory," *Microelectronics Reliability*, vol. 29, no. 6, pp. 993–1001, 1989.

[128] M. Schoeberl, S. Korsholm, T. Kalibera, and A. Ravn, "A Hardware Abstraction Layer in Java," *ACM Transactions on Embedded Computing Systems*, vol. 10, pp. 1–42, November 2011.

[129] S. Zug, M. Schulze, and J. Kaiser, "Programming abstractions and middleware for building control systems as networks of smart sensors and actuators.," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on,*, pp. 1–8, September 2010.

[130] S. Jorg, J. Tully, and A. Albu-Schaffer, "The Hardware Abstraction layer-Supporting control design by tackling the complexity of humanoid robot hardware," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on,*, pp. 6427–6433, May 31 2014- June 7 2014.

[131] J. Chudoba, L. Preucil, and R. Mazl, "A control System for Multi-Robotic Communities," in *Emerging Technologies and Factory Automation 2006 (ETFA'06), IEEE Conference on*, pp. 827–832, September 2006.

[132] *Tiobe Index website. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci, 2017. Accessed: 2017-05-10.*

[133] *jMonkeyEngine project. [Online]. Available: https://github.com/jMonkeyEngine/jmonkeyengine, 2017. Accessed: 2017-05-10.*

[134] *OSP3D Library. [Online]. Available: http://www.um.es/fem/Javadoc/OSP3D/index.html, 2017. Accessed: 2017-05-10.*

[135] *Definición de encoder. [Online]. Available: http://www.lbaindustrial.com.mx/que-es-un-encoder/, 2017. Accessed: 2017-05-10.*

[136] F. Esquembre, "Facilitating the Creation of Virtual and Remote Laboratories for Science and Engineering Education," *IFAC-PapersOnLine*, vol. 48, no. 29, pp. 49–58, 2015.

[137] D. Flanagan, *JavaScript - La Guía Definitiva.* 2007.

[138] Stäubli, *Manual de Referencia VAL3*, 2008.

[139] *Scara Omron - Manuales. [Online]. Available: https://industrial.omron.es/es/products/x-series#downloads, 2017. Accessed: 2017-05-10.*