UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Ampliación del sistema de cinemática inversa para robots sociales manipuladores

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Ampliación del sistema de cinemática inversa para robots sociales manipuladores

Autor: Mercedes Paoletti Ávila

Tutor: Pablo Bustos García de Castro

Co-Tutor: Luis J. Manso Fernández-Argüelles

Abstract

This work presents improvements on the inverse kinematics system of the social robot Shelly. These improvements enable the robot to use visual feedback to overcome to some extent the inaccuracy caused by motors' backslash and calibration errors.

This new system consists of three main modules: the basic inversed kinematics module, developed in the TFG *Cinemática Inversa en Robots Sociales*, a second module in charge of the 3D–spatial executor of the robot's arm, and a third module, in charge of the whole system's visual feedback.

The objective of this work is to significantly increase the trustworthiness, the efficacy and the efficiency of the new inversed kinematics system for the low cost social robot Shelly.

Resumen

En este trabajo se presenta la ampliación al sistema de cinemática inversa diseñado para el robot Social Shelly y desarrollado en el Trabajo Fin de Grado del alumno, *Cinemática Inversa en Robots Sociales*. Este nuevo sistema se compone de tres módulos principales: el módulo de cinemática inversa básico desarrollado en el TFG, un segundo módulo encargado del control 3D-espacial del efector final del brazo robótico y un tercer módulo encargado del control y la retroalimentación visual del sistema completo. El objetivo que persigue este trabajo es incrementar de forma significativa la fiabilidad, la eficacia y la eficiencia del nuevo sistema de cinemática inversa para el robot social de bajo coste Shelly.

Siguiendo la normativa acerca de los Trabajos Fin de Máster de la Escuela Politécnica de Cáceres (Universidad de Extremadura), por la cual el contenido del trabajo debe reflejar la capacidad del alumno para redactar un documento en inglés, el 50% del contenido de este trabajo se presentará en ese idioma, incluyendo el resumen (abstract) del proyecto y las conclusiones como parte obligatoria.
Por otra parte, el trabajo tendrá su correspondiente versión en Español, incluyendo la traducción del 50% escrito en inglés.

Contents

List of Tables

List of Figures

1 Introducción

Desde finales de la II Guerra Mundial, el desarrollo tecnológico ha experimentado un importante impulso, en especial en el campo de la informática. En muy poco tiempo (apenas 50 años) la informática ha evolucionado a un ritmo vertiginoso, de tal manera que conceptos tan impensables hace escasos años como computador portátil, Internet, *cloud*, *big data*, *tablet* o *smartphone* están tan asimilados en la sociedad actual que no concebimos un mundo sin ellos.

Uno de los términos procedentes de este gran desarrollo tecnológico, que mayor popularidad ha recibido estos años es el de **robótica**. A pesar de no ser un concepto tan novedoso[1] como los que hemos manejado antes (algunos de ellos apenas cuentan con veinte años, como el concepto de *Internet de las cosas*, o incluso con menos, como los *wereables*), la robótica ha llamado poderosamente la atención del ser humano en las últimas décadas, mitificándose y amplificándose su significado a través de películas y novelas de ciencia-ficción, de tal forma que su visión y la de los robots, fuera del ámbito investigador e industrial, se ha desvirtuado en cierta medida.

Acostumbrados a la figura del robot humanoide, capaz de percibir, pensar (con un razonamiento perfecto e inteligente) y actuar de una forma impecable, más parece la imagen de un superhombre que el de una máquina con bastantes e importantes limitaciones, siendo una de ellas el eje central de este trabajo: **el movimiento**.

Para situar correctamente al lector en el marco de este trabajo de investigación, es necesario repasar los conceptos de robótica y robots, repasando su historia y evolución, así como remarcando las líneas de investigación abiertas y los problemas a los que se enfrenta este campo. Esto constituirá una buena base de la que partir, para introducir después, de una forma más amplia y minuciosa, el problema al que pretende dar respuesta este trabajo, así como la solución desarrollada durante él. De

[1] La Revolución Industrial del siglo XVIII impulsó la creación y evolución de agentes mecánicos, como el torno mecánico motorizado de Babbitt (1892) o, más tarde, el mecanismo programable para pintar con spray de Pollard y Roselund (1939). Aunque el concepto de autómata existe desde mucho antes, ya en el siglo I a. C. Herón de Alejandría hablaba de autómatas.

esta forma, este trabajo se estructurará principalmente en cinco apartados:

1. La presente introducción al campo de la robótica, donde señalamos, de una forma muy somera, el hilo conductor que vertebra este trabajo de investigación.

2. Los objetivos que se pretenden alcanzar en el desarrollo de este trabajo, en cuanto al movimiento de los robots se refiere.

3. La base matemática en la que se sustenta el presente trabajo, así como los antecedentes del sistema robótico, los problemas y dificultades que planteaba y que se pretenden resolver mediante el desarrollo de un sistema robótico más robusto y fiable, que de respuesta a los objetivos planteados en el punto anterior.

4. El método empleado, donde se desgranará y especificará la solución alcanzada.

5. Los resultados obtenidos del trabajo, donde se expondrán los objetivos alcanzados y los problemas generados durante la investigación.

6. Unas conclusiones finales que resuman y cierren el trabajo de investigación.

¿Preparado? Esto empieza en... 3... 2... 1...

2 Objetivos

Como adelantábamos en el apartado 1, antes de plasmar los objetivos sobre el papel, conviene introducir al lector en el mundo de la robótica, sus conceptos, su evolución y sus objetivos. De esta forma le resultará mucho más sencillo de comprender los retos que este trabajo se propone investigar y resolver.

2.1 Un poco de cultura robótica nunca está de más

2.1.1 Los comienzos siempre son difíciles

Desde la aparición de los primeros homínidos y a lo largo de toda nuestra historia, el ser humano siempre ha diseñado y construido herramientas, mecanismos y dispositivos que le ayuden en sus tareas diarias, ya sea cazar, sumar y restar, tejer, cocinar, escribir... En definitiva ”artefactos” que mejoren su vida.

Un tipo de artefacto especial es el **autómata**. Estos mecanismos (compuestos de palancas, poleas y dispositivos hidráulicos) eran definidos por los griegos (*automatos*) como máquinas capaces de imitar la figura y los movimientos de un ser vivo, y se empleaban normalmente con fines lúdicos[2].

Siglos después aparecen otros autómatas, como el *Hombre de Hierro* de Alberto Margo (s. XIII), la *Cabeza Parlante* de Bacon (también del s. XIII) o el *Gallo de Estrasburgo* (del s. XIV y que aún se conserva).

Figure 1: Gallo de Estrasburgo

[2]La mitología griega ya apuntaba maneras con las *Kourai Khryseai*, las doncellas doradas forjadas por el dios Hefesto como autómatas con inteligencia, fuerza y habla

Durante el Renacimiento también surgieron numerosos autómatas, como los desarrollados por el inventor español, Juanelo Turriano, cuyo *Hombre de Palo* era capaz de andar y mover la cabeza y los brazos. Aunque los más famosos sean los diseñados por Leonardo Da Vinci, entre los que se encuentran el *León Mecánico* y el robot humanoide construido para Ludovico Sforza.

Durante los siglos XVII y XVIII surgieron nuevos autómatas cuya función principal era entretener al pueblo con su exhibición, como los autómatas japoneses *Karakuri*, usados en el teatro y en festejos religiosos.

A finales del s. XVIII y comienzos del XIX se desarrollan los autómatas más ingeniosos de esta primera aproximación a los robots actuales. Sus funciones ya no son las de entretener, sino la de ayudar al hombre en su trabajo. En especial en la industria textil. Destacan el *telar mecánico* de Vaucanson, la *Hiladora Giratoria* de Hargreaves, el *telar mecánico* de Cartwright y el *telar programable mediante tarjetas perforadas* de Jacquardt.

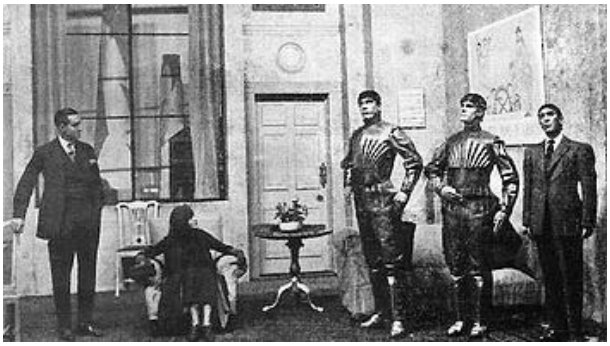
Es en este momento cuando el concepto de autómata cambia radicalmente: ya no es un instrumento lúdico y para divertir a las masas, es un medio de producción. Ha llegado la automatización industrial y con ella, el auge de los robots.

Fecha	Autómata	Inventor
1352	Gallo de Estrasburgo	Desconocido
1499	León Mecánico	Leonardo Da Vici
1525	Hombre de palo	Juanelo Turriano
1738	Flautista, tamborilero, pato, diversas muñecas	Jacques de Vaucanson
1769	Jugador de ajedrez	W. Von Kempelen
1770	Escriba, organista, dibujante	Familia Droz
1770	Hiladora giratoria	Hargreaves
1800	Autómatas Karakuri	Desconocido
1805	Muñeca dibujante	H. Maillardet

Table 1: Ejemplos de autómatas famosos [1]

2.1.2 La robótica desde el s. XX hasta hoy

En 1920 aparece por primera vez la palabra **robot**, dentro de la obra de teatro del dramaturgo checho Karel Capek, *R.U.R. (Rossumovi univerzální roboti)*. La obra trata sobre una empresa que construye humanos artificiales (robots) con el fin de aligerar la carga de trabajo del resto de la humanidad. Pese a ser creadas para ayudar, estas máquinas, capaces de pensar, se enfrentarán con la sociedad hasta destruir la humanidad.



(a) Escena con tres robots (a la derecha) en una representación de lo años 20

(b) Revolución de los robots (1922)

Figure 2: Dos escenas de la obra de teatro *R.U.R*

Etimológicamente, *robot* viene de la palabra checa *robota*, que significa *servidumbre* y en polaco *trabajo* (observemos algunas expresiones, como *brudna robota* = trabajo sucio, o *papierkowa robota* = papeleo). Actualmente, hay múltiples formas de definir *robot*, con más o menos precisión, más o menos abstracto. Por ejemplo, según el **diccionario de Merriam-Webster**, un robot es:

"(1) un aparato mecánico parecido a un ser humano y que actúa como un ser humano.
(2) Una persona eficiente pero insensible.
(3) Dispositivo que realiza automáticamente tareas repetitivas.
(4) Algo guiado por controles automáticos."[2]

Sin embargo puede que la definición del **Instituto de Robots de América** (Robot Institute of America, RIA) sea la que mejor define el concepto de robot:

> "Un robot es un manipulador reprogramable y multifuncional concebido para transportar materiales, piezas, herramientas o sistemas especializados; con movimientos variados y programados, con la finalidad de ejecutar tareas diversas."

Esta última definición explica bastante bien el cometido de un robot: una máquina controlada por un computador que trabaja bajo supervisión humana (ya sea de forma autónoma o teleoperada) que es diseñada para sustituir y/o ayudar al humano en tareas repetitivas, que impliquen cierto riesgo para la vida o sean simplemente inviables para una persona.

Antes de la aparición de los primeros robots industriales, la ejecución de ciertas tareas arriesgadas, como manejar materiales peligrosos, eran llevadas a cabo por *teleoperadores* o *telemanipuladores*. Estas máquinas eran meras extensiones mecánicas que un operario humano controlaba de forma remota, reproduciendo con cierto grado de exactitud los movimientos del trabajador. La Figura. 3 nos muestra el primero de ellos, el *M1* desarrollado en 1948 por Raymond Goertz. Su objetivo era manipular elementos radioactivos y consistía en un dispositivo mecánico maestro-esclavo. El operador podía observar directamente a través de un cristal de seguridad el resultado de sus acciones, y sentía a través del dispositivo maestro, las fuerzas que el mecanismo esclavo ejercía sobre el entorno.

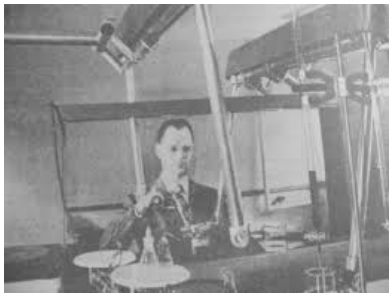
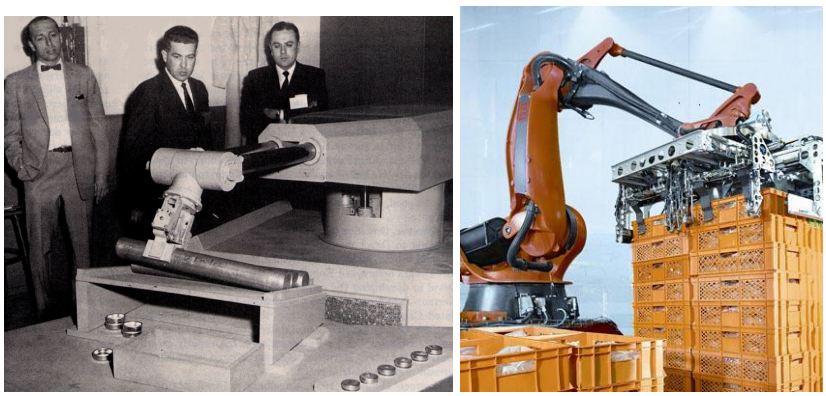


Figure 3: Fotografía de 1948 en la que Raymond Goertz manipula químicos mediante M1, el primer telemanipulador maestro-esclavo mecánico, en el Laboratorio Nacional de Argonne USA

Los primeros robots, como tales, aparecen a finales de los años 40[3] y principios de los 50. En 1954 Goertz presenta su nuevo telemanipulador *E1* con accionamiento eléctrico y dos servos de control [3]. También es en ese año, cuendo el inventor y empreario estadounidense George Devol patenta el primer robot manipulador programable[4]. Años más tarde, y en asociación con el ingeniero físico Joseph Engelberger, Devol desarrollaría el primer brazo robótico, el *Unimate*, instalado en la cadena de montaje de la General Motors.



(a) Robot Unimate 1960 (b) Robot industrial moderno

Figure 4: Evolución de los robots manipuladores industriales

El *Unimate* es considerado como el punto de partida de la **robótica industrial**. A partir de ese momento el objetivo que se marca es sustituir al operador humano por un programa informático. Es el germen inmediato de los *robots industriales*. Según la **Federación Internacional de Robótica (IFR)**, un robot industrial de manipulación:

"[...] es una máquina de manipulación automática, reprogramable y multifuncional [...] 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"

[3]Junto al inicio de la informática y la computación moderna, con la aparición del primer computador, ENIAC, las primeras patentes robóticas surgieron en 1946 con los primeros y primitivos robots del estadounidense George Charles Devol para el traslado de máquinaria.

La estructura de este tipo de robots se divide en:

1. **Articulaciones y eslabones**: motores, engranajes y enlaces rígidos conectados entre sí, que permiten el movimiento del robot.

2. **Pinzas y útiles**: efectores diseñados para manejar herramientas, piezas y materiales, desarrollando tareas propiamente industriales (ensamblar, atornillar, fundir, ordenar...).

En la década de los 60, los robots introducen **sensores**, por ejemplo cámaras, originando los *robots perceptores*. Los sensores aportan información del entorno al robot, lo que permite, además de perfeccionar su funcionamiento, ejecutar diferentes acciones dependiendo de las mediciones obtenidas.
A finales de los 60 comienza la investigación en el campo de la inteligencia artificial[4], alcanzando su auge en los años 80, en los que se desarrollan las técnicas de reconocimiento de voz y de objetos, y se implementan los primeros robots fuera del campo de la industria: robots asistenciales, con fines militares, de seguridad... Es el comienzo de los *robots de servicios*[5].

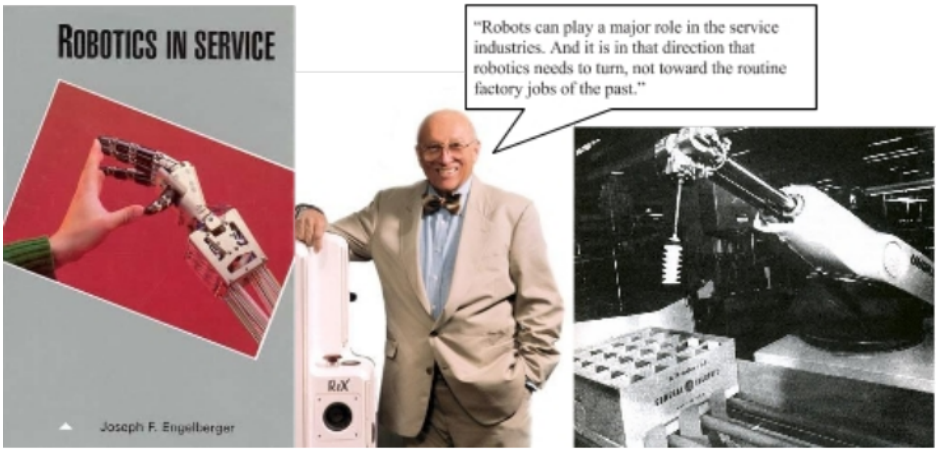


Figure 5: Inicio de la Robótica de Servicio

[4]En paralelo se suceden diversos hitos informáticos que facilitan el estudio de este campo, como el uso del transistor, la aparición de distintos lenguajes de programación (COBOL, Algol, LISP, FORTRAM...), la multiprogramación, el comienzo de los microprocesadores...

[5]Término acuñado por Joseph Engelberger en 1989, año de publicación de su libro *"Robots in Service"*

Los robots de servicios puede realizar múltiples tareas, ayudando tanto a humanos como a otros equipos informáticos, mecánicos, electrónicos... Además, pueden operar de forma ”semi” o completamente autónoma. Dentro de esta categoría se encuentran los *robots sociales*. Son robots que se comunican con los humanos siguiendo una serie de comportamientos sociales y unas normas vinculadas a su función. Existe una gran variedad de robots sociales: de ocio y entretenimiento, educativos (como el recientemente desarrollado LearnBot[5]), publicistas y vendedores (como el robot Gualzru[6, 7]), de asistencia médica...

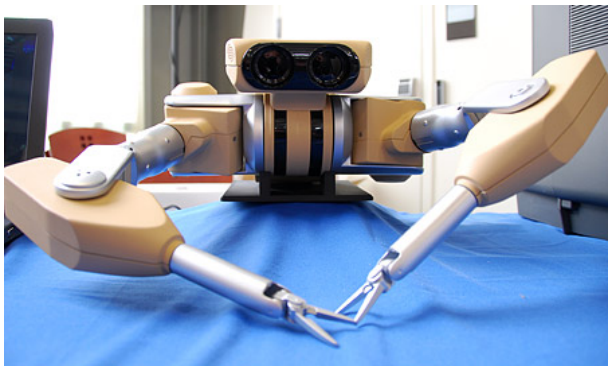


Figure 6: Robot social manipulador (Da Vinci, robot cirujano)

En cualquier caso, todos los tipos de robots sociales tienen el objetivo de satisfacer el bienestar del ser humano: *son herramientas al servicio de la persona*.
Este trabajo se centra en los **robots sociales domésticos manipuladores**, capaces de ofrecer servicios en los que se requiera que el robot interactúe y se comunique con las personas de su entorno, manipulando objetos cotidianos y llevando a cabo tareas del hogar. También se baraja la vertiente de robótica asistiva, más centrada en la interactuación y comunciación con personas mayores o discapacitadas, ofreciéndoles seguridad y comodidad.

2.2 Líneas de investigación en el campo de la Robótica

Una vez repasada la evolución de los robots, acotando el rango de aplicación de este trabajo a los robots sociales domésticos manipuladores, es momento de presentar los

retos y problemas a los que se enfrenta la robótica moderna.

Aún queda lejos la construcción de un robot con capacidades equiparables a las de un ser humano, por los retos que representa, tanto en el campo hardware como en el software. A pesar del abaratamiento de los sensores y en general de todos los componentes electrónicos, así como las mejoras realizadas en los procesadores y dispositivos de almacenamiento, el hardware de un robot supone un gran obstáculo económico. Si se pretende conseguir un robot robusto, multitarea y de gran precisión se debe estar dispuesto a desembolsar una cantidad importante de dinero. Por ejemplo, el robot industrial *Kuka KR 60-3* puede costar más de 60.000 euros, mientras que el robot social *Maggie* alcanzó los 30.000 euros.
Por otra parte el software del robot también supone un reto en sí mismo. En cierta medida, podríamos decir que la ”inteligencia” del robot viene determinada por el sistema y la arquitectura software que disponga. Gracias a los procesos software que se ejecutan dentro del robot, éste es capaz de extraer información útil de la masa desestructurada de datos que recibe de los sensores y del resto del hardware que lo compone. De esta forma es capaz de reaccionar o comportarse de manera distinta dependiendo de la información que procese. Dos objetivos en los que actualmente se está investigando mucho son:

1. El **reconocimiento de patrones**, como por ejemplo cierto tipo de objetos (lo que implica el subcampo de la visión artificial) o el habla (estrechamente ligado al procesamiento del lenguaje natural). Lo que se pretende es no sólo reconocerlos y diferenciarlos, sino también comprenderlos y aprenderlos.

2. Los **planificadores** de inteligencia artificial. Es decir, el estudio de las normas y reglas de comportamiento que guían las acciones y ejecuciones de los robots en los entornos en los que trabajan.

3. La **cinemática** del robot, la capacidad de coordinar el movimiento (posición, velocidad y aceleración) de cada elemento estructural del robot para generar una

trayectoria determinada con el objetivo de alcanzar una posición determinada en el espacio. El estudio de la cinemática es imprescindible para los robots manipuladores, que necesitan moverse y manipular objetos con precisión para cumplir con su función.

Al final, el objetivo que se persigue al investigar estas áreas es construir robots **inteligentes**, capaces de distinguir las situaciones y entornos en los que trabajan, aprender de esos entornos y tomar decisiones funcionales en base a estos factores, y **autónomos**, capaces de moverse e interactuar con los objetos, elementos o las personas de su entorno.

Este trabajo pretende ahondar en la segunda problemática, en la autonomía del robot entendida desde el punto de vista del movimiento, es decir desde el punto de vista de **la cinemática del robot**.

2.2.1 Problema de la cinemática

Como adelantábamos en el apartado 2.1.2, los robots se componen de articulaciones y eslabones, pudiendo disponer de efectores finales como en el caso de los robots manipuladores. La cadena formada por esas articulaciones y eslabones forman lo que llamamos **cadena cinemática**.

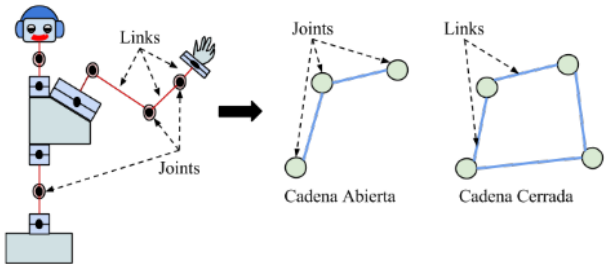


Figure 7: Esquema de una cadena cinemática. Tipos de cadenas cinemáticas

De forma simplificada, la cadena cinemática que forman los robots están compuestas por segmentos fijos (eslabones o **links**) unidos a conectores móviles (motores, **joints** o articulaciones) que le proporcionan la capacidad de movimiento.

Según la disposición de estos elementos podemos distinguir dos tipos de cadenas cinemáticas:

1. **Cadenas cinemáticas abiertas (seriales)**: aquellas en las que existen (y se diferencian claramente) un extremo inicial (una base) y un extremo final (un efector). En este tipo de cadenas todos los joints son activos lo que significa que cada uno tiene capacidad para moverse de forma independiente. Los robots seriales utilizan este tipo de cadenas en sus estructuras, como los robots *SCARA* o *KUKA*.

2. **Cadenas cinemáticas cerradas (paralelas)**: en estas cadenas los joints se dividen en activos, los que producen el movimiento, y pasivos, los que adaptan la estructura a ese movimiento. Los robots paralelos poseen este tipo de cadenas. Cuentan con un órgano terminal, o plataforma móvil conectada a la base mediante varias cadenas cinemáticas seriales independientes. Se trata de robots muy rígidos y ligeros, capaces de soportar altas aceleraciones. Un ejemplo es el robot industrial *Adept Quattro* cuya arquitectura es muy parecida a la del joystick *Falcon*.

Otra característica importante de las cadenas cinemáticas es el **grado de libertad** (GDL o DOF) de las mismas. Para calcularlo se sigue la **fórmula de Grübler**:

$$DOF = \lambda \cdot (N - j - 1) + \sum_{i=1}^{j} f_i \tag{1}$$

Donde λ es el grado de libertad del espacio de trabajo de la cadena cinemática (por ejemplo, en el espacio 3D sería 6, tres traslaciones más tres rotaciones), N es el número de links de la cadena, incluyendo la base sobre la que se sostiene, j es el número de joints de la cadena cinemática y f_i es el grado de libertad de cada joint[6]. Normalmente, los robots manipuladores son cadenas cinemáticas abiertas en las que

[6]Los joints pueden tener tantos grados de libertad como movimientos puedan hacer. Por ejemplo, si un joint sólo puede girar en una dirección tendrá sólo un grado de libertad. Sin embargo, si un joint puede girar y deslizar en una dirección tiene dos grados de libertad, o si puede girar en tres direcciones tendrá tres grados de libertad. Normalmente los joints tienen un máximo de tres grados de libertad

el grado de libertad coincide con el número de joints

Una vez vistos estos conceptos y, haciendo hincapié en el apartado 2.2 anterior, el problema al que se enfrentan todos los robots manipuladores, sociales o industriales, de cadena abierta o cerrada, con más o menor grado de libertad, es el *movimiento* y la *manipulación*. Estos deben ser capaces de *moverse* hacia ciertas posiciones de destino y *manipular* los objetos que se encuentren en esas poses[7]. Para enfrentarse a este problema podemos adoptar dos posibles soluciones:

1. Mediante posiciones de destino **enseñadas y aprendidas**: por ejemplo los robots industriales se mueven hacia poses objetivo que les han sido enseñadas con anterioridad.

   El primer paso es entrenar al robot para que lleve su efector hacia ciertos puntos de destino de tal forma que, en el segundo paso, cuando los puntos son alcanzados, se leen los valores de los sensores de posicionamiento de cada motor y se almacenan sus valores angulares. Así, cuando el robot deba regresar a esa pose, leerá los ángulos correspondientes que debe asignar a cada motor. Este método sólo nos permite enseñarle un número limitado de posiciones.

2. Mediante el **cálculo interno** de los valores angulares para alcanzar la pose objetivo: en este caso, al robot se le indica una posición y una orientación de destino en el espacio, sin existir un entrenamiento previo. El encargado de calcular los valores angulares que deben adoptar los motores para alcanzar la posición objetivo es el propio robot.

   Esta forma de resolver el problema permite mover el efector a todas las posiciones que se encuentren dentro del rango de trabajo del robot, lo que al final le permite alcanzar un ilimitado número de posiciones.

En ambos casos, se está haciendo uso de la **cinemática del robot**, aunque aplicada de dos formas distintas. En el primer caso se trata de **cinemática directa**, mientras que

[7] A las posiciones de destino las llamaremos también **poses** o **targets**)

en el segundo se utiliza la **cinemática inversa**.

Para explicar estos dos conceptos, tomaremos como ejemplo un brazo robótico de dos grados de libertad en un sistema de coordenadas 2D, que forma un cadena cinemática abierta, tal y como observamos en la figura 8. Esta cadena se compone de una base, dos joints (los nodos 1, que sería el hombro, y 2, el codo), dos links de longitudes $L1$ (brazo) y $L2$ (antebrazo), y un efector final rígido (el nodo 3 o mano):

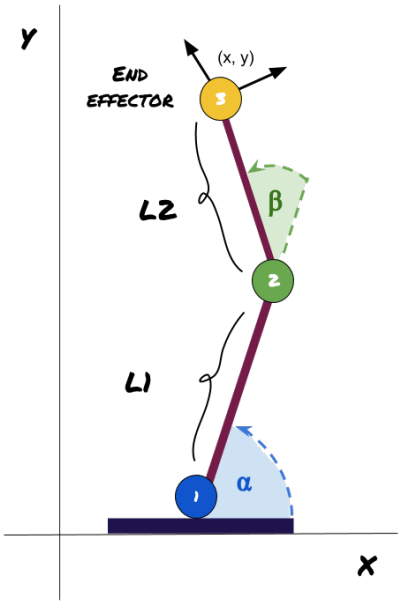


Figure 8: Esquema de brazo robótico en un sistema 2D

Cinemática Directa El problema al que se enfrenta la cinemática directa es *determinar la posición del efector final*, $P3$, es decir calcular sus coordenadas (x, y) y su orientación, conociendo de antemano los valores angulares de cada joint y las longitudes de cada link:

$$P3(p3_x, p3_y) = F(\theta, L), \text{ donde } \theta = [\alpha, \beta] \text{ y } L = [L1, L2].$$

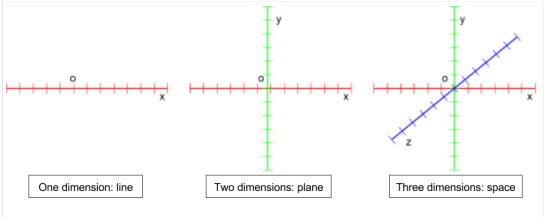
En el ejemplo sencillo de la figura 8, la función de cinemática directa se formularía como:

$$p3_x = L1 \cdot \cos(\alpha) + L2 \cdot \cos(\alpha + \beta)$$
$$p3_y = L1 \cdot \sin(\alpha) + L2 \cdot \sin(\alpha + \beta)$$

Para poder obtener estas ecuaciones se necesita describir y representar la geometría del brazo robótico, utilizándose en este caso la **Representación de Denavit-Hartenberg**. Este método propone un modelo matricial para establecer sistemáticamente un sistema de coordenadas[8] para cada elemento de la cadena cinemática. De esta forma D-H será una matriz de transformación homogénea que representa los sistemas de coordenadas de cada elemento del brazo robótico con respecto al sistema de coordenadas del elemento inmediatamente anterior[9]

Además de facilitar la descripción geométrica del sistema robótico, la representación D-H proporciona un método algoritmico universal para derivar las ecuaciones

[8]Tengamos en cuenta que un **sistema o marco de referencia** no es otra cosa que un conjunto de ejes que forman un sistema de coordenadas con el que calculamos y estudiamos el movimiento y la pose de un cuerpo, marcando la ubicación del ente observador respecto al cuerpo observado. A lo largo de este proyecto utilizaremos marcos de referencia basados en ejes y coordenadas **cartesianos**.
Por otra parte, el sistema cartesiano está definido por uno, dos o tres ejes ortogonales (perpendiculares) entre sí, idénticamente escalados, de tal forma que la posición de un punto P en un sistema cartesiano de dos dimensiones (2D) vendrá dada por el valor de las coordenadas X e Y del punto, *P(Px, Py)*, que a su vez son las proyecciones ortogonales del vector de posición de P, con origen en el centro del sistema O y extremo en el punto P, sobre los ejes X e Y.



[9]Básicamente, lo que D-H propone es que en un robot, cada joint tiene su propio sistema de referencia (que puede estar trasladado del anterior joint por la longitud del link que los une o rotado por el joint que le precede) por lo que, si se quiere calcular la posición de una determinada articulación o de un punto cualquiera del robot, habrá que realizar una serie de transformaciones para pasar de un sistema de referencia a otro. Estas transformaciones consisten en una serie de productos matriz por vector de tal forma que, si tenemos un brazo robótico como el de la figura 8 en el espacio 3D, por ejemplo, y queremos calcular las coordenadas del nodo 3, $P3_3 = (p3_x, p3_y, p3_z)_3$ en el sistema del nodo 1, $P3_1 = (p3_x, p3_y, p3_z)_1$, habría que calcular las matrices de transformación para pasar del nodo 3 al nodo 2 (M_3^2), y del nodo 2 al nodo 1 (M_2^1), de tal forma que nos quedaría una ecuación tal que: $P3_1 = M_2^1 \cdot M_3^2 \cdot P3_3$.
Si esta expresión la pasamos a forma matricial nos quedaría una expresión como:

$$\begin{pmatrix} p3_x \\ p3_y \\ p3_z \\ 1 \end{pmatrix}_1 = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix}_2^1 \cdot \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix}_3^2 \cdot \begin{pmatrix} p3_x \\ p3_y \\ p3_z \\ 1 \end{pmatrix}_3 = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}_2^1 \cdot \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}_3^2 \cdot \begin{pmatrix} p3_x \\ p3_y \\ p3_z \\ 1 \end{pmatrix}_3$$

cinemáticas de cualquier brazo o sistema robótico. Con esta notación se pueden expresar tanto la posición como la orientación del efector final en función del desplazamiento y las rotaciones de sus articulaciones.
En una cadena cinemática abierta o serial la solución siempre es única, es decir, para unos parámetros específicos (unos valores angulares para las articulaciones y unas longitudes para los eslabones) habrá siempre una única posición del efector final dada por: $PN_0 = M_1^0 \cdot M_2^1 \cdot M_3^2 \cdot \ldots \cdot M_N^{N-1} \rightarrow PN_0 = \prod_{i=1}^{N} M_i^{i-1}$
Por el contrario, una cadena paralela no tiene una única solución, por lo que para unos parámetros determinados puede haber más de una posición final del efector. La cinemática directa de un robot paralelo es bastante más difícil que la de un robot serial, y se resuelve mediante método numéricos, no geométricos.

Cinemática Inversa Cuando hablamos del problema de cinemática inversa (*Inverse Kinematics*), su dificultad de resolución da un giro de 180º, tanto para las cadenas cinemáticas abiertas, en las que el problema se complica (su resolución precisa de métodos números y ecuaciones no lineales en cierta medida complicadas), como en las cerradas, en las que, por contra, la resolución del problema se simplifica, empleándose incluso sencillos métodos geométricos.
La cinemática inversa permite trasladar y rotar el efector final de la cadena cinemática desde un punto inicial A a un punto objetivo P, una vez conocidas una serie de variables:

1. Las coordenadas del punto objetivo, P, que debe alcanzar el efector final.
2. El número de joints que componen la cadena cinemática y los límites que éstos presenten (por ejemplo los valores angulares mínimo y máximo que las articulaciones puedan alcanzar).
3. La longitud de los links que unen esos joints.

Con estos datos, la cinemática inversa se encarga de *resolver la secuencia de ángulos de los joints para mover el efector final desde una posición inicial, hasta la posición*

objetivo.

Tomemos como referencia el ejemplo de brazo robótico 2D que nos presenta la imagen 8[10]. El objetivo único de la cinemática inversa es encontrar la función inversa a la de cinemática directa, una F^{-1} que devuelva los ángulos de los joints, dadas las coordenadas del punto objetivo $P = (p_x, p_y)$ y las longitudes de los segmentos del brazo, $L = [L1, L2]$:

$$\theta = F^{-1}(P, L), \text{ donde } \theta = [\alpha', \beta'], P = (p_x, p_y) \text{ y } L = [L1, L2].$$

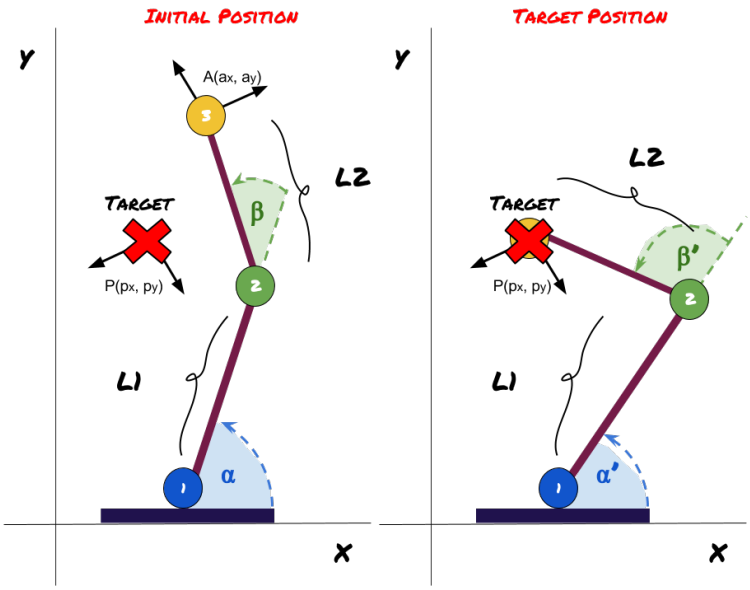


Figure 9: Brazo robótico de cadena abierta: problema de la cinemática inversa

A estas alturas puede verse con claridad el problema que supone resolver esta F^{-1}: ¿cómo se puede calcular la inversa de las funciones de cinemática directa, $p_x = L1 \cdot \cos(\alpha') + L2 \cdot \cos(\alpha' + \beta')$ y $p_y = L1 \cdot \sin(\alpha') + L2 \cdot \sin(\alpha' + \beta')$? La solución al problema de cinemática inversa no es precisamente obvia ni sencilla en las cadenas abiertas ya que:

1. La solución está íntimamente ligada a la configuración del robot, de la que

[10]Note el lector que se trata de una cadena abierta. En este trabajo se ilustrarán los ejemplos de cinemática inversa, así como la problemática que surge en torno a ella, tomando como referencia las cadenas cinemáticas seriales o abiertas

depende de forma directa (número y organización de DOF, enlaces... etc). Además pueden darse infinitas soluciones para los mismos datos iniciales.

2. La solución no es sistemática, puesto que las ecuaciones de cinemática del robot no son lineales.

3. No siempre existe una solución cerrada.

Por todo ello, para dar una solución aceptable al problema de la cinemática inversa en cadenas abiertas es necesario recurrir a métodos matemáticos genéricos, que permitan solucionarlo en tiempo real (mediante seguimiento de trayectorias) y añadir restricciones propias de la configuración de la cadena cinemática, acercándose a la solución más óptima.

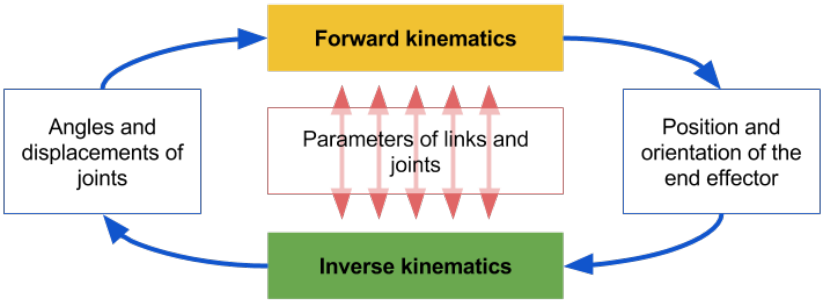


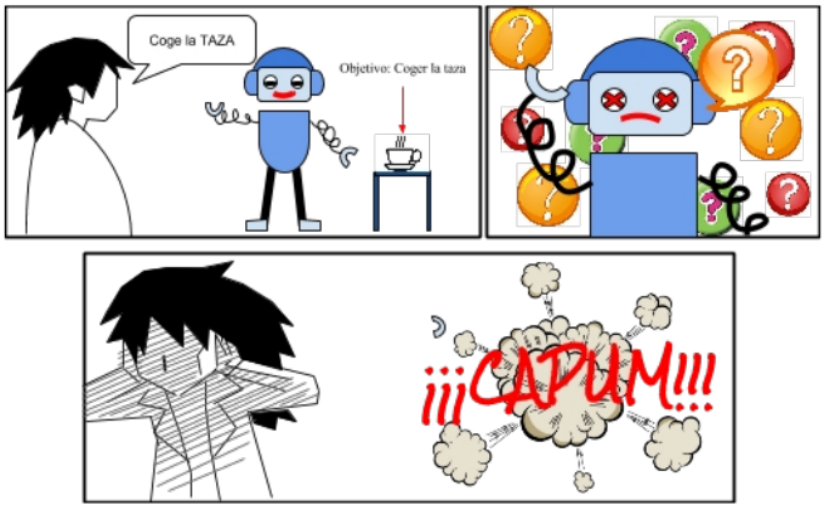
Figure 10: Comparativa entre la cinemática directa y la cinemática inversa

2.3 Propósito del TFM

Una vez introducidos los conceptos de *robots sociales de servicio* y de exponer una visión general del *problema cinemático* al que se enfrentan aquellos robots que necesitan manipular objetos, estamos en una posición óptima para marcar los objetivos que pretende alcanzar este Trabajo Fin de Máster.

Hasta ahora, la temática debe resultar bastante conocida para el lector, ya que este trabajo de investigación toma como punto de partida el anterior Trabajo Fin de

Grado, *Cinemática Inversa en Robots Sociales*[8]. Sin entrar en muchos detalles[11], el problema al que pretendía dar solución el anterior TFG era el *desarrollo de un sistema de cinemática inversa para robots sociales*, partiendo completamente desde cero.



Para ello, se estudiaron diferentes algoritmos que resuelven el problema de la cinemática inversa, decantándonos finalmente por implementar el algoritmo de búsqueda local de **Levenberg-Marquardt**, por su facilidad de comprensión, su rapidez a la hora de encontrar soluciones y su sencillez a la hora de programar.

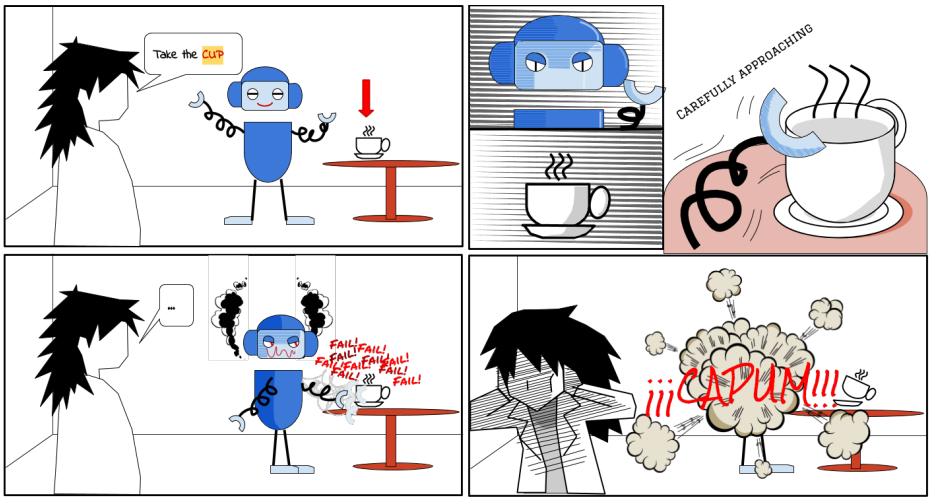
Por otra parte, para la construcción de este primer sistema cinemático (enteramente programado en *C++*) se empleó el framework de código abierto para robótica **Robocomp**, desarrollado por el laboratorio de robótica y visión artificial de la Escuela Politécnica de Cáceres, *Robolab*[12], en la Universidad de Extremadura. Este framework proporciona una serie de librerías, componentes y herramientas software que permiten desarrollar programas para robots de una forma fácil, eficaz y distribuida.

Por último, este software de cinemática inversa se desarrolló con vistas para ser empleado en el robot social **Ursus 3.0**, desarrollado por el equipo de Robolab, donde se testeó repetidamente hasta alcanzar unos resultados aceptables, aunque no óptimos.

[11] Dedicaremos más adelante un apartado completo al sistema de cinemática desarrollado durante el anterior TFG, con el objetivo de profundizar en los problemas y las limitaciones que planteaba y que dieron lugar a un nuevo sistema robótico, que es el objeto de esta investigación.

[12] Se puede visitar el sitio web del grupo en la URL `https://robolab.unex.es/`

Es precisamente a causa de esos resultados poco óptimos y sujetos a ciertos problemas y limitaciones, impuestos tanto por la arquitectura software, demasiado compleja, como por las carencias que la propia implementación software del algoritmo sufría, que surge esta nueva revisión del sistema de cinemática inversa.



En este trabajo de investigación nos proponemos ahondar en la problemática de la cinemática inversa, desarrollando un nuevo sistema software que tenga como objetivos principales:

1. Solventar las deficiencias que presentaba el antiguo sistema: mejorar el ratio de error mediante autocalibración, estabilizar y normalizar las soluciones aportadas y ampliar el rango de casos y situaciones soportadas.

2. Implementar una estructura software más sencilla, que sea a la vez robusta y escalable, además de fácil de mantener y mejorar. Para ello se diseñará una arquitectura modular en el que cada módulo se encargará de realizar una única operación que complemente el correcto funcionamiento del sistema completo.

Este nuevo sistema se basará en la implementación del algoritmo de **Levenberg-Marquardt** desarrollada en el anterior TFG, que será convenientemente revisada, simplificada y mejorada.

Siguiendo la línea marcada en el TFG, la nueva solución[13] será desarrollada mediante técnicas de programación orientada a componentes (C.O.P. de sus siglas en inglés), usando el framework de código abierto para robótica **Robocomp**[9]. En este caso, el lenguaje que se utilizará para desarrollar el proyecto principal será *C++*, mientras que el lenguaje empleado en la fase de testeo y pruebas será *Python*.

Por último, aplicaremos el sistema cinemático sobre la nueva versión del robot social Ursus, llamado ahora **Shelly**, desarrollado también por Robolab. El objetivo final sigue siendo el mismo: que el robot mueva su brazo para alcanzar cualquier objeto que se le marque como objetivo, como puede ser una taza, un lápiz o un libro, en un tiempo y con una carga de cómputo razonable.

[13] Esta solución incluirá tanto el nuevo sistema de cinemática inversa como su propio sistema de validación y pruebas en un robot real.

3 Antecedentes

Una vez acotado el problema a la creación de un software de cinemática inversa aplicado a robots sociales-manipuladores, en este apartado repasaremos la base matemática en la que se sustentan tanto el antiguo como el nuevo sistema cinemático, con el objetivo de profundizar en el núcleo de la solución adoptada.
Además, explicaremos resumidamente el estado final que alcanzó el antiguo software de cinemática. De esta forma, podremos comprender mejor las necesidades que nos impulsaron al desarrollo de una nueva y mejorada arquitectura software.

3.1 Las matemáticas de la cinemática inversa

Comenzaremos nuestra excursión matemática clasificando primero los métodos que intentan resolver el problema de la cinemática inversa. Básicamente existen cuatro tipos:

- **Métodos geométricos**. Son aquellos que usan relaciones trigonométricas y geométricas para calcular las soluciones al problema de la cinemática inversa. Se clasifican en *directos* e *inversos*. Estos métodos suelen ser suficientes para robots con pocos grados de libertad, sin embargo no es un método genérico, ya que depende de la arquitectura del robot.

- Métodos que utilizan **matrices de transformación homogéneas** entre los diferentes sistemas de referencia con los que trabaja el robot.

- **Métodos de desacoplamiento cinemático**. Se usan en cierto tipo de robots con 6-DOF. Este método separa el problema en dos partes, posicionamiento del efector final en el punto deseado y orientación del efector final.

- **Soluciones numéricas iterativas**, parten de unos parámetros o mediciones iniciales, x_0, e iteran hasta encontrar nuevos valores, x_k, que converjan a la solución óptima del problema. Los cálculos que se realizan en cada iteración suelen ser productos de matrices y vectores.

- Otros métodos: como métodos de reducción polinómica, métodos iterativos apoyados por redes neuronales, algoritmos genéticos...

En este trabajo se ha elegido como método de resolución para la cinemática inversa el **algoritmo de Levenberg-Marquardt**. Este método se enmarca dentro de las soluciones numéricas iterativas, ya que se trata de un algoritmo iterativo utilizado para resolver problemas no lineales de mínimos cuadrados[10][11] en los que se busca una solución que disminuya una función de error (*damped least-squares solution*). En el caso de la cinemática, la función de error a minimizar será la distancia entre el punto objetivo y la posición del efector final robótico.
Siguiendo el esquema del TFG anterior, antes de explicar este algoritmo es necesario repasar una serie de conceptos matemáticos que serán la base del método de Levenberg-Marquardt. De esta forma lograremos entender de una forma más completa su funcionamiento.
Así pues, este método es una interpolación entre el **descenso de gradiente** y el **método de Gauss-Newton**[12].

3.1.1 Método del descenso de gradiente

Incluido entre los algoritmos generales de descenso, el método del descenso de gradiente (también llamado método del descenso más pronunciado o método de Cauchy), es un método de optimización muy usado para minimizar una *función diferenciable multivariante*[14].
Pero antes de estudiar el funcionamiento de este método, conviene repasar primero los conceptos de *campo escalar* y *campo vectorial*, además del cálculo del *gradiente de una función*.

Campo escalar y campo vectorial Imaginemos un conjunto abierto Ω en $\mathbb{R}^N$ y una función que opera sobre ese conjunto, $f : \Omega \to \mathbb{R}^M$. Con estos datos podemos

[14]Estas funciones se caracterizan por operar sobre más de una variable, $f(x_1, x_2, ..., x_n)$ y admitir, además, derivadas en cualquier dirección, pudiendo ser aproximadas al menos hasta primer orden por otra aplicación afín. Este concepto se entenderá mejor al estudiar el polinomio de Taylor.

definir los campos de la siguiente manera:

1. Cuando $M = 1$ y $N = 1$ se obtiene una **función de variable real**, $f(x)$, donde $x \in \Omega$, que recibe el nombre de variable independiente, y la y o $f(x)$ se llama variable dependiente o imagen.

2. Cuando $M = 1$ y $N > 1$ se obtiene un **campo escalar** o una función de N variables. Sería el caso de las funciones multivariables $f(x)$, donde $x = x_1, x_2, ..., x_N \in \Omega$, y en el que f asigna un valor real a cada vector de variables $x = x_1, x_2, ..., x_N$.

3. Cuando $M > 1$ y $N > 1$ se obtiene un **campo vectorial** M-dimensional en N variables. Así la función f se corresponde con $f = (f_1, f_2, ..., f_M)$ (donde cada f_i es un campo escalar en $\mathbb{R}^N$) y la variable $x \in \Omega$ será $x = x_1, x_2, ..., x_N$.

Cálculo del gradiente de una función Supongamos una función de variable real, $G(x)$, continua y derivable en el punto $x = a$. Entonces, la derivada de G en $x = a$, $G'(a)$, representará lo que varía la función en el entorno de ese punto.
Ahora supongamos un campo escalar $D(x_1, x_2, x_3)$, continuo y derivable en $x_1 = a$. Entonces, la derivada de D con respecto a x_1 en $x_1 = a$ indicará lo que varía el campo con respecto a x_1 en el entorno de $x_1 = a$. De igual manera, si derivamos D con respecto a x_2 en $x_2 = b$, obtendremos la variación del campo con respecto a x_2 en el entorno de $x_2 = b$. Generalizando este concepto, podemos decir que para campos escalares $D(x_1, x_2, ..., x_n)$, las derivadas de D con respecto a cada una de sus variables independientes $x_1, x_2, ..., x_n$, indicarán lo que varía el campo en el entorno de esas variables, y se llaman *derivadas parciales* de D: $\frac{\partial D(x_1, x_2, ..., x_n)}{\partial x_1}$, $\frac{\partial D(x_1, x_2, ..., x_n)}{\partial x_2}$, ..., $\frac{\partial D(x_1, x_2, ..., x_n)}{\partial x_n}$
Precisamente, para obtener el gradiente de un campo escalar se necesita calcular las derivadas parciales de éste[15], de tal manera que, para el campo escalar $D(x_1, x_2, ..., x_n)$,

[15] Para funciones reales $f(x)$ de una única variable, el gradiente de la función coincide con la derivada: $\nabla f(x) = \frac{\partial f(x)}{\partial x} = f'(x)$

su gradiente se calcula como:

$$\nabla D = \left(\frac{\partial D}{\partial x_1}, \frac{\partial D}{\partial x_2}, ..., \frac{\partial D}{\partial x_n} \right)^T \tag{2}$$

De esta forma, el gradiente será un vector n dimensional, compuesto por las n derivadas parciales de D (llamadas *componentes*), las cuales proporcionarán información acerca de la variación de D con respecto a ellas. Así, ∇D proporcionará, en cada punto del campo escalar donde esté definido, la dirección local en la que éste varía de una forma más pronunciada (la dirección de máximo crecimiento).

Método del descenso de gradiente Si ∇D nos indica la dirección de máximo crecimiento del campo D, $-\nabla D$ nos indicará la dirección de máximo decrecimiento. El método del descenso más pronunciado se valdrá de esta dirección para *descender* de forma iterativa por D, hasta localizar un punto con gradiente nulo, lo que implicaría haber encontrado un mínimo[16].

El funcionamiento es simple: el método parte de un punto inicial x^1 en $\mathbb{R}^n$ ($x^1 = [x_1^1, x_2^1, ..., x_n^1]$) del campo $H(x) = H(x = x_1, x_2, ..., x_n)$, y va iterando $(x^1, x^2, x^3, ..., x^m)$, seleccionando en cada iteración, la dirección de decrecimiento dada por $-\nabla H$, hasta que alcance un punto x_j donde $|\nabla H| = 0$

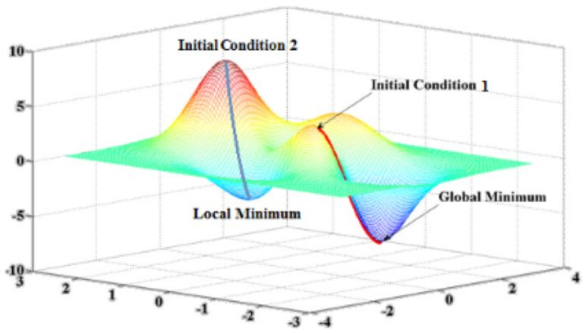


Figure 11: Método del descenso de gradiente

Una vez toomado un punto inicial x^1, para calcular los siguientes puntos

[16]El gradiente ∇D se hace cero en los puntos máximos, mínimos y de inflexión de la función $D(x)$

$x^2, x^3, ..., x^m$ a analizar en las siguientes iteraciones 2, 3, ..., m, el método del gradiente aplica la fórmula $x^{k+1} = x^k + \lambda^k \cdot s^k$, donde:

- x^k es el punto a evaluar en la iteración k y x^{k+1} es el punto a evaluar en la siguiente iteración, $k+1$.
- s^k es la dirección de búsqueda de máximo descenso, $s^k = -\nabla H(x^k)$
- λ^k es un escalar que determina la longitud de paso en la dirección s^k (el salto entre una iteración y otra)

Este método presenta un problema: aunque minimiza la función $H(x)$ satisfactoriamente en la primeras iteraciones, tiende a hacerse excesivamente lento en las últimas, alargando el tiempo de cómputo del algoritmo.

3.1.2 Método de Gauss-Newton

El método de Gauss-Newton es un algoritmo iterativo de minimización no lineal muy utilizado para resolver problemas de mínimos cuadrados no lineales. Pero, al igual que hemos hecho con el descenso más pronunciado, para comprender en profundidad cómo funciona este último algoritmo, es necesario estudiar los pilares en los que se sustenta: por una parte las *aproximaciones de Taylor* y, por otra, el *método de Newton-Raphson*.

Teorema de Taylor Este teorema permite aproximar una función $G(x)$, derivable n veces en el entorno de un punto $x = a$, $(a-\varepsilon, a+\varepsilon)$, que pertenece al dominio de la función, mediante un polinomio P, cuyos coeficientes dependen de las derivadas de la función en ese punto.
Pensemos en un punto $x = b$ cercano al punto $x = a$ tal que $b \in (a-\varepsilon, a+\varepsilon)$ (donde ε representa un margen pequeño), y donde la función $G(b)$ es medible y derivable n veces. En ese caso existe un polinomio P de orden n definido mediante la ecuación:

$$P_n(b) = G(a) + G'(a)(b-a) + \frac{G''(a)}{2!}(b-a)^2 + ... + \frac{G^n(a)}{n!}(b-a)^n \qquad (3)$$

Un caso muy importante, ya que se usará en el algoritmo de Gauss-Newton, es el **polinomio de Taylor de primer orden**, P_1. Siguiendo la ecuación 3, el polinomio $P_1(b)$ responde a la forma:

$$P_1(b) = G(a) + G'(a)(b-a) \tag{4}$$

La ecuación 4 demuestra que el polinomio de Taylor de primer grado de $G(x)$ es la recta tangente a dicha función en el punto $x = a$[17].

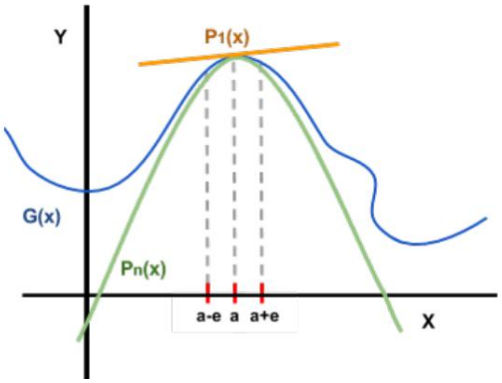


Figure 12: Aproximación del polinomio de Taylor de grado 1 (amarillo) y 2 (verde) a la función $G(x)$ en el punto a

Además, el polinomio P_m (con $m = 1,2,...,n$) cumple con $G(a) \simeq P_m(b)$ si $b \in (a-\varepsilon, a+\varepsilon)$. Esta propiedad implica que la solución de P_m en $x = b$ (dentro del entorno del punto a) se **aproxima** a la solución de la función G en $x = a$, de tal forma que:

$$G(a) = P_m(b) + R_{m+1}(b) \tag{5}$$

Esto significa que *el polinomio aproxima el valor de la función $G(x)$ en el entorno del punto $x = a$ más un error, $R_{m+1}(b)$.*

Tanto la ecuación de aproximación 5, como la ecuación del polinomio de primer grado

[17] Esto se puede comprobar mediante la ecuación principal de la recta $y = n + m \cdot x$. En nuestro caso, si expandimos $P_1(x) = G(a) + G'(a)(x-a)$ obtendremos:

$$P_1(x) = G(a) - G'(a) \cdot a + G'(a) \cdot x$$

Por lo que $n = G(a) - G'(a) \cdot a$ y $m = G'(a)$

4, serán de vital importancia en el desarrollo del método para la resolución de la cinemática inversa.

Otro caso interesante de estudio es el polinomio de Taylor aplicado no a funciones de variable real, sino a campos escalares del tipo $G(x_1, x_2, ..., x_n)$.

Pongamos un ejemplo sencillo: queremos aproximar la función $G(x_1, x_2)$, derivable n veces, en el entorno definido por la circunferencia esférica Θ, con centro en el punto $a = (a_1, a_2)$ (perteneciente al dominio de la función) y radio ε. Si existe un punto $b = (b_1, b_2)$ cercano al punto a tal que $b \in \Theta(a, \varepsilon)$, entonces existe un polinomio de grado n, P_n, que aproxima la función:

$$P_n(b_1, b_2) = G(a_1, a_2) + \frac{\partial G}{\partial b_1}(b_1 - a_1) + \frac{\partial G}{\partial b_2}(b_2 - a_2) + \frac{1}{2!} \cdot \left(\frac{\partial^2 G}{\partial b_1^2}(b_1 - a_1)^2 + \frac{\partial^2 G}{\partial b_2^2}(b_2 - a_2)^2 + 2\frac{\partial^2 G}{\partial b_1 \partial b_2}(b_1 - a_1)(b_2 - a_2) \right) + ... + \frac{1}{n!} \cdot \left(... \right)$$

Como podemos ver, en este caso, hay que acompañar cada variable con su derivada parcial. Tomemos solamente el polinomio de primer grado:

$$P_1(b_1, b_2) = G(a_1, a_2) + \frac{\partial G}{\partial b_1}(b_1 - a_1) + \frac{\partial G}{\partial b_2}(b_2 - a_2) \tag{6}$$

Esta fórmula demuestra que el polinomio $P_1(x_1, x_2)$ define el plano tangente a la función $G(x_1, x_2)$ en el punto $b(b_1, b_2)$. Además, la podemos abreviar como $P_1(b_1, b_2) = G(a_1, a_2) + \sum_{i=1}^{2} \frac{\partial G}{\partial b_i}(b_i - a_i)$, que es exactamente igual a $P_1(\vec{b}) = G(\vec{a}) + \nabla G \cdot (\vec{b} - \vec{a})$.

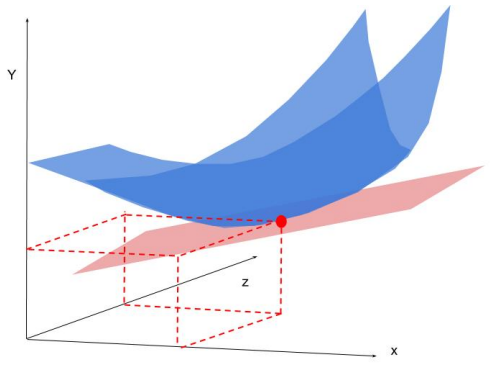


Figure 13: Aproximación del polinomio de Taylor de grado 1 (representado como un plano rosa) a la función G (marcada en azul) en el punto a (señalizado en rojo)

Para polinomios de grado n con campos escalares de dos o más variables, obtendríamos hiperplanos (planos de dimensión n) tangentes al campo original.

Método de Newton-Rapshon Es un algoritmo iterativo para optimización no lineal, basado en funciones $f(x)$, con $x = x_1, x_2, ..., x_n$, del tipo C^2[18] para las que existe un punto $x = x^*$ tal que $f(x^*) = 0$ y $f'(x^*) \neq 0$, es decir, el punto $x = x^*$ es una raíz de f (ver tabla 2). El problema al que se enfrenta Newton-Rapshon es precisamente cómo encontrar esa x^*. Para ello, el método parte de un punto inicial $x_0 = (x^* - h)$, que es justamente un desarrollo del polinomio de Taylor:

$$f(x^*) = f(x_0 + h) = f(x_0) + f'(x_0) \cdot h + \frac{f''(x_0)}{2!} \cdot h^2$$

Si $x^* - x_0 = h$ es lo mismo que $x_{i+1} - x_i = h$, podemos sustituir la h y expresar el anterior polinomio como:

$$f(x_{i+1}) = f(x_i) + f'(x_i) \cdot (x_{i+1} - x_i) + \frac{f''(x_i)}{2!} \cdot (x_{i+1} - x_i)^2$$

El siguiente paso sería igualar el polinomio anterior a cero para poder resolverlo, $f(x_{i+1}) = 0$. Sin embargo, este polinomio puede resultar muy difícil de solucionar, por lo que Newton-Raphson linealiza la ecuación:

$$f(x_{i+1}) = f(x_i) + f'(x_i) \cdot (x_{i+1} - x_i) + \cancel{\frac{f''(x_i)}{2!} \cdot (x_{i+1} - x_i)^2}$$

De esta forma, si despejamos x_{i+1} obtendremos **la fórmula de Newton-Raphson**

[18] Las funciones se clasifican en tres tipos:

- Clase C^1 o *funciones diferenciables continuas*: cuando las primeras derivadas parciales de la función son continuas.
- Clase C^n con $n \geq 1$ o *funciones diferenciables finitas*: cuando las derivadas parciales de orden n son continuas. En el caso de Newton-Raphson, una función de clase C^2 significa que sus derivadas parciales primeras y segundas existen y son continuas.
- Clase C^∞ o *función continuamente diferenciable*: cuando las derivadas parciales de orden n son continuas, para cualquier valor de $n \geq 1$. Es decir, existen todas las derivadas de todos los órdenes.

$$0 = f(x_i) + f'(x_i) \cdot (x_{i+1} - x_i)$$
$$0 = \frac{f(x_i)}{f'(x_i)} + x_{i+1} - x_i$$
$$x_i - \frac{f(x_i)}{f'(x_i)} = x_{i+1} \tag{7}$$

Podemos llegar a la misma ecuación 7 mediante el ejemplo gráfico propuesto en la imagen 14. En ella tenemos una función $f(x)$ en azul, y un punto de partida, x_0, marcado en rojo. El primer paso que realiza Newton-Raphson es trazar la recta tangente a la función en el punto x_0 con pendiente $f'(x_0)$. Esta recta responde a la ecuación $y - f(x_0) = f'(x_0) \cdot (x - x_0)$[19].

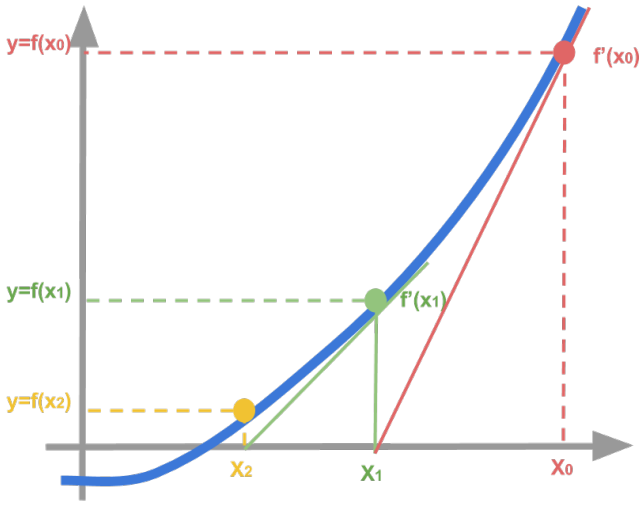


Figure 14: Funcionamiento del método Newton-Raphson en el plano.

El siguiente paso será encontrar el punto x_1. Para ello iguala $y = 0$ y despeja la $x = x_1$:

$$x_0 - \frac{f(x_0)}{f'(x_0)} = x_1$$

[19]Si nos damos cuenta, esta expresión es idéntica a la ecuación principal de la recta, $y = n + m \cdot x$, donde (x, y) es el punto por donde pasa la recta, n es el punto de intersección en la ordenada (eje Y) y m la pendiente de la recta. En nuestro caso:

$$y - f(x_0) = f'(x_0) \cdot (x - x_0)$$
$$y = f(x_0) + f'(x_0) \cdot (x - x_0)$$
$$y = f(x_0) + f'(x_0) \cdot x - f'(x_0) \cdot x_0$$
$$y = (f(x_0) - f'(x_0) \cdot x_0) + f'(x_0) \cdot x$$

Por lo que $n = f(x_0) - f'(x_0) \cdot x_0$ y $m = f'(x_0)$.

De esta forma obtendríamos el siguiente punto, x_1, al que volveríamos a calcular la recta tangente en $f(x_1)$ con pendiente $f'(x_1)$, para calcular el punto x_2 de la siguiente iteración. Aí pues, si generalizamos la fórmula anterior, extraeremos otra vez la ecuación de Newton-Raphson 7

Sin embargo, para que la función f converja en $x_{i+1} = x^*$ necesitamos que la ecuación de Newton-Raphson sea continua en el entorno de x^* y derivable:

$$\left(x_i - \frac{f(x_i)}{f'(x_i)}\right)' = \frac{f''(x_i) \cdot f(x_i)}{f'(x_i)} < 1$$

Éste algoritmo presenta un problema: depende en exceso del punto x_0 inicial y tiende a atascarse en mínimos locales, además de que posee una convergencia cuadrática cuando ésta se da. Pero será el pilar de nuestro último método básico, el **método Gauss-Newton**.

Método Gauss-Newton Como avanzábamos al comienzo de esta sección, el método de Gauss-Newton es un algoritmo iterativo de minimización no lineal, al igual que Newton-Raphson. En este caso, el algoritmo se utiliza para la resolución de problemas de mínimos cuadrados no lineales en los que el objetivo es encontrar el mínimo de una función no lineal, $S(x)$, que es la suma de los cuadrados de otras funciones:

$$S(x) = \sum_{i=1}^{m} f_i^2(x) \tag{8}$$

Donde $f_1, f_2, ..., f_m$ son funciones no lineales de varias variables, $x = x_1, x_2, ..., x_n$. Si las agrupamos en forma de vector:

$$F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ ... \\ f_m(x) \end{pmatrix}$$

podremos representar el problema de mínimos cuadrados de la ecuación 8 como $S(x) = F^T(x) \cdot F(x)$. De esta manera, lo que Gauss-Newton busca es encontrar la secuencia de valores de $x^* = [x_1^*, x_2^*, ..., x_n^*]$ que cumpla con $S(x^*) = 0 \rightarrow F^T(x^*) \cdot F(x^*) = 0$.

Para converger en esa solución, Gauss-Newton utiliza el método del gradiente para descender por S, hasta encontrar un punto donde el gradiente sea nulo, lo que indica que ha alcanzado un mínimo, $|\nabla S(x_i)| = 0$, o hasta que alcance el umbral de convergencia o error aceptable deseado, $|\nabla S(x_i)| < e$. El gradiente de descenso de S se calcula como:

$$\nabla S = \sum_{j=1}^{n} \sum_{i=1}^{m} 2 \cdot f_i \cdot \frac{\partial f_i}{\partial x_j}$$

Teniendo en cuenta que la matriz *Jacobiana*, J, responde a la forma:

$$J(x) = \begin{pmatrix} \frac{\partial f_1(x)}{x_1} & \frac{\partial f_1(x)}{x_2} & \dots & \frac{\partial f_1(x)}{x_n} \\ \frac{\partial f_2(x)}{x_1} & \frac{\partial f_2(x)}{x_2} & \dots & \frac{\partial f_2(x)}{x_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_m(x)}{x_1} & \frac{\partial f_m(x)}{x_2} & \dots & \frac{\partial f_m(x)}{x_n} \end{pmatrix} = \begin{pmatrix} (\nabla f_1)^T \\ (\nabla f_2)^T \\ \dots \\ (\nabla f_m)^T \end{pmatrix}$$

podemos reducir la ecuación del gradiente a $\nabla S = 2 \cdot J^T(x) \cdot F(x)$. Si lo que buscamos es descender, entonces utilizaremos $-\nabla S = -2 \cdot J^T(x) \cdot F(x)$, incluso podemos eliminar la constante 2, simplificando la expresión a $-\nabla S = -J^T(x) \cdot F(x)$. En el fondo, estamos utilizando la información que nos devuelve la primera derivada de cualquier función:

Derivada	Significado
$F'(c) \rightarrow (+-)$	Existe un máximo relativo en $x = c$
$F'(c) \rightarrow (-+)$	Existe un mínimo relativo en $x = c$
$F'(c) \rightarrow (--)$	$x = c$ no es un mínimo ni un máximo
$F'(c) \rightarrow (++)$	$x = c$ no es un mínimo ni un máximo

Table 2: Información de la primera derivada de una función.

Pero esta información no es suficiente. Es necesario utilizar la información que nos ofrece la segunda derivada, sin embargo (y al contrario que el método original de Newton) Gauss-Newton hace una aproximación a la matriz de segundas derivadas parciales conocida como *Hessiano*, H. Mientras que el método original de Newton calcula el Hessiano como:

$$H(x) = J^T(x) \cdot J(x) \cdot \sum_{i=1}^{m} f_i(x) \cdot \nabla^2 f_i(x)$$

Gauss-Newton la aproxima como:

$$H(x) \approx \sum_{i=1}^{m} \frac{\partial f_i(x)}{\partial x_j} \cdot \frac{\partial f_i(x)}{\partial x_k} = J^T(x) \cdot J(x) \tag{9}$$

Resulta una expresión bastante más sencilla. Y con la información que da la segunda derivada, podemos razonar que $J^T(x) \cdot J(x) > 0$.

Derivada	Significado
$F''(c) < 0$	Existe un máximo relativo en $x = c$
$F''(c) > 0$	Existe un mínimo relativo en $x = c$
$F''(c) = 0$	Falla el criterio en $x = c$. Puede ser un punto de inflexión

Table 3: Información de la segunda derivada de una función.

Por otro lado, Gauss-Newton se basa en la aproximación lineal de la función $S(x)$ mediante un desarrollo de Taylor de primer orden (ecuación 5). De esta forma, y partiendo de $S(x)$ en un punto $x = a \in Dom(S)$, Gauss-Newton considera que para pequeñas perturbaciones $\pm\Delta a$, es posible obtener una serie de Taylor tal que $S(a \pm \Delta a) = S(a) + e$, donde e es el error generado por la aproximación.
Así pues juntando estas expresiones, Gauss-Newton utiliza $x^{k+1} = x^k + \Delta x$ como fórmula para ir iterando sobre diversos valores del vector de variables x, hasta encontrar uno que anule la función S, $S(x*) = 0$:

$$x^{k+1} = x^k - (J^T(x^k) \cdot J(x^k))^{-1} \cdot J^T(x^k) \cdot F(x^k)$$

Podemos comprobar que el incremento de cada iteración Δx es igual a $(J^T(x^k) \cdot J(x^k))^{-1} \cdot J^T(x^k) \cdot F(x^k)$, así que para resolver la ecuación sólo tenemos que despejar:

$$x^k + \Delta x = x^k - (J^T(x^k) \cdot J(x^k))^{-1} \cdot J^T(x^k) \cdot F(x^k)$$
$$\Delta x = \cancel{-x^k + x^k} - (J^T(x^k) \cdot J(x^k))^{-1} \cdot J^T(x^k) \cdot F(x^k)$$
$$((J^T(x^k) \cdot J(x^k))^{-1})^{-1} \cdot \Delta x = -J^T(x^k) \cdot F(x^k)$$

$$J^T(x^k) \cdot J(x^k) \cdot \Delta x = -J^T(x^k) \cdot F(x^k) \tag{10}$$

Finalmente obtenemos la ecuación normal 10 que permite calcular el incremento óptimo que se debe sumar en cada iteración del algoritmo de Gauss-Newton, hasta que éste alcanza el mínimo de la función S.

3.1.3 Método de Levenberg-Marquardt

Una vez repasados el algoritmo del descenso de gradiente y el método de Gauss-Newton, estamos en disposición de estudiar el **algoritmo iterativo de Levenberg-Marquardt**. Éste se comporta como un método de descenso rápido cuando la solución actual está lejos de un mínimo. Sin embargo, cuando está cerca de un mínimo local, se comporta como el método de Gauss-Newton. Así, consigue converger rápidamente y encontrar una solución a pesar de estar lejos del mínimo final. Pero, al igual que los anteriores métodos de optimización no lineal, la solución final alcanzada por Levenberg-Marquardt depende demasiado de las condiciones iniciales de ejecución, pudiendo quedar atrapado en mínimos locales[13].
Este método parte de una función objetivo $F(x) = [f_1(x), f_2(x), ..., f_m(x)]^T$, con $x = [x_1, x_2, ...x_n]$. Siguiendo la aproximación de Taylor, Levenberg-Marquardt toma como expresión base la siguiente ecuación:

$$F(x + \Delta x) \simeq F(x) + J \cdot \Delta x \tag{11}$$

Si observamos bien, comprobaremos la correlación existente entre las ecuaciones 6 y 11. Por otro lado, el método necesita una función de error cuya expresión se define

como $\varepsilon = P_{target} - F(x)$, donde P_{target} es un resultado objetivo (por ejemplo un punto en el espacio que debe ser alcanzado por un brazo robótico). Al sustituir en la expresión 11 obtendremos: $P_{target} - F(x) - J \cdot \Delta x = \varepsilon - J \cdot \Delta x = J \cdot \Delta x - \varepsilon = 0$.
Para minimizar el error, $J \cdot \Delta x - \varepsilon$ debe ser ortogonal a la columna espacio de la matriz J. En otras palabras, su producto escalar debe ser $0 \rightarrow J^T \cdot (J \cdot \Delta x - \varepsilon) = 0$. De esta expresión resolveremos el cálculo de los Δx que se aplican en cada iteración del método de Levenberg-Marquardt:

$$J^T \cdot (J \cdot \Delta x - \varepsilon) = 0$$
$$J^T \cdot J \cdot \Delta x = J^T \cdot \varepsilon$$
$$\Delta x = (J^T \cdot J)^{-1} \cdot J^T \cdot \varepsilon$$

Esta resulta ser una expresión similar a la ecuación normal de Gauss-Newton, 10, donde $J^T \cdot J = H$ es la aproximación a la matriz Hessiana, cuadrada e invertible si tiene rango completo, y $g(x) = J^T \cdot \varepsilon$ es la dirección de descenso de la función, el gradiente de descenso.
Por último, Levenberg-Marquardt añade un ”amortiguador” μ que modifica el comportamiento del algoritmo:

- Cuando μ es próximo o igual a cero, Levenberg-Marquardt se comporta exactamente como el método de Gauss-Newton.
- Cuando μ es alto, Levenberg-Marquardt se comporta como el método de descenso de gradiente, diferenciándose en el paso, que es ligeramente más pequeño.

$$\Delta x = (H + \mu \cdot I)^{-1} \cdot g(x) \tag{12}$$

El método de Gauss-Newton resulta más rápido y más exacto a la hora de minimizar la función de error, por lo que en cada iteración en la que se va descendiendo de forma exitosa, μ disminuye su valor acercando el comportamiento de Gauss-Newton.

Implementación del algoritmo de Levenberg-Marquardt Para la implementación del actual sistema de Cinemática Inversa se ha tomado como

referencia el método propuesto por los profesores griegos Manolis Lourakis y Antonis Argyros[14], al que se le han introducido algunas variaciones para adaptarlo a C++[8].

Algorithm 1 Levenberg-Marquardt algorithm

procedure LEVENBERG MARQUARDT(F, P, x^0)
  $\varepsilon_1,\ \varepsilon_2,\ \varepsilon_3,\ \varepsilon_4,\ k_{max},\ \tau$
  $k := 0;\ \ \nu := 2\ \ x := x^0;$
  $H := J^T \cdot J;\ \ e_x := P - F(x)\ \ g := J^T \cdot e_x;$
  $stop := (\| g \|_\infty \leq \varepsilon_1);\ \ \mu := \tau \cdot max_{i=1,...,m}(H_{ii});$
  while (**not** $stop$) **and** $k < k_{max}$ **do**
    $k := k + 1$
    do
      SOLVE $(H + \mu \cdot I) \cdot \Delta x = g$
      if $\| \Delta x \| \leq \varepsilon_2 \cdot (\| x \| + \varepsilon_2)$ **then**
        $stop := True;$
      else
        $x_{new} := x + \Delta x;$
        $\rho := (\| e_x \|^2 - \| P - F(x_{new}) \|^2);$
        if $\rho > 0$ **then**
          $stop := (\| e_x \| - \| P - F(x_{new}) \|) < \varepsilon_4 \cdot \| e_x \|;$
          $x = x_{new}$
          $H := J^t \cdot J;\ \ e_x := P - F(x)\ \ g := J^t \cdot e_x;$
          $stop := (stop$ **or** $\| g \|_\infty \leq \varepsilon_1);$
          $\mu := \mu \cdot max(\frac{1}{3}, 1 - (2 \cdot \rho - 1)^3);\ \ \nu := 2$
        else
          $\mu := \mu \cdot \nu;\ \ \nu := 2 \cdot \nu$
        end if
      end if
    while $(\rho > 0)$ **or** $(stop)$
    $stop := (\| e_x \| \leq \varepsilon_3);$
  end while
  $x^{final} := x;$
end procedure

Como podemos observar, el algoritmo recibe como parámetros de entrada:

- F: función vectorial de cinemática directa $F : R^m \leftarrow R^n$ con $n \geq m$.
- P: vector de coordenadas objetivo (*pose*) $P = [x_1, x_2, ..., x_i] \in R^i$.

- $x^0 \in R^n$: un vector de ángulos iniciales $x^0 = [x_1^0, x_2^0, ..., x_n^0]$ correspondientes a la posición actual del efector robótico.

Retornando como salida x^{final}, un vector con los nuevos valores angulares de cada joint de la estructura que minimiza la función de error.
Además, el algoritmo de Levenberg-Marquardt propuesto depende de una serie de variables y constantes, tales como e_x, que es el vector de error, ρ que es la relación de ganancia que indica si el algoritmo se acerca a un mínimo o no, μ que es el factor de amortiguación, g que es el gradiente de descenso y los distintos ε y τ que son umbrales de cuyos valores dependerá la solución obtenida[20]:

1. ε_1: es un umbral para medir la magnitud del gradiente de descenso, g. Se asegura de que el gradiente vaya descendiendo correctamente.

2. ε_2: es un umbral de aceptación del incremento Δx. Cuando los incrementos son muy pequeños (del orden de milésimas) significa que el método de Levenberg-Marquardt ha convergido en un mínimo, por lo que no tiene sentido seguir iterando.

3. ε_3: es el umbral del error e_x. Esto permite configurar el algoritmo de Levenberg-Marquardt para encontrar distintas soluciones de mayor o menor precisión.

4. ε_4: es un umbral para controlar la reducción del error después de aplicar un Δx.

Por último, I es la matriz identidad cuadrada de la misma dimensión que el hessiano. Cuando el algoritmo no encuentra unos incrementos que mejoren el error, la μ

[20]Como se verá más adelante, esta forma de implementar el algoritmo es muy dependiente de los valores asignados a los umbrales, de tal forma que dado un mismo problema, con la misma situación de partida, el mismo objetivo y el mismo número de iteraciones, las soluciones pueden ser mejores o peores dependiendo del valor de los umbrales. En verdad son opciones de minimización del error cuando el algoritmo calcula una solución para el problema de cinemática inversa

aumenta, dando más peso a la diagonal del hessiano:

$$\begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} + \begin{pmatrix} \mu & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{pmatrix} = \begin{pmatrix} h_{11}+\mu & h_{12} & h_{13} \\ h_{21} & h_{22}+\mu & h_{23} \\ h_{31} & h_{32} & h_{33}+\mu \end{pmatrix}$$

Para terminar, esta implementación aplica dos controles:

1. $if(\| \Delta x \| \leq \varepsilon_2 \cdot (\| x \| + \varepsilon_2)$: este control evita que el método se estanque cuando los incrementos se vuelven demasiado pequeños, indicando que el método ha convergido y ahorrando tiempo de cómputo.

2. $if \rho > 0$: este control es lo mismo que decir $e_{old} > e_{new}$, indicando entonces si el método está descendiendo correctamente por la función de error, caso en el que actualiza los valores angulares con el incremento calculado Δx, o si no está descendiendo ($e_{old} < e_{new}$), caso en el que no se aplica el incremento calculado Δx al vector de valores angulares, por el contrario se aumenta el factor de amortiguación μ.

3.2 La cinemática inversa de Shelly

Superado el mal trago de las matemáticas, nos podemos preguntar *"y todo esto... ¿con qué fin? ¿Para qué robot? ¿Cómo se utiliza?..."*. El objetivo último del nuevo sistema de cinemática inversa es ser utilizado para resolver los problemas de manipulación de robots reales. Más concretamente, de brazos robóticos reales, de bajo coste, desarrollados en el Laboratorio de Robótica y Visión Artificial de la Universidad de Extremadura, **Robolab**, dotándolos de la capacidad de alcanzar objetivos no prefijados y manipular objetos cotidianos del entorno. Al final, el planteamiento que subyace en este proyecto es el diseño y producción de robots sociales de bajo coste pero que mantengan una gran calidad funcional. Esto se traduce en que, a pesar de utilizar motores menos precisos y con más holguras, las prestaciones no decaen gracias a un software robusto y de calidad, capaz de calibrar el sistema físico y corregir esas

deficiencias en cuanto al hardware se refiere.

En particular, el sistema de cinemática inversa diseñado e implementado durante este trabajo se validará sobre el robot social de bajo coste diseñado y desarrollado por Robolab, **Shelly**[21] (antiguamente conocido como *Ursus*[15]).

Las primeras versiones de este robot (imagen 15) fueron implementadas para la asistencia a personas con una cierta discapacidad[16]. En concreto para la rehabilitación de niños con trastornos motores debido a la parálisis cerebral hemipléjica y obstétrica del plexo braquial[17].

El objetivo de estas primeras versiones era facilitar las labores de rehabilitación de estos pacientes mediante la sucesión de ejercicios diseñados para mover los brazos: el robot se encargaba de monitorizar los movimientos del paciente (básicamente, mover los brazos a ciertas posiciones, con ciertas alturas y de cierta manera) y de guiarlo en los ejercicios, bajo la supervisión del personal médico, a la vez que calculaba una serie de datos y estadísticas clínicas con respecto al ejercicio, suponiendo un gran incentivo para el paciente y un soporte muy útil para el personal médico, que podían utilizar los datos generados para ajustar los ejercicios y mejorar el proceso de rehabilitación.

Figure 15: Antiguas versiones de Ursus: Ursus 1 y Ursus 2

En cuanto a la estructura de estas primeras versiones, seguían la presima del bajo coste y se componían de un torso estático y dos brazos de cinco grados de libertad

[21] Actualmente se encuentra en fase de mantenimiento y actualización. Se han eliminado los dos brazos antiguos, sustituyéndolos por un brazo de mayor precisión, con menos DOF y sin holguras, lo que ha cambiado irremediablemente el sistema cinemático. Para este trabajo se tendrá en cuenta sólo la configuración antigua de los dos brazos.

(DOF) cada uno (tres grados de libertad en el hombro y dos en el codo). Además, disponían de un cuello de tres DOF para mover la cabeza y una boca articulada para dotar al robot de una mayor expresividad. Por otra parte, la estructura incluía una cámara PrimeSense RGB-D para seguir el movimiento de los niños. Por último, para hacerlo más atractivo a los jóvenes pacientes, la estructura estaba recubierta por la piel de un oso de peluche.

En cuanto a la cinemática de estas primeras versiones, los robots contaban con un modelo muy simple en el que la cinemática era directa: la configuración del brazo robótico para ejecutar los ejercicios y las posiciones eran introducidos directamente en el sistema. Eran simplemente valores previamente almacenados y enviados directamente a los motores. *No hay ningún módulo que calcule la cinemática inversa del robot para alcanzar las posiciones requeridas en cada ejercicio.*

La nueva versión del robot Shelly difiere de las anteriores en que ya no está pensado como asistente de rehabilitación, sino como asistente doméstico. Su funcionalidad va mucho más allá de reconocer al paciente, monitorizarlo y realizar cálculos internos para obtener los datos clínicos. El nuevo robot debe ser capaz de interactuar con todo tipo de personas y también con diversos objetos cotidianos, presentes en un entorno tan variable como es el doméstico.

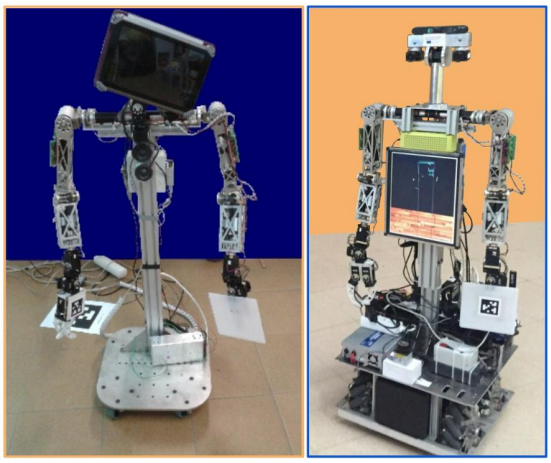


Figure 16: Ursus 3.0 y 3.1, ahora rebautizado como Shelly

Es justamente por la gran demanda de acciones que implica ser un robot doméstico por lo que Shelly debe aumentar su complejidad, tanto en su arquitectura hardware como software. A pesar de los múltiples avances en cuanto al software se refiere[22], este trabajo se centrará en la tarea cotidiana de coger una taza con el efector final del brazo robótico del robot en un tiempo razonable y con un error mínimo.

3.2.1 Configuración cinemática de Shelly

Al igual que en las anteriores versiones, el hardware del robot Shelly ha sido diseñado y desarrollado completamente en el laboratorio Robolab[19]. Con el objetivo de ser un robot social y autónomo, de propósito general y asequible para implantarlo en cualquier ambiente doméstico, se han añadido a la arquitectura sensores propioceptivos[23], sensores relacionados con la percepción visual (cámaras RGB-D) y sensores para la navegación autónoma (láser Hokuyo y cámara Asus Xtion Pro). Además, en cuanto a cinemática se refiere, Shelly dispone de dos brazos:

- El brazo derecho: tiene 9-DOF, correspondientes a cinco motores Faulhaber y dos motores Dynamixel que forman el brazo, más otros dos motores Dynamixel que forman las pinzas del efector final del robot. El sistema de cinemática inversa desarrollado durante este proyecto se probará sobre este brazo.

- El brazo izquierdo: dispone de 5-DOF, que se corresponden a los cinco motores Faulhaber del brazo. Este brazo dispone de una bandeja a modo de efector final.

Shelly está equipado con una pantalla sensible al tacto colocada en el pecho y una base omnidireccional con cuatro ruedas Mecanum de 200mm, capaz de soportar 100Kg de peso y que puede trabajar en modo direccional mediante configuración software. La plataforma de la base tiene dos pisos:

[22] Durante el desarrollo de este proyecto, se han desarrollado varios módulos software que añaden y mejoran considerablemente la autonomía de este robot doméstico, como es por ejemplo el sistema de navegación[18].

[23] Un sensor propioceptivo es aquel que sirve para medir el estado interno del robot. Indican al robot cuando es tiempo de recargar sus baterías, cuando un motor se está sobrecalentando o cuando un componente no funciona correctamente. Dentro de este tipo de sensores se encuentran los sensores que dan información de la posición y orientación del robot

- El primer piso contiene cuatro baterías de 24V/20Ah, un cargador, un inversor de 600W y la electrónica para cambiar entre los modos de carga y de batería sin interrupción. Gracias a esto el robot puede funcionar de forma autónoma por períodos de hasta ocho horas, dependiendo de la cantidad de movimiento y la carga de la CPU.

- En el segundo piso hay un pequeño clúster formado por NUCs i5 e i7 de Intel. Están conectados a un switch de 1Gb, que proporciona suficiente energía al clúster de forma eficiente. Dentro de esos NUCs está almacenado todo el sistema software del robot y es allí donde se despliegan los componentes software.

Por último, Shelly cuenta con un cuello 4-DOF, formado por cuatro motores Dynamixel (pan-cuello, tilt, pan-left, pan-right).

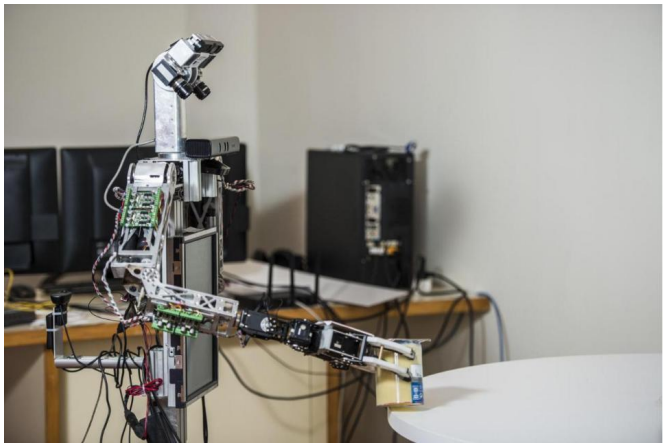


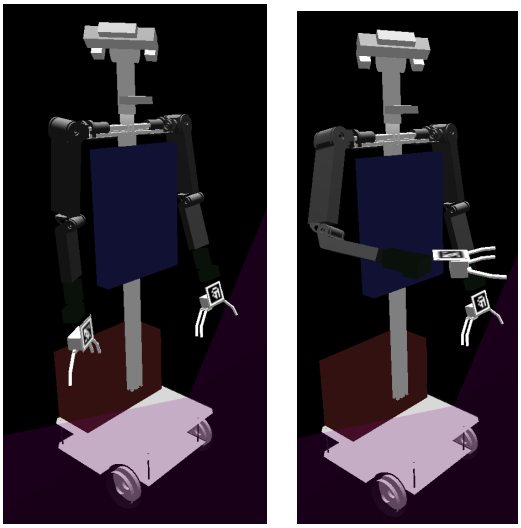
Figure 17: En esta imagen se puede apreciar el brazo derecho del robot Shelly de 9-DOF. También se aprecian las dos cámaras RGBD y la pantalla táctil. *Fotografía de Esteban Martinena*

3.2.2 Antiguo software cinemático de Shelly

El software que sustenta el sistema cinemáticp del robot ha evolucionado en el tiempo. Como dijimos, para las versiones 1 y 2 de Ursus, el software que controlaba el movimiento de sus brazos era muy primitivo, tanto es así que no existía verdaderamente cinemática inversa, sino un conjunto finito de posiciones cuyos valores angulares correspondientes estaban almacenados. Las posiciones alcanzables por el

robot estaban preestablecidas de antemano, de tal modo que si el robot debía mover su efector final a la posición P_B desde la posición P_A, sólo tenía que leer desde fichero los valores angulares de los joints correspondientes para esa posición. En definitiva, lo que Ursus 1 y Ursus 2 utilizaban era pura cinemática directa.

Para la versión 3.1 del robot se diseñó el primer sistema de cinemática inversa, basado en el algoritmo descrito de Levenberg-Marquardt (seccion 3.1.3). La naturaleza del problema cinematico derivó en un sistema bastante complejo y difícil de mantener, aunque su funcionamiento era correcto y bastante preciso en simulación.



(a) Posición de descanso (b) Target

Figure 18: Experimento del robot Shelly simulado con el antiguo sistema de cinemática inversa. El target se sitúa en $(t_x = 200\ mm,\ t_y = 950\ mm,\ t_z = 400\ mm,\ r_x = 0\ rad,\ r_y = -0.78\ rad,\ r_z = 0\ rad)$. Se alcanza con un error de $0.012\ mm$ en traslación y $0.2\ rad$ en rotación

Ahondando en el software de cinemática primitivo El primer software desarrollado para dar soporte a la cinemática inversa del robot se componía de dos módulos o componentes[8]

1. **inverseKinematicsTesterComp**: se trataba de un componente de validación del algoritmo cinemático. Proporcionaba una interfaz de usuario desde la que se configuraban las poses objetivo o targets que el robot debía alcanzar con su

efector final.

Estos targets se caracterizan por estar en el espacio 6D de tres traslaciones más tres rotaciones $[t_x, t_y, t_z, r_x, r_y, r_z]$ y por una matriz de pesos cuya finalidad es dar mayor o menor importancia a cada componente de traslación y de rotación del target. Además, los targets se podían clasificar en tres tipos:

- **POSE6D**: es el típico target a resolver. Viene definido por un vector 6D donde se recogen las coordenadas de traslación y las orientaciones en los tres ejes XYZ de la pose, $[t_x, t_y, t_z, r_x, r_y, r_z]$. El target es enviado al componente de cinemática para que una parte del robot lo resuelva, es decir, para que el efector final de una parte del robot se posicione y se oriente en las coordenadas que indique el target.
- **ALIGNAXIS**: este tipo de target viene definido por un vector 6D, en el que las componentes de traslación están anuladas (ya que no son necesarias) y sólo están marcadas las componentes de rotación de los ejes XYZ, $[\_, \_, \_, r_x, r_y, r_z]$, y por un vector binario de tres componentes, $[x, y, z]$ que representan los tres ejes de coordenadas XYZ, e indica el eje con el que se debe alinear el efector final.
- **ADVANCEAXIS**: este target viene definido por un vector dirección de tres componentes $[x, y, z]$ que representan los tres ejes de coordenadas XYZ, y un real d que indica la distancia que debe avanzar el efector final de una parte del robot a lo largo del vector dirección.

Estos tres tipos de targets aumentaban en gran medida la precisión del movimiento del efector final de Shelly, aunque añadían también complejidad al módulo de cinemática. En especial a la hora de calcular la función de error $e = P_{target} - F(x)$ del algoritmo de Levenberg-Marquardt.

2. **inverseKinematicsComp**: era el componente principal. Codificaba el algoritmo de resolución de la cinemática inversa y las estructuras necesarias para llevarlo a cabo. En su interfaz, declaraba e implementaba los métodos para

recibir los distintos tipos de targets, resolverlos mediante el algoritmo de Levenberg-Marquardt propuesto por Manolis Lourakis y Antonis Argyros[14], y mover el brazo robótico a la posición deseada aplicando los ángulos obtenidos.

Este último componente es sin lugar a dudas el pilar de la cinemática inversa, ya que concentraba toda la complejidad del software. Además, gran parte de su estructura y funcionamiento se han mantenido en esta nueva versión del software cinemático, por lo que resulta más que conveniente repasar ligeramente su estado inicial.

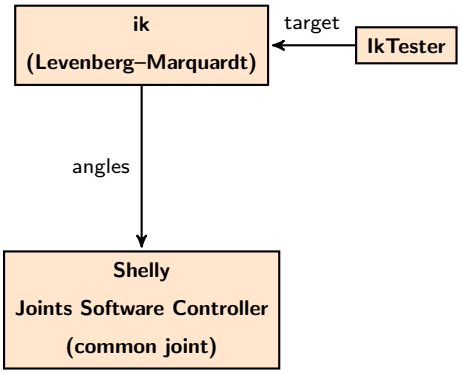


Figure 19: Estructura inicial del sistema de Cinemática Inversa. En él se observa el componente de validación que envía targets al componente base de la cinemática, para que éste último los resuelva. Los ángulos obtenidos mediante el algoritmo de Levenberg-Marquardt son enviados al componente de control de las articulaciones del robot.

Por dentro, el componente **inverseKinematicsComp** era una compleja máquina de estados a la espera de recibir targets para cualquier parte del robot (cabeza[24], brazo derecho[25] o brazo izquierdo[26]) a través de los métodos de entrada de su interfaz:

1. **setTargetPose6D**: recibía targets del tipo *POSE6D*, para ser resueltos mediante Levenberg-Marquardt de una forma clásica.

[24]Codificada como **HEAD**, cuya cadena cinemática, está formada por los motores *head_yaw_joint* y *head_pitch_joint*, siendo el efector final o *tip*, el *rgbd_transform*

[25]Codificado como **RIGHTARM**. La cadena cinemática se compone de siete motores: *rightShoulder1*, *rightShoulder2*, *rightShoulder3*, *rightElbow*, *rightForeArm*, *rightWrist1* y *rightWrist2*. El efector final de esta parte del robot se llama *grabPositionHandR*

[26]Codificado como **LEFTARM**. La cadena cinemática de esta parte del robot se compone de cinco motores: *leftShoulder1*, *leftShoulder2*, *leftShoulder3*, *leftElbow*, *leftForeArm*. El efector final será *grabPositionHandL*

2. **pointAxisTowardsTargets**: que recibía targets del tipo *ALINGAXIS*, para que el efector final del robot se alineara con el eje que el target indique. Como hemos visto, se trata de un target de rotación, en el que no importa la traslación del efector, sólo que esté rotado de la misma forma que el target.

3. **AdvanceAlongAxis**: este método recibía targets del tipo *ADVANCEAXIS*, cuyo objetivo hemos visto que es mover el efector final robótico una determinada distancia L a lo largo de una dirección marcada por un vector v_{dir}. Se trata de un método muy útil de aproximación al target.

Por otra parte, para resolver cada target implementaba el algoritmo de Levenberg-Marquardt siguiendo el esquema propuesto por Lourakis y Argyros [14], pero ampliando su estructura para contemplar no sólo los tres tipos de targets sino otros casos adversos para la cinemática.

Primera ampliación: Matriz de pesos W_e Repasemos la ecuación principal del algoritmo de Levenberg-Marquardt: $\Delta x = (H + \mu \cdot I)^{-1} \cdot g \rightarrow \Delta x = (H + \mu \cdot I)^{-1} \cdot J^t \cdot e$. De esta expresión se deduce que el vector de error calculado, e, influye mucho en el cálculo del algoritmo. Para solucionar esta dependencia se introdujo el concepto de matriz de peso W_e[20] tal que $\Delta x = (H + \mu \cdot I)^{-1} \cdot J^t \cdot (W_e \cdot e)$.
W_e es una matriz diagonal cuyos valores están comprendidos entre 0 y 1. Esta matriz por un lado absorbe las diferencias métricas entre las traslaciones y las rotaciones, y por otro, pondera la importancia de cada restricción de traslación y de rotación.

$$W_e \cdot e = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} e_{tx} \\ e_{ty} \\ e_{tz} \\ e_{rx} \\ e_{ry} \\ e_{rz} \end{pmatrix} = \begin{pmatrix} e_{tx} \\ e_{ty} \\ e_{tz} \\ 0 \\ e_{ry} \\ e_{rz} \end{pmatrix}$$

Segunda ampliación: Bloqueo de motores En el mundo real los joints representan motores reales de un robot, que tienen un rango, definido por unos ángulos máximo y mínimo, dentro del cual pueden girar. Dicho de otra forma: *los motores presentan límites*[21]. Sin embargo, el algoritmo de Levenberg-Marquardt calcula unos incrementos que siempre son añadidos a los ángulos originales de los joints, $\Delta x = (H + \mu \cdot I)^{-1} \cdot J^t \cdot (W_e \cdot e)$ y $x_{new} = x + \Delta x$, pudiendo darse el caso de que los incrementos superen los límites de un motor.

Para evitar esta situación, el algoritmo implementado añade un bucle interno que comprueba si algún ángulo calculado supera los límites del joint que le corresponda y, de darse el caso, bloquea ese joint y lo elimina de los cálculos[22]. Para ello se emplea un vector de motores bloqueados, L_b, donde $L_b[i] = 0$ significa que el joint i está libre y $L_b[i] = 1$ que está bloqueado (con $i = 0, ..., n_{joints}$).

Tercera ampliación: Distinción entre joints prismáticos y no prismáticos Este ampliación suponía una pequeña modificación en el cálculo de la matriz jacobiana[22] utilizada en el algoritmo, ya que la jacobiana guarda de alguna manera las relaciones que existen entre las velocidades de movimiento de las articulaciones y las velocidades de movimiento del extremo del robot:

1. Para joints no prismáticos[27] los elementos de traslación se calculan como $t_i = z_i - 1$, donde la posición relativa del joint i, q_i, se calcula con respecto al eje en el que se traslada el link anterior, z_{i-1}. Los elementos de rotación del joint i se calculan como el producto vectorial $r_i = -(z_{i-1} X t_{i-1})$.

2. Si el joint es prismático[28] los elementos de traslación siempre se anulan, $t_i = 0$, mientras que los elementos de rotación se calculan como $r_i = z_{i-1}$.

[27] Aquellos cuyos movimientos implican una rotación o una rotación más traslación
[28] Aquellos cuyo movimiento es de traslación

3.2.3 Problemática del antiguo software cinemático

Como resultado del punto anterior, se obtenía un algoritmo de Levenberg-Marquardt muy completo y detallado, pero a la vez complejo y, tal vez, inconsistente en cierta medida:

- El algoritmo era muy sensible a las diferentes constantes ε propuestas por Lourakis[14]. Tanto es así que para un mismo target, el cambio en el ajuste de esas variables significaba un resultado final completamente distinto.

- El algoritmo presentaba en algunas ocasiones, comportamientos extraños cuando se necesitaba ”ajustar” el resultado: el hecho de aumentar el número de iteraciones del algoritmo (k_{max}) sobre un target no presentaba los mismos resultados que ejecutar varias veces el target, empeorando notablemente el resultado en el primer caso.

- Dado un mismo target, el algoritmo calculaba soluciones completamente distintas dependiendo del punto de partida inicial del efector. Por supuesto, los mejores resultados se obtenían partiendo de una posición inicial cercana al target.

- El algoritmo era especialmente sensible a las rotaciones, llegando al extremo de empeorar el error de traslación con el fin de cuadrar la rotación.

- El algoritmo se estancaba a menudo en mínimos locales (hecho que abortaba la ejecución). Esta situcación se debía a incrementos muy pequeños que eran incapaces de mejorar la solución.

- El algoritmo trabaja en metros y radianes, mientras que el sistema de referencia de RoboComp lo hace en milímetros y radianes (lo veremos en el siguiente apartado). Esto supone modificar toda la representación del mundo y del robot para adaptarla al algoritmo. No es un problema grave, sin embargo obliga a cambiar de escala, volviéndolo incómodo. Además, a pesar de introducir la matriz de pesos W_e, en el fondo no hay un sistema claro que escale el peso de las unidades de longitud con las unidades angulares.

- El algoritmo no detectaba objetos externos al robot, como por ejemplo las mesas, por lo que al mover el brazo cerca de estos objetos podía chocar contra ellos.

Por otra parte, todo el componente de cinemática inversa se basaba exclusivamente en la información de su modelo interno, el cual no era actualizado por ningún proceso externo, como por ejemplo, la realimentación visual. Es decir, se basaba sólo en los cálculos del modelo interno del robot. Esto se asemeja a mirar un punto en una pared, cerrar los ojos e intentar tocar el punto recordando su posición.

Media error traslación	30 cm
Media error rotación	1.95 rad
Media de logros	64%
Media de mínimos locales	46%
Tiempo medio de ejecución	5.8 s

Table 4: Resultados de la cinemática inversa

Este hecho provocaba una inexactitud entre la posición dónde estaba verdaderamente el target a alcanzar y la posición del efector final robótico. Si a eso le unimos las holguras y problemas que presentan de por sí los motores de bajo coste que forman los brazos del robot, obtendremos un primer sistema de cinemática inversa impreciso y con tendencia a atascarse en mínimos locales entorno a un 46% de las ejecuciones.
Como colofón, para poner en marcha el antiguo sistema de cinemática se necesitaba una pre-calibración de los motores antes de ejecutar cualquier experimento, resultando al final un trabajo lento, tedioso y con altas probabilidades de no funcionar con exactitud.

Por todo ello, se hace necesario:

- Una revisión a fondo del algoritmo de Levenberg-Marquardt, que mejore el porcentaje de aciertos, elimine las dependencias de las distintas constantes

ε, causantes en gran medida de las anomalías del software, y reduzca su complejidad, con el fin de facilitar su mantenimiento y mejora.

- Añadir un sistema que amortigüe los problemas de calibración y holguras con el objetivo de mejorar el funcionamiento del robot en tiempo real y que reduzca la aleatoriedad de las soluciones proporcionadas por la cinemática inversa. Lo que se persigue es, por una parte evitar que la cinemática inversa resuelva targets desde posiciones iniciales muy lejanas y que el resultado varíe en exceso, y por otra, reducir los errores provocados por las holguras de los motores.

- Implementar un sistema que detecte posibles colisiones del brazo robótico con elementos de su entorno y las evite, limitando el espacio de trabajo del brazo.

4 Desgranando el Problema

Mecánicamente, los robots están formados por una cadena cinemática que se compone por una serie de eslabones unidos por articulaciones que permiten el movimiento entre cada dos eslabones consecutivos[23]. En particular las articulaciones se clasifican según el movimiento que puedan generar: movimientos de desplazamiento, de giro o de desplazamiento más giro.

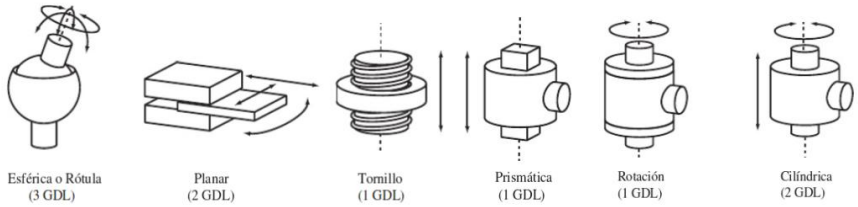


Figure 20: Tipos de articulaciones y sus grados de libertad (GDL = DOF)

En robótica, las articulaciones más usadas son las *articulaciones prismáticas*, en las que el movimiento generado entre dos eslabones es deslizante, y las *articulaciones de rotación* o revoluta, en las que el movimiento generado es de rotación, giratorio.
Las especificaciones técnicas de esos elementos pueden dar una idea aproximada de cómo funcionará el robot, si tendrá más o menos error de posicionamiento, si los motores se sobre-calentarán antes o después, si el sistema es más o menos preciso...
En cualquier caso, estas especificaciones se quedan cortas a la hora de estudiar la precisión de un robot, ya que, por un lado, todos los componentes que forman la cadena cinemática tendrán ciertas tolerancias y errores de fábrica, y, por otro, al ensamblarlos siempre se pueden cometer errores. Y precisamente de estos errores dependerá que la cinemática del robot funcione mejor o peor.
Los errores que puede contener la cadena cinemática de cualquier brazo robótico se clasifican, a grandes rasgos, en dos tipos: **errores geométricos**, provocados por diferencias entre los parámetros reales (longitudes y ángulos reales de la estructura) y los considerados en el modelo del robot, y **errores no geométricos**, provocados por deformaciones de origen térmico y dinámico, errores de redondeo en el cálculo de la transformación cinemática o el mal funcionamiento de los elementos de la cadena

cinemática, por ejemplo.

Para resolver estos errores es necesario hacer un estudio completo de la calibración del sistema y de las holguras que pueda presentar.

4.1 Calibración

La identificación de parámetros geométricos del sistema robótico o, simplemente, la **calibración** del sistema robótico tiene como objetivo estudiar y optimizar los parámetros del robot que afectan a su sistema cinemático[24]: las longitudes de los eslabones y el ángulo de giro de las articulaciones. Al final, lo que se persigue es mejorar el comportamiento o la precisión del sistema robótico una vez que está ya montado.

Un brazo robótico mal calibrado siempre cumple que, cuando se le envía a una posición target, es capaz de alcanzarla con un error e_c de calibración, sin importar la posición de partida.

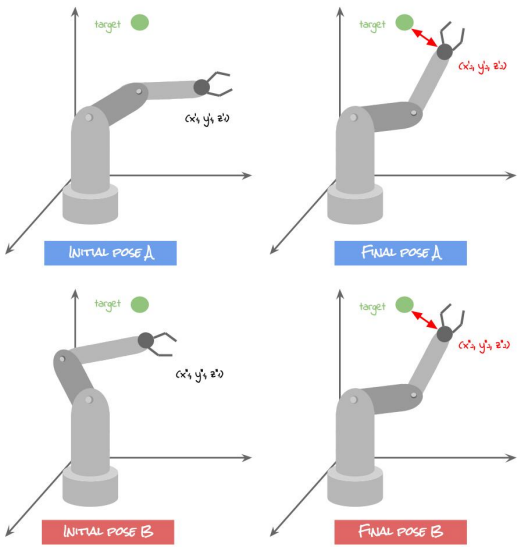


Figure 21: El error de calibración siempre es **repetible**. A pesar de partir desde dos posiciones distintas A y B, siempre se alcanza la misma pose final con el mismo error con respecto al target

Los errores repetibles o sistemáticos pueden ser compensados y eliminados

mediante técnicas de calibración, lo que permite mejorar la exactitud del robot. Las técnicas de calibración se pueden clasificar de múltiples formas. Una de ellas es por los parámetros que estudian[25]:

1. **Calibración estática**: su objetivo es identificar aquellos parámetros que influyen en las características estáticas del robot manipulador, es decir las medidas internas del robot, como la excentricidad de los engranajes de la cadena cinemática, factores de transmisión y acoplamiento, conformidades de los enlaces, geometría de los ejes de las articulaciones u *off-sets* de los ángulos de los Joints.

2. **Calibración dinámica**: su objetivo es identificar aquellos parámetros que influyen en las características dinámicas del robot, como las fricciones en el movimiento de las articulaciones o la distribución de masa en los eslabones. Para ello es necesario haber llevado acabo una calibración estática antes.

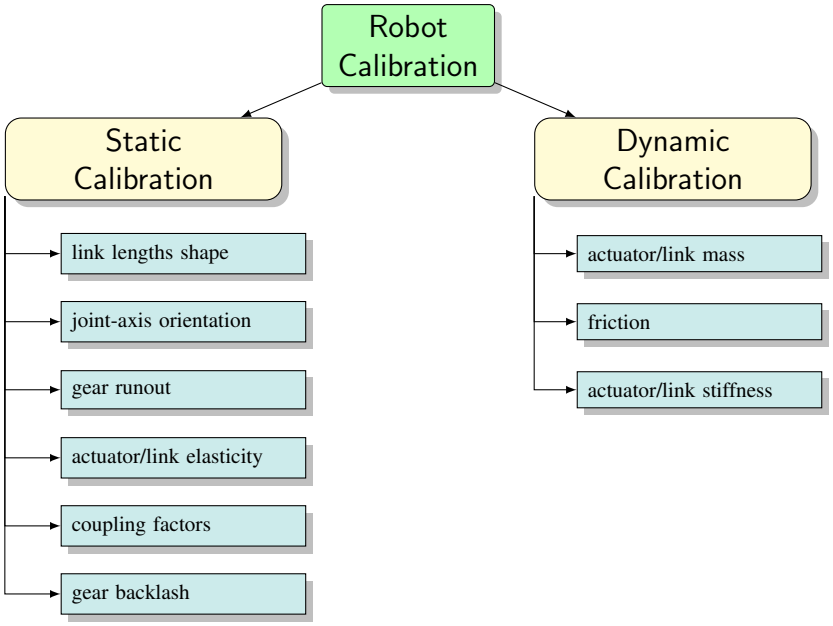


Figure 22: Tipos de calibraciones según Bernard y Albright

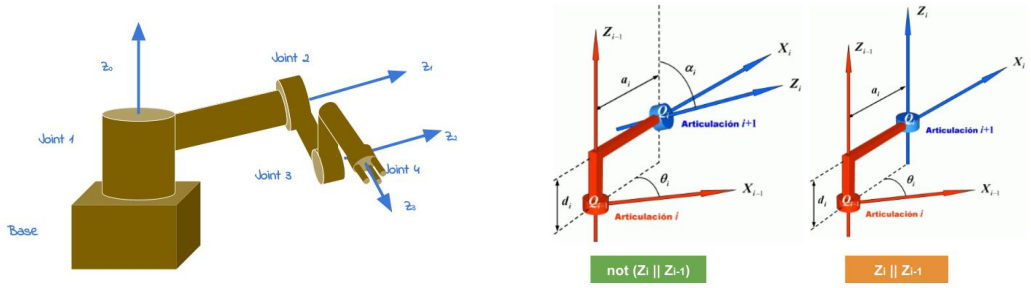
.

Existen varios métodos de calibración, como por ejemplo los basados en los cuatro **parámetros de Denavit-Hartenberg**. La representación Denavit-Hartenberg es

un procedimiento para describir la estructura cinemática de una cadena compuesta por articulaciones con un grado de libertad. Denavit-Hartenberg establece que seleccionándose de forma adecuada el sistema de referencia de cada elemento de la cadena será posible pasar de un sistema a otro mediante cuatro simples transformaciones:

- Rotación con ángulo α_i alrededor del eje Z_{i-1}
- Traslación con distancia d_i a lo largo del eslabón con eje Z_{i-1}
- Traslación a lo largo de x_i una distancia a_i.
- Rotación alrededor del eje x_i un ángulo α_i

Estos parámetros se calculan mediante la asignación de sistemas de referencia a cada joint de la cadena cinemática:



(a) A cada articulación se le asigna un sistema de referencia y un eje de giro, de tal forma que la cadena cinemática siempre crezca en el eje Z. Así, la articulación i-ésima girará sobre el eje Z_{i-1}.

(b) Cálculo de los parámetros de D-H cuando los ejes Z_i y Z_{i-1} no son paralelos (primer caso) o, por el contrario, son paralelos (segundo caso)

Figure 23: Representación Denavit-Hartenberg

Para calibrar un robot se puede utilizar la calibración cinemática, tanto de bucle abierto como de bucle cerrado. El primer método de calibración deja libre el efector final del robot manipulador. Primero se miden varias veces la localización real del efector para distintos valores de las variables articulares de

Denavit-Hartenberg, y después, para esos mismos valores de las variables articulares, se calcula la localización según el modelo ideal de la cadena cinemática. Se comparan los resultados,y se obtiene una estimación óptima de los parámetros de Denavit-Hartenberg que minimice las diferencias entre el modelo y el robot real.
El segundo método de calibración sigue los mismos pasos que la calibración de bucle abierto salvo que ahora se restringe el movimiento del efector final en uno o varios grados de libertad mediante restricciones de orientación, restricciones de movimiento sobre una línea, restricciones de movimiento sobre un plano...
Normalmente la calibración de bucle abierto, aunque puede ser muy precisa en ciertos sistemas, es más complicada y obtiene peores resultados que la de bucle cerrado. En la calibración de bucle cerrado se puede utilizar otro robot manipulador que calibre el actual sistema robótico, o un sistema de calibración mediante información visual proporcionada por una o varias cámaras.
En el caso de Shelly, utilizaremos un modelo ideal del robot que recoge la estructura de links y joints del robot. La disposición de este modelo ha sido calculada obeservando la cadena cinemática del robot real y teniendo en cuenta las especificaciones de los motores dadas por el fabricante correspondiente. Sin embargo, es bastante probable que el robot real y su modelo ideal no coincidan en sus medidas, ni en las longitudes de los eslabones ni en los ángulos de giro de los joints. Ante este hecho se hace necesario calibrar el robot, sin embargo este proceso no es una tarea sencilla ni perdurable en el tiempo, ya que cualquier pequeña perturbación como un choque o un roce contra algún objeto o el sobecalentamiento de los motores puede echar por tierra los resultados de la calibración.
Lamentablemente, para el robot Shelly actualmente no existe un método de calibración de este tipo.

4.2 Holguras en los brazos robóticos

Otro problema al que se debe enfrentar la cinemática inversa del robot es a las holguras de la cadena cinemática.

Se adelantó al comienzo de este capítulo que las articulaciones más usadas en robótica son las prismáticas y las de rotación o revoluta, en las que el movimiento generado es de rotación, giratorio. Por otro lado, los robots manipuladores necesitan un mínimo de seis articulaciones para poder posicionarse y manejar objetos en el espacio 3D. Estas articulaciones deben cumplir con una serie de requisitos para que el robot funcione bien y pueda manipular objetos de forma precisa:

1. El sistema de transmisión del movimiento debe ser pequeño y de poco peso, de gran rendimiento y que no presente **holguras** en su estructura, capaz de soportar un funcionamiento prolongado y continuo.

2. Los reductores deben ser pequeños y ligeros, con un **juego angular** pequeño, que provoquen poco rozamiento y que no afecten negativamente a la velocidad y precisión del robot.

Las holguras presentes en un sistema robótico provocan errores no sistemáticos, de tal forma que, cuando a un brazo se le ordena ir a una posición target desde dos puntos de partida distintos, no sólo no alcanzará la posición deseada, si no que además lo hará con errores distintos.

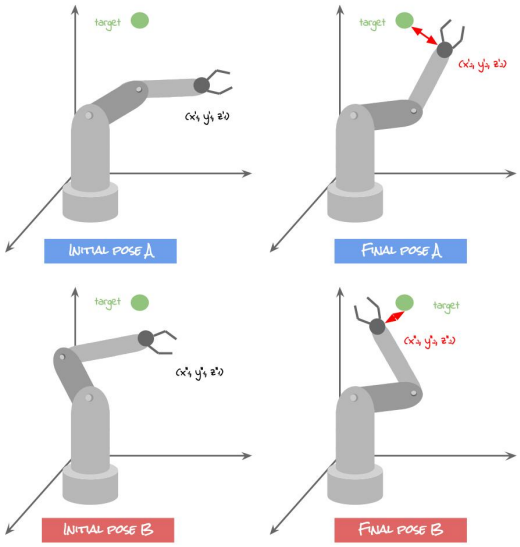


Figure 24: El error de holgura no es **repetible**. A pesar de tener un mismo target, las posiciones finales varían. Incluso si las posiciones de partida A y B fueran la misma, las posiciones finales serían distintas, con un error distinto.

Este tipo de error no es sistemático por lo que no se corrige con la calibración del sistema robótico. Se hace necesario otro sistema más complejo que pueda calcular y revertir los errores producidos por las holguras.

El brazo de Shelly sobre el que se ejecuta la cinemática inversa, como dijimos, cuenta con nueve grados de libertad para poder alcanzar cualquier posición que esté dentro de su rango de trabajo. Estas articulaciones están colocadas en un armazón de metal, conectadas unas a otras mediante engranajes ligeros y protegidas por caparazones de plástico diseñados por una impresora 3D. Las articulaciones se dividen en dos tipos: cinco motores faulhaber que constituyen los tres motores del hombro del robot, el codo y el antebrazo, y cuatro servos Dynamixel que forman los dos motores de la muñeca (RX 64) y los motores de las pinzas del efector final.

Toda la estructura se sustenta sobre elementos de bajo coste, diseñados, construidos y finalmente montados en el laboratorio Robolab. Por lo que las holguras y el juego angular del brazo robótico es bastante grande:

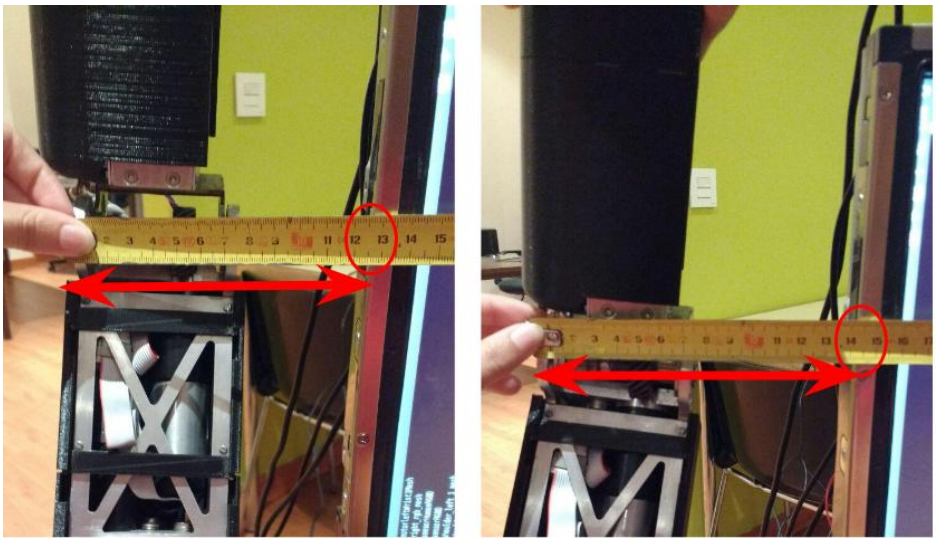


Figure 25: Holgura en el segundo faulhaber del hombro del robot

En la figura (25) se puede observar la holgura que se produce en la estructura al empujar el brazo robótico a partir del segundo faulhaber del hombro. A la altura del codo se puede observar que hay cerca de un centímetro y medio entre la posición de descanso y la posición modificada.

Además, la posición cero de las articulaciones puede variar si el motor es rotado a la fuerza, por ejemplo si el brazo robótico impacta o si el motor choca contra algún objeto rígido.

Al final este problema se traduce en que, cuando el módulo de cinemática inversa recibe un target, hará sus cálculos sobre un modelo que no se corresponde con la realidad y con una estructura poco precisa, de tal forma que, si para alcanzar la posición objetivo el codo debe estar rotado exactamente $\frac{-\pi}{2}$, la articulación correspondiente sólo estará rotada $\frac{-\pi}{3}$ debido al juego angular, por lo que el efector final no puede alcanzar la pose deseada.

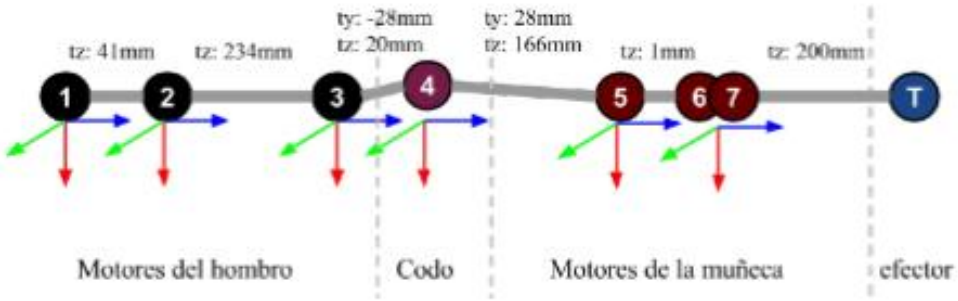


Figure 26: Cadena cinemática del brazo derecho de Shelly. No se añaden los dos dynamixel del efector final

La cinemática inversa hará sus cálculos sobre el modelo, sin tener en cuenta los problemas del robot real, y propondrá unos valores angulares para cada joint de la estructura, en la creencia de que el brazo robótico adoptará esa configuración de forma exacta. En la realidad, las holguras de la estructura harán casi imposible que el efector alcance la pose objetivo, produciéndose un error entre el target y la pose final alcanzada por el efector robótico.

4.3 Error de posicionamiento y repetibilidad en bucles abiertos

Como se ha visto en las secciones 4.1 y 4.2, los dos problemas principales a los que se enfrentan las cadenas cinemáticas abiertas que forman los brazos robóticos manipuladores, son la calibración y las holguras.

Ambos problemas afectan al sistema robótico, perjudicando su precisión de posicionamiento y su repetibilidad. En este punto, se hace necesario distinguir estos dos términos[26]:

1. La **repetibilidad** es la habilidad de un sistema robótico para regresar a la misma posición y orientación, dentro del espacio de trabajo del robot, desde cualquier punto de inicio. Se ve afectada negativamente por las holguras.

2. La **precisión** o **exactitud** del robot es la habilidad que tiene para moverse con fidelidad a una posición deseada en un espacio tridimensional. Se ve afectada por la calibración.

Los errores de calibración no impiden que el sistema robótico tenga repetibilidad, ya que dado un target el sistema robótico es capaz de situarse en la misma posición, a pesar de partir de puntos iniciales distintos. El problema es que la posición alcanzada siempre será el target más el error que provoque la mala calibración. Por otro lado, los fallos de holguras hacen que el sistema no sea ni preciso ni con una repetibilidad alta.

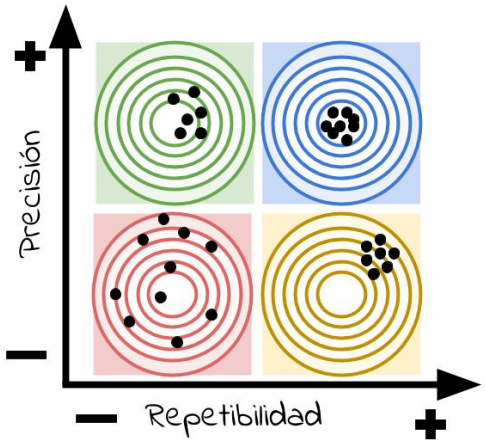


Figure 27: Distintas configuraciones de un robot manipulador: (en verde) mucha precisión pero poca repetibilidad, (en azul) mucha precisión y mucha repetibilidad, (en rojo) poca precisión y poca repetibilidad, (en amarillo) poca precisión y mucha repetibilidad.

El problema principal de las cadenas abiertas, como son los robots manipuladores, es que no poseen ningún mecanismo que les permita auto-corregir los errores de

posición y orientación del efector final, por lo que es necesario un software adicional y de un modelo cinemático que lo supla. Con estos elementos el objetivo a perseguir será *cerrar la cadena cinemática*.

Internamente, cuando se introduce un target en el sistema robótico, éste se referencia en el sistema base del robot. La cinemática inversa crea un bucle o un polígono mal cerrado entre el target y el efector final. A partir de ahí, la cinemática inversa intentará cerrar ese bucle moviendo el efector final hacia la posición objetivo marcada. Para ello irá minimizando el error, que es la ”apertura” del bucle o la distancia entre el efector y el target.

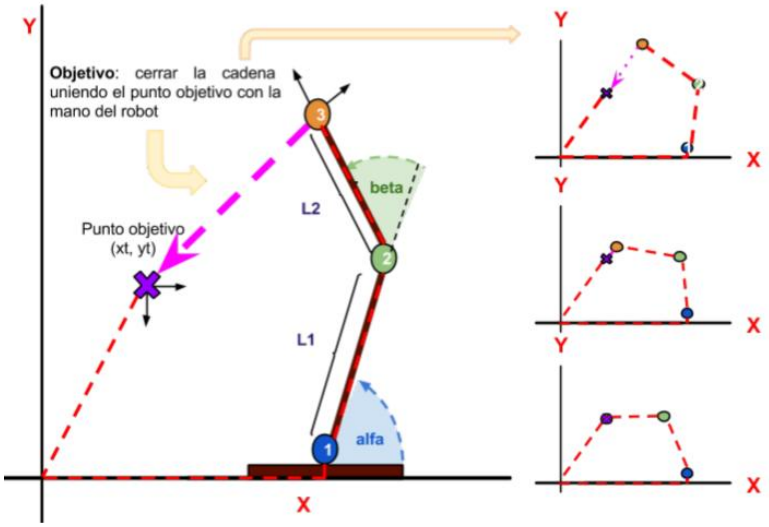


Figure 28: Objetivo de la cinemática inversa, reducir la apertura del bucle

Esto es suficiente para el modelo interno del robot. Pero no para el robot real, en el que la mala calibración y el juego angular trastocan la cadena y el bucle no cierra. Una posible solución para superar estos problemas es proporcionar algún tipo de **feedback** al sistema. Para estos casos, lo que más se utiliza es la realimentación visual.

La realimentación visual viene dada por una cámara o una serie de cámaras, calibradas a su vez, y perfectamente referenciadas dentro del sistema robótico[29], de tal manera que la posición del target y del efector final robótico pueda ser calculada desde varios

[29]Estas cámaras pueden ser externas al robot o estar colocadas en la misma estructura del sistema robótico. También pueden ser fijas o móviles.

sistemas de referencia. Esto proporcionará información redundante al sistema, muy útil para corregir las desviaciones que los elementos de la cadena cinemática del robot provoquen.

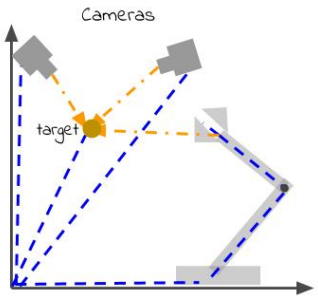


Figure 29: El sistema de cinemática se complementa con varias cámaras que proporcionan realimentación visual, muy útil para comprobar el posicionamiento tanto del efector final como del target

4.4 Planificación de trayectorias: evitar colisiones

Los robots manipuladores son capaces de moverse en un rango de posiciones definido por la configuración del brazo robótico. La cinemática inversa hace posible que el manipulador pueda mover su efector final a todas las posiciones que estén en su espacio de acción.

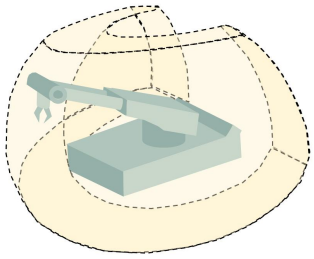


Figure 30: El área de trabajo se compone de todas aquellas posiciones físicamente alcanzables por el efector final del brazo robótico

El funcionamiento básico de la cinemática inversa es calcular unos ángulos que permitan al efector final posicionarse sobre el target, de tal manera que el robot moverá su efector final a través del espacio de trabajo, desde su posición inicial hasta su posición final, formando una trayectoria.

Este funcionamiento simple de la cinemática inversa supone un problema cuando hay objetos susceptibles de provocar choques, golpes o accidentes dentro del área de trabajo. Sería el caso de coger una taza que está encima de una mesa. El robot debe aproximarse hasta tener la taza dentro de su espacio de trabajo para poder cogerla. Sin embargo, también entrará en ese espacio la mesa. La mesa supone un **obstáculo** dentro del espacio de trabajo del robot.

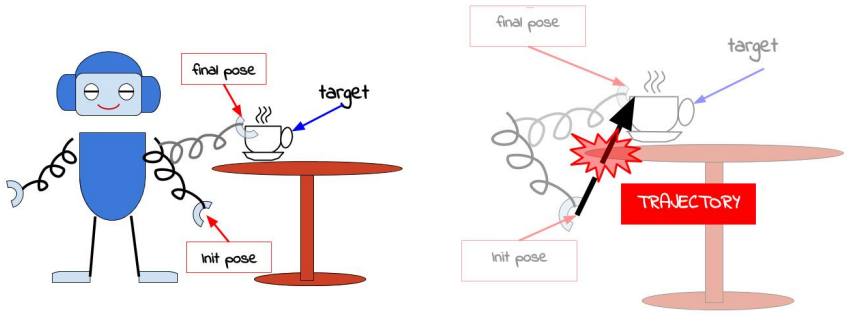


Figure 31: El robot traza una trayectoria desde su posición inicial hasta la posición objetivo, sin tener presente la mesa que hay por medio. Como resultado, cuando el robot mueva su brazo, chocará contra la mesa.

El robot no sólo puede chocar contra objetos externos, como una mesa o una pared, también puede chocar contra sí mismo. En el caso de Shelly, el brazo robótico puede impactar contra la pantalla táctil que el robot tiene instalada en su pecho.

Para evitar estos incidentes es necesario llevar un control de la trayectoria o **planificación del movimiento**[27]. Esta planificación pretende resolver el problema de encontrar un camino adecuado, libre de obstáculos, que el robot pueda seguir para alcanzar la posición objetivo desde cualquier posición inicial. El problema no es trivial ya que dada una configuración pueden existir múltiples caminos que lleven desde el punto A hasta el punto B. Todo dependerá de las restricciones y costes que implique cada camino.

Algunas técnicas de planificación se basan en simular previamente el movimiento para calcular si hay choques o no antes de ejecutar verdaderamente el movimiento en el robot real. Otras técnicas se basan en muestreos del espacio, grafos y árboles de posicionamiento.

Entre las técnicas más conocidas destaca la **planificación con generación aleatoria**[28]. Se trata de una técnica de muestreo que divide el espacio de configuraciones C del sistema robótico en posiciones discretas, mediante una rejilla regular. A cada celda se le asigna un valor o coste dependiendo de su proximidad a un obstáculo o a la cercanía con los puntos origen y destino. El objetivo de esta técnica es generar una trayectoria basada en el descenso del gradiente, de tal forma que el camino calculado vaya en una dirección hasta que se alcance el mínimo absoluto, representado este, por la posición de destino. Sin embargo, este algoritmo es sensible a los mínimos locales, en los que el gradiente se queda atascado.

Para evitar esta situación, la planificación con generación aleatoria *genera configuraciones aleatorias*, comprobando las colisiones y el desplazamiento generado. En esta planificación se basa la familia de algoritmos **árboles RRT** (*Rapidly-Exploring Random Tree*). Son algoritmos de consulta simple por muestreo y se fundamentan en recoger información de estados o posiciones accesibles para el brazo robótico y que pertenecen al espacio libre de obstáculos.

Algorithm 2 Basic RRT algorithm

procedure BASICRRT(x_{init}, K, Δt)
  $\tau.init(x_{init})$
  while $(k < K)$ **do**
    $x_{rand} \leftarrow RandomState()$
    $x_{near} \leftarrow NearestNode(x_{rand}\tau)$
    $u \leftarrow BestPerformance(x_{rand}, x_{near}, \&x_{new}, \&success)$
    if *success* **then**
      $x_{new} \leftarrow Integrate(x_{near}, u, \Delta t)$
      $\tau.AddNode(x_{new})$
      $\tau.AddLink(x_{near}, x_{new}, u)$
      $k \leftarrow k+1$
    end if
    Return τ
  end while
end procedure

El funcionamiento es bastante intuitivo: el algoritmo RRT Básico toma una configuración inicial x_{ini} del espacio de configuraciones libre de obstáculos y colisiones

C_{free}, y va iterando hasta alcanzar un máximo de iteraciones, K. En cada iteración, genera **aleatoriamente** una nueva configuración x_{rand} del espacio C, y se calcula su configuración almacenada más cercana, x_{near}[30]. El resultado de integrar esos dos nodos se irá almacenando, añadiendo ramas en todas las direcciones posibles, por lo que el árbol RRT irá creciendo hasta que se alcance la configuración final x_{final}.

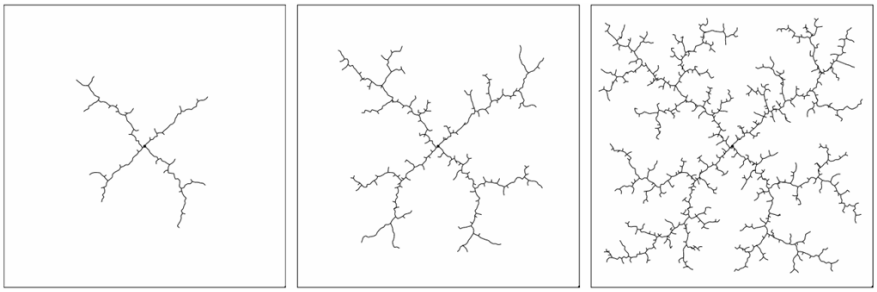


Figure 32: Expansión del árbol RRT

De esta forma el algoritmo RRT calcula un camino continuo conectando la configuración inicial, x_{ini}, con la configuración final, x_{final} mediante técnicas probabilísticas y teniendo en cuenta los obstáculos y las restricciones existentes, de tal forma que todas las configuraciones que forman el camino pertenecen a C_{free}.

El algoritmo básico (2) explora de forma aleatoria y densamente el espacio de configuraciones del robot, C, añadiendo a la estructura aquellas configuraciones libres de obstáculos y colisiones, C_{free}. Por otra parte, RRT no necesita establecer campos potenciales, ahorrando tiempo de cómputo, asegurando una exploración equiprobable de todo el espacio de configuraciones. Además, resulta un algoritmo sencillo de entender e implementar, su ejecución es rápida y fácil de integrar en escenarios complejos.

Existen más algoritmos de planificación de caminos, muchos de ellos modificaciones y mejoras del algoritmo RRT básico, como el RRT-Connect, mucho más agresivo ya que en cada iteración añade todos los nodos que puede con respecto a x_{rand}, o los RRT Bidireccionales.

[30]Para ello se necesita una métrica R que relaciones x_{rand} con x_{near} y que tenga en cuenta algunas restricciones. Por ejemplo la distancia euclídea entre las dos configuraciones y que no existan obstáculos o colisiones entre ambos

5 Material y método

Una vez planteados los problemas a los que el antiguo sistema de cinemática inversa no podía hacer frente, es el momento de describir en esta sección los cambios y reajustes del nuevo sistema de cinemática inversa. Este nuevo sistema software, al igual que todo el software descrito en este trabajo, ha sido desarrollado sobre **RoboComp**[29][9], y actualmente es usado por su comunidad.

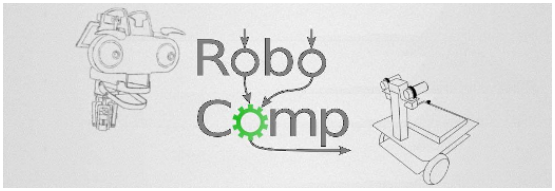


Figure 33: Logo de RoboComp

5.1 RoboComp

RoboComp surgió en 2005[31] como un framework de código abierto[32] diseñado para la programación orientada a componentes y dotado de herramientas especificas. Su objetivo fundamental es servir como apoyo a los desarrolladores software de robótica. RoboComp se basa en dos pilares fundamentales. Por un lado, el paradigma de la *Programación Orientada a Componentes*. Caracterizado por descomponer el sistema en varios componentes lógicos, capaces de comunicarse entre sí a través del intercambio de mensajes mediante interfaces. Cada componente ofrece un determinado servicio, por lo que al final se tiene una estructura en la que los componentes se intercambian información para llevar a cabo sus operaciones, lo que permite un desarrollo de códigos muy ágil y rápido, además de que la estructura de componentes es muy flexible y adaptable a la naturaleza del problema. Por otro lado, utiliza el middleware *Internet Communications Engine* (Ice) para la comunicación entre componentes, por lo que puede trabajar en entornos muy heterogéneos, desde

[31] En su desarrollo actual participan Universidades de Extremadura, Málaga, Jaén, Castilla La Macha y la Carlos III de Madrid además de la empresa Indra.

[32] Su código está completamente disponible en el repositorio https://github.com/robocomp

distintos sistemas operativos a componentes con implementaciones en distintos lenguajes de programación.

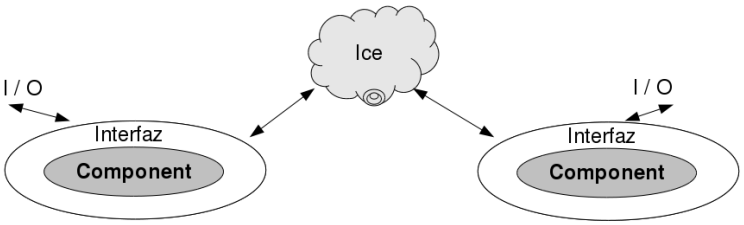
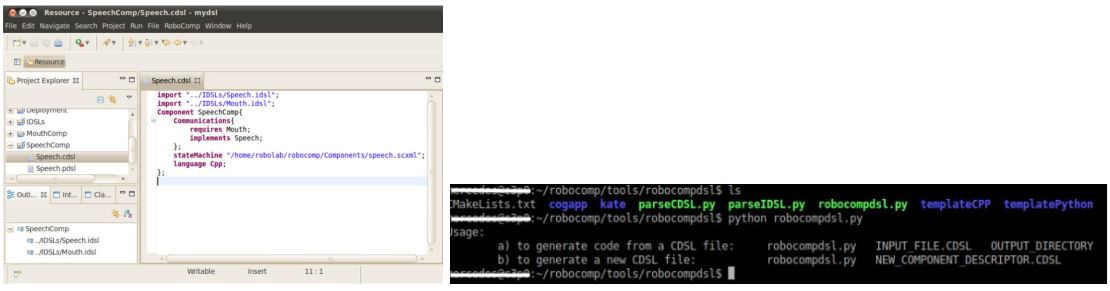


Figure 34: Estructura de componentes. Comunicación a través de interfaces y sobre el middleware ICE

A muy alto nivel, RoboComp se puede dividir en dos partes: la primera consta de una serie de **herramientas** y **librerías**, que facilitan enormemente la tarea de diseño, desarrollo y depuración, y la segunda se compone de una gran variedad de **componentes** ya implementados y preparados para ser integrados en proyectos de mayor envergadura fácilmente.

5.1.1 Herramientas y librerías de RoboComp

En RoboComp abundan las herramientas que facilitan tanto el desarrollo, como el seguimiento y la actualización de los componentes que se desarrollen. Entre todas las herramientas, en este proyecto han destacado dos: el generador de componentes **RoboComp DSL** y el organizador de componentes **RoboComp Manager**.

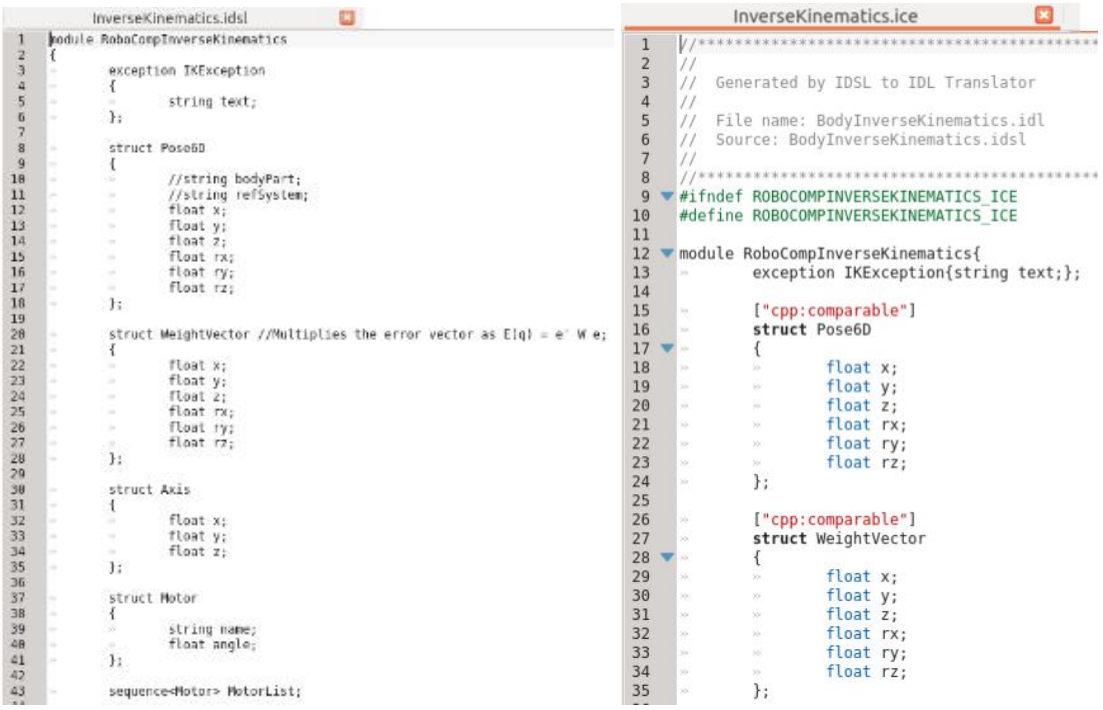


(a) Visor DSL antes (b) DSL editor actual

Figure 35: Evolución del generador de componentes

El generador de componentes **RoboComp DSL** ha evolucionado desde un visor y

gestor de documentos basado en el entorno Eclipse hasta un script en python capaz de generar los componentes basándose en cinco tipos de ficheros: CDSL, IDSL, PDSL, DDSL e InnerModelDSL[29].



(a) Fichero IDSL original

(b) Fichero ICE derivado del IDSL

Figure 36: Ficheros para declarar una interfaz de un componente.

De entre todos los ficheros proporcionados[33], para generar los componentes de cinemática inversa se necesitarán los ficheros **CDSL**[34], **IDSL**[35] e **InnerModelDSL**[36]. Para generar las interfaces de comunicación de los componentes, el programador tan solo debe escribir su fichero IDSL, en el que definirá los métodos de la interfaz, las

[33]Los ficheros PDSL definen plantillas de parámetros de configuración. Los DDSL describen qué componentes se van a usar, cómo se deben ejecutar y qué configuración deben usar)

[34]Para definir la estructura del componente: los proxies de comunicación, el lenguaje de programación, las interfaces y los tópicos usados por el componente además de otros aspectos como el soporte gráfico de Qt, dependencias con otras clases o librerías...

[35]Para definir las interfaces por las que se comunicarán los componentes y sus tópicos, tipos básicos de datos, tipos enumerados, excepciones...

[36]Un fichero XML que describe toda la cinemática del robot y su entorno. Representa representa el modelo interno del robot. Mediante la herramienta **rcisInnerModel** se transforma en una clase C++ (**InnerModel**)muy útil para el proyecto

estructuras y tipos de datos que se utilizarán y las excepciones propias de la interfaz. Preparado el IDSL se genera su correspondiente fichero .ice.

Para generar el componente, el programador debe indicar que interfaces va a implementar (*implements*) o a cuales va a conectarse (*requires*), así como el lenguaje de programación (C++, Python,...) y si utilizará o no una interfaz gráfica de usuario (*use QT*). Con la herramienta RoboCompDSL se generará automáticamente gran parte del código del componente (inicialización de proxies, interfaces, timers...), simplificando y reduciendo notablemente la labor del desarrollador[29]. De esta forma cualquier componente que haya sido creado con la ayuda de la herramienta RoboCompDSL tendrá siempre dos partes bien definidas:

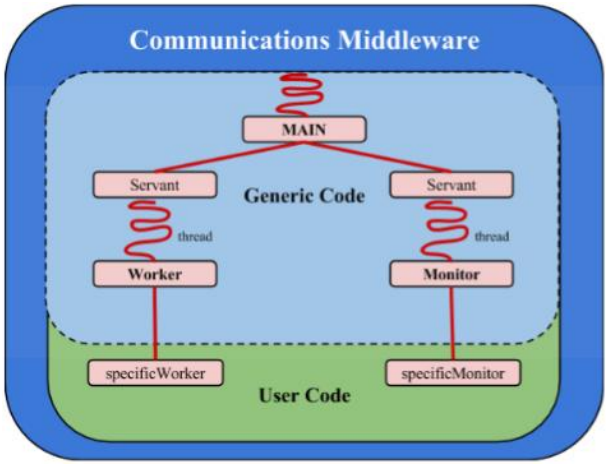
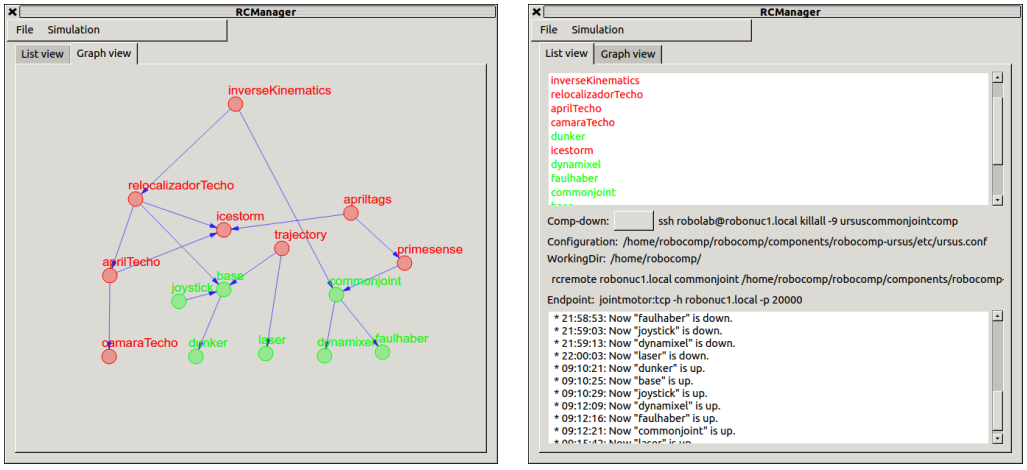


Figure 37: La estructura de cualquier componente de RoboComp se divide en dos partes: una parte genérica que contiene la lógica de la comunicación entre los componentes y la estructura general del componente, y una parte específica donde se almacena el código del desarrollador.

El funcionamiento interno del componente es sencillo. El componente dispone de un programa principal o *main* (guardado en el fichero *nombre_del_componente.cpp*) que se encarga de llamar a la clase autogenerada *Monitor*. El Monitor es ejecutado por un hilo que lee los parámetros de configuración del componente (guardados en el fichero de configuración correspondiente), y lo inicializa, levantando a su vez el hilo que ejecutará la clase *Worker*. El Worker tiene una parte autogenerada (*generic*) que se encarga de la incicialización de los proxies y los parámetros de comunicación,

y otra específica del programador (*specific*). Esta clase es la que lleva a cabo toda la funcionalidad del componente mientras que el Monitor se queda dormido, despertando periódicamente para llevar a cabo tareas de monitorización sobre el Worker. Además, el componente dispone de la interfaz *CommonBehavior*, que le permite acceder y cambiar los parámetros comunes del componente en tiempo de ejecución.

Una herramienta muy útil en el transcurso de este proyecto ha sido el administrador de componentes **RoboComp Manager** o **RCManager**. Esta herramienta es francamente útil, sobre todo cuando el número de componentes que necesita el robot para funciona es elevado. Sirve para controlar y especificar los componentes de RoboComp, su puesta en marcha, su ejecución y detención. Se trata de una simple e intuitiva interfaz de usuario en la que se muestra, en forma de grafo, los componentes (nodos del grafo) que están en ejecución (nodos verdes), o no (nodos rojos) y las relaciones y dependencias existentes entre ellos (enlaces del grafo).



(a) Graph view (b) List view

Figure 38: Los nodos verdes indican aquellos componentes que se están ejecutando en ese momento, mientras que los nodos rojos son componentes que aún no han sido levantados. Las dependencias vienen dadas por las flechas azules de tal forma que, por ejemplo, el componente apriltags necesita conectarse al componente primesense

El programador puede levantar los componentes o parar su ejecución desde el RCManager. Tan sólo se debe pulsar sobre el componente deseado y seleccionar la opción UP o DOWN. También puede consultar un histórico de acciones llevadas a cabo

con el RCManager o los path de los ficheros de configuración de cada componente. Por debajo, RCManager se sustenta sobre un fichero XML, que contiene información de configuración (colores de los nodos, tamaños, posiciones) y comandos para lanzar y finalizar la ejecución de los componentes.

```
<rcmanager>
  <generalInformation>
    ... <!-- SOME CONFIGURATION PARAMS -->
  </generalInformation>
  <!-- THE COMPONENT -->
  <node alias="COMP_NAME" endpoint="COMP:tcp -h IP -p PORT">
      <dependence alias="OTHER_COMP" /> <!-- Dependence with other comp. -->
      <workingDir path="WORKING DIRECTORY" />
      <upCommand command="rcremote IP COMP_NAME BIN_PATH ./bin/EXECUTABLE
          --Ice.Config=CONFIG_FILE"/>
      <downCommand command="ssh IP_MACHINE killall -9 EXECUTABLE"/>
      <configFile path="PATH/CONFIG_FILE" />
      <!-- TO DRAW THE NODE INTO THE RCMANAGER UI -->
      <xpos value="116.463101156" />
      <ypos value="20.0105420209" />
      <radius value="10.0" />
   </node>
</rcmanager>
```

Otra herramienta útil, teniendo en cuenta el carácter distribuido de los componentes en distintas máquinas, es la utilización de la herramienta **RoboComp Remote**. Esta herramienta junto con **RoboComp Remote Server** sirven para comunicar dos computadores remotos de forma fácil y sencilla mediante una dirección IP y una contraseña, sobre el protocolo TCP y en el puerto 4242.

Además de estas herramientas, RoboComp dispone de una serie de librerías muy útiles y completas, destinadas a tratar temas tan variados como la visión por computador, el cálculo de matrices, acceso al hardware, widgets gráficos, lógica difusa... Entre ellas destacan **QMat** e **InnerModel**. La primera librería proporciona métodos para el manejo de matrices, muy útiles en las operaciones con matrices de transformación, rotación y traslación de vectores, cálculo de ángulos... etc. La segunda librería se encarga de representar la cinemática del robot y del mundo que lo rodea. Se ocupa de la representación del cuerpo del robot y del entorno en el que trabaja. Esta librería supone un gran apoyo a la hora de realizar cálculos geométricos

y algebraicos relacionados con el manejo de sistemas de referencia.

Podría decirse que *InnerModel* es el núcleo de RoboComp y está escrita en C++ y se basa en descripciones, almacenadas en ficheros XML, de la cinemática del robot y del entorno que lo rodea, sirver para calcular operaciones, como transformaciones geométricas entre elementos, consultar el modelo cinemático, modificarlo o actualizarlo.

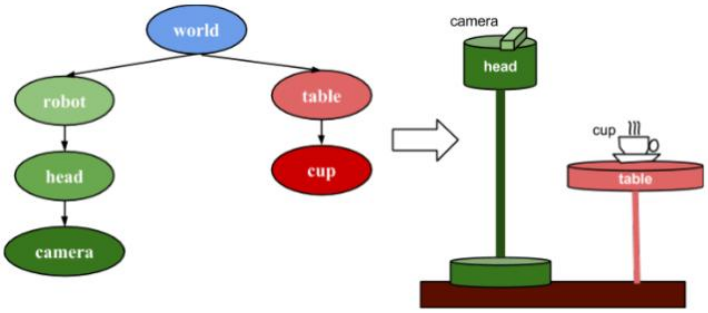


Figure 39: Representación del entorno con innerModel

Las descripciones XML forman un **árbol cinemático** donde cada nodo contiene la transformación geométrica respecto a su padre. En este árbol se define el propio robot y su entorno, los nodos representan elementos del entorno, como por ejemplo, tazas, mesas, paredes...

Por ejemplo, para describir el modelo propuesto en la figura 43 (en la que aparecen un robot con una cámara y una mesa con una taza) mediante una representación XML se necesitaría el siguiente código:

- Código para representar la mesa con la taza

```
<innerModel>
  <transform id="world">
    <transform id="floor">
      <transform id="table">
        <mesh id="table_m" file="../t_mesh.jpg" tx="0" ty="100"
        tz="600" rx="1.57" ry="0" rz="0" scale="80"/>
          <transform id="cup" tx="0" ty="700" tz="500">
            <mesh id="cup_m" file="../c_mesh.jpg" scale="15"/>
        </transform> <!-- fin taza -->
      </transform> <!-- fin mesa -->
      .....
```

- Código para representar el robot con la cámara:

```
....
<differentialrobot id="robot" port="10004">
  <transform id="head" ty="1002" tz="-120">
    <joint id="neck" port="20000" axis="y" min="0.0" max="3.14">
      <transform id="cameraPose" ty="10"/>
        <camera id="camera" width="640" height="480"/>
      </transform> <!-- Fin camara -->
    </joint> <!-- Fin cuello -->
  </transform> <!-- Fin Cabeza -->
 </differentialrobot> <!-- Fin robot -->
</transform> <!-- fin suelo -->
</transform> <!-- fin mundo -->
</innerModel>
```

En este código se puede observar varias etiquetas que definen el modelo geométrico del mundo representado en la figura 43. La estructura de etiquetas anidadas indican las relaciones geométricas del modelo. Así, al representar en el árbol cinemático una etiqueta *A* anidada dentro de otra etiqueta *B*, *A* será un nodo que "cuelgue" del nodo *B*. De este ejemplo de fichero XML, interesan fundamentalmente dos tipos de nodos:

1. Los nodos *transform* guardan las componentes de traslación (*tx*, *ty* y *tz*) y rotación (*rx*, *ry* y *rz*) de la matriz de transformación entre el nodo actual y el anterior.

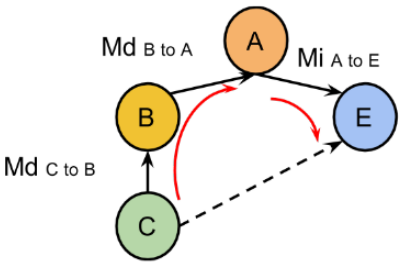
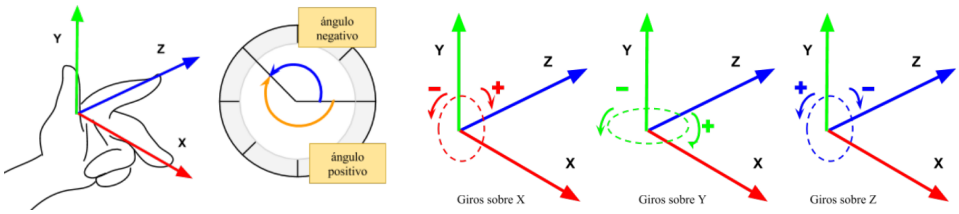


Figure 40: Transformaciones para pasar del sistema de referencia C al sistema de referencia E. Se asciende por el árbol cinemático calculando las matrices de transformación directas de cada nodo hijo con su padre ($Md_{C\ to\ B}$ y $Md_{B\ to\ A}$), hasta llegar a la raíz, momento en el que se desciende por el árbol calculando las matrices de transformación inversas desde el nodo padre hacia el nodo hijo ($Mi_{A\ to\ E}$)

2. Los nodos *joint* simulan articulaciones que giran sobre un eje (*axis*) con las restricciones que les impongan un ángulo máximo de rotación (*max*) y un

ángulo mínimo (*min*). Llegados a este punto, es conveniente introducir las características del sistema de referencia que utiliza RoboComp.



(a) Regla de la mano izquierda (b) Sentido de giro de los ejes

Figure 41: Regla de la mano izquierda y ejes cartesianos

RoboComp utiliza un sistema de ejes cartesianos cuya unidad de medida son los milímetros, para la traslación, y los radianes, para la rotación. Además, el sistema de referencia se rige por la **regla de la mano izquierda**, o regla de Flemming, en la que dedos pulgar, índice y corazón perpendiculares entre sí forman los tres ejes cartesianos del sistema, Y, Z y X, respectivamente, y donde el sentido de los ángulos es positivo en sentido horario y negativo en sentido anti-horario. Para construir la matriz de rotación compuesta se tiene en cuenta el orden $R_x \cdot R_i \cdot R_z$, es decir, primero se rota en X, después en Y y por último en Z.

Aparte, *InnerModel* tiene definidos muchos más nodos, como los *mesh* para cargar mallas, o nodos especiales que representan distintos tipos de sensores, como *camera* o *laser*.

La facilidad y utilidad de esta herramienta convierte a *InnerModel* en una base fundamental tanto del antiguo como del nuevo sistema de cinemática inversa del robot. Para visualizar de forma gráfica el contenido de un árbol cinemático, RoboComp cuenta con la herramienta **RCIS** o *Robocomp Component Inner Model Simulator*, basado en el motor de gráficos 3D de software libre para videojuegos *Open Scene Graph*. Esta potente herramienta ofrece para cada nodo especial descrito en el árbol cinemático (*joint*, *laser*, *camera*...) las mismas interfaces de comunicación de los componentes reales asociados a ellos (por ejemplo, para los nodos *joint* ofrece la

misma interfaz que el componente *commonjointComp*, encargado de gestionar los componentes *DynamixelComp* y *FaulhaberComp*, que a su vez leen los parámetros de los motores reales del robot y modifican sus valores angulares). De forma que la comunicación es transparente al resto de componentes, es decir, un componente que necesite el movimiento de un motor, escribe la orden de la misma forma, sin importarle si este es simulado o real.

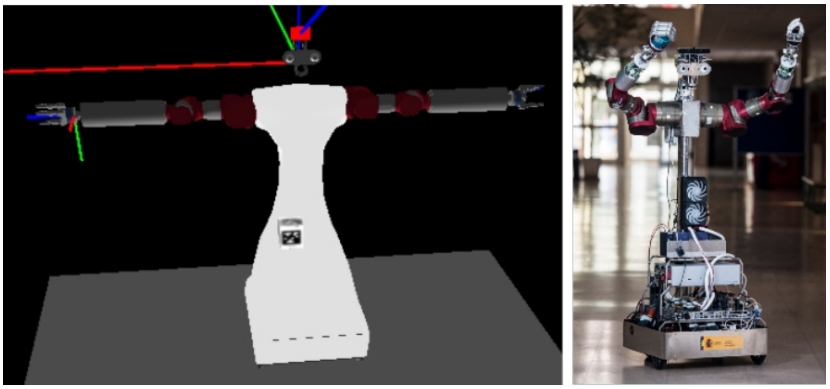


Figure 42: Representación del entorno con innerModel

5.2 Una estrategia en dos etapas: movimiento en bucle abierto seguido de servo control visual en una representación local de espacio libre

Ante los problemas descritos en la sección 4, se plantea un nuevo reto: *¿Cómo hacer frente a estas dificultades?*. Por un lado, los brazos de Shelly son cadenas abiertas, por lo que sus movimientos son independientes (el movimiento del codo no obliga a la muñeca a girar. Además que el movimiento de la muñeca no estaría condicionado ni restringido por el movimiento del brazo) y no proporcionan feedback a la cadena cinemática, y por otro, estos brazos tiene defectos de calibración y sus motores y enlaces tienen ciertas holguras que producen un elevado índice de error en el posicionamiento.

Ante esta situación desfavorable para la cinemática inversa, este trabajo propone, diseña e implementa una solución para reducir los efectos negativos de estas

deficiencias: añadir al sistema de cinemática inversa realimentación visual (**Visual servo control**).

De esta forma, el bucle abierto que forma la cadena cinemática del brazo se ”cierra” gracias a la información redundante que proporciona tanto el modelo cinemático, como la o las cámaras instaladas.

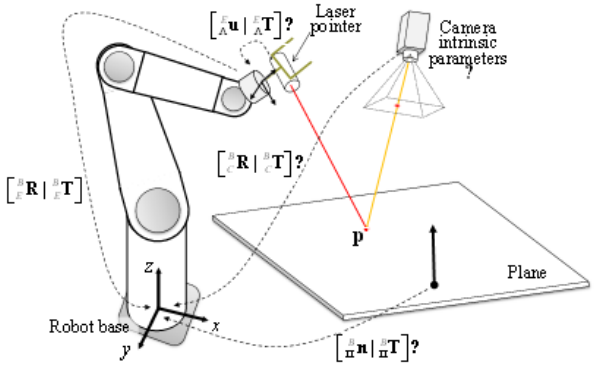


Figure 43: Esquema de control visual

La información visual obtenida por medio de cámaras es un potente recurso y, es la herramienta más utilizada para reconocer entornos y objetos[30].

Existen varias configuraciones para obtener información visual del problema cinemático. Dos de ellas serían la configuración **endpoint open-loop** (EOL), que define aquellos sistemas que sólo observan el punto objetivo, el target, y la configuración **endpoint closed-loop** (ECL), que define aquellos sistemas que además de observar el target también recogen información del efector final. Los sistemas de tipo EOL siempre llevan una cámara acoplada en el efector final, por lo que la exactitud del posicionamiento está directamente ligado a la exactitud de la calibración ”mano-ojo”. Esto no ocurre con los sistemas ECL, en los que la cámara está situada de tal manera que capta tanto el target como el efector final. Por tanto, no dependen del error de calibración del sistema ”mano-ojo”.

En este proyecto la información visual será utilizada para diseñar un **control basado en posición**[31], en el que la información del modelo geométrico del sistema,

más las características extraídas de la imagen estimen de forma aceptable la posición y la orientación del efector final con respecto a la cámara[37].

Este tipo de sistemas dividen el control en dos partes: Por un lado estará el cálculo y la aplicación del feedback del sistema. Por el otro, estarán los problemas de estimación presentes en el cálculo de la posición a partir de la información visual obtenida.

Para llevar a acabo una correcta aproximación de la pose del efector final, los sistemas de control basados en posición calculan el error e existente entre la pose actual del efector, x_e, y la pose del target, x_t. Este error representa la restricción cinemática visual del sistema, y debe ser convenientemente minimizado $\rightarrow e = 0$. Reducir el error existente entre la pose del efector final y la del target implica mover la cadena cinemática a través de una trayectoria, hasta que el efector se posicione con error $e = 0$ sobre el target.

Esta solución no elimina los defectos de calibración entre el robot real y el modelo, ni las holguras de los motores, sin embargo tiene en cuenta estos factores a la hora de calcular los valores angulares del brazo. Así, cuando el brazo se mueva hacia la posición target, el control llevado a cabo con la información visual mitigará el error de posicionamiento.

5.3 Diseño general del sistema

Una vez descrito el entorno de programación, así como las principales herramientas y librerías utilizadas en el desarrollo de este proyecto, es el momento de introducir el nuevo sistema de cinemática inversa.

Para hacer frente a las carencias y defectos del antiguo sistema software, así como a los problemas descritos con anterioridad, se ha desarrollado una nueva estructura para la cinemática. Esta estructura se compone de tres módulos distintos de cinemática inversa[18]:

[37] Otro método de control es el basado en **la imagen**, donde los valores de control se calculan a partir de las mediciones y características de la imagen directamente. Esto reduce considerablemente el tiempo computacional ya que se elimina la fase de interpretación de la imagen y los cálculos de calibración y modelado.

- El primer módulo es el componente básico de cinemática inversa, *inversekinematicscomp* o **IK**. Contiene el algoritmo de Levenberg-Marquardt revisado y simplificado, así como todos los cálculos de más bajo nivel para obtener los valores angulares que lleven al efector final robótico a una posición target.

- El segundo módulo es, a grandes rasgos, un planificador de caminos para el efector final robótico, *ikgraphgeneratorcomp* o **IKG**. Este componente crea y almacena un grafo que representa las configuraciones libres del espacio o rango de trabajo del efector robótico, C_{free}. Se encarga de generar caminos libres de obstáculos y colisiones, simbolizados por los vértices del grafo, entre las distintas posiciones representadas por los nodos del grafo. Cada nodo del grafo almacena el espacio euclidiano de la posición y la configuración angular necesaria del brazo robótico para que el efector pueda alcanzarla.

- El tercer y último módulo es el encargado de proporcionar *feedback* al sistema de cinemática inversa. Es el *visualikcomp* o **VIK**. Este componente recibe información visual proporcionada por una cámara e intenta corregir los errores de posicionamiento del efector final mediante esa información. De cierta forma, este componente calibra la cadena cinemática del brazo robótico.

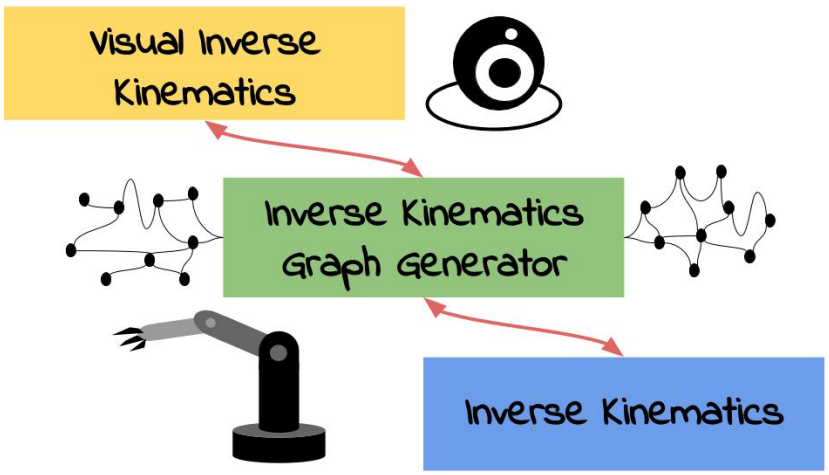


Figure 44: Esquema conceptual del nuevo sistema de cinemática inversa

Los tres módulos comparten la misma interfaz (*inversekinematics.idsl*), modificando sólo su comportamiento a la hora de manejar los targets que reciben. La comunicación entre los módulos es bidireccional: simplificando mucho el comportamiento real del sistema, los módulos de nivel superior van ”pasando” el target al módulo inferior para que éste realice sus cálculos y corrija la posición del efector final robótico. Una vez finalizados los cálculos, los módulos inferiores transmiten el resultado a los módulos superiores, para que realicen sus correcciones oportunas.

Antes de entrar en más detalle, es conveniente explicar ciertos módulos anexos al nuevo sistema de cinemática y sin los cuales la cinemática inversa jamás podría llevarse a cabo.

5.4 Componentes HAL

Denominamos **HAL** (*Hardware Abstraction Layer*) a aquellos elementos que funcionan como una capa o interfaz entre el software de alto nivel y el hardware de la máquina. Este tipo de elementos o componentes software proporcionan una plataforma consistente y sirven como mediadores entre el software de las aplicaciones y el nivel físico del computador.
Este proyecto cuenta con sus propios componentes HAL. En total tres componentes HAL organizados de la siguiente forma:

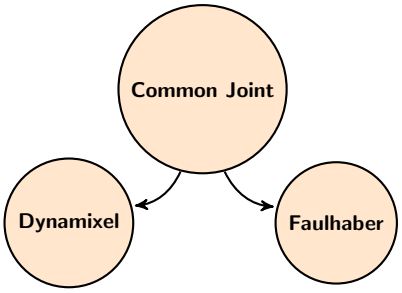


Figure 45: Componentes HAL del sistema

En el nivel más bajo nos encontramos con los componentes **DynamixelComp** y **FaulhaberComp**. Estos componentes son los encargados de manejar los motores

del robot. Leen el estado del hardware de los motores y permiten realizar consultas como el identificador del bus de los motores, el número de motores levantados, las restricciones impuestas por los ángulos mínimo y máximo de cada motor, la velocidad de giro o la posición cero del motor. También permiten cambiar el estado de los motores, modificando su valor angular o su velocidad de giro.

Por encima de estos componentes se encuentra **commonjoint**. Éste sirve como capa intermedia entre los componentes de más alto nivel y los componentes *DynamixelComp* y *FaulhaberComp*. Básicamente, unifica los componentes de bajo nivel de tal forma que cualquier otro componente que quiera mover el brazo no tendrá que conectarse al *DynamixelComp* y al *FaulhaberComp* mediante dos proxies, con un sólo proxy al *commonjoint* es suficiente, resultando mucho más fácil su uso y comprensión.

Además de comprobar el estado de los motores a través de *DynamixelComp* y *FaulhaberComp* y de actualizar el modelo interno del robot mediante los valores angulares obtenidos, *commonjoint* lleva a cabo una serie de comprobaciones sobre los motores para evitar las colisiones de los mismos con otros objetos.

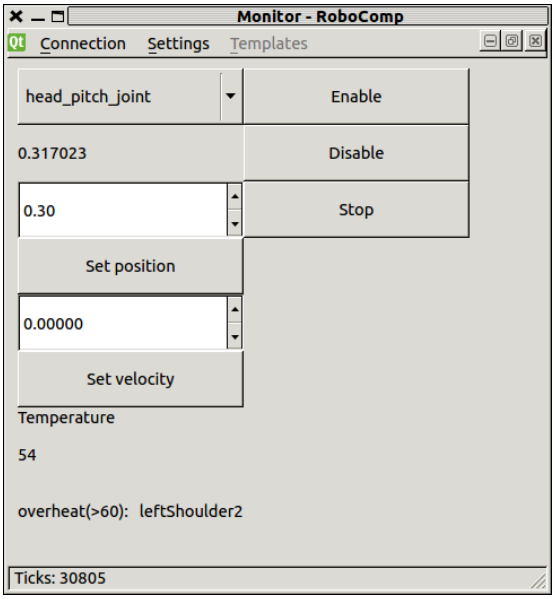


Figure 46: Vista de la aplicación RCMonitor

RoboComp cuenta con una herramienta bastante útil para manejar los motores del

robot (tanto los motores reales como los motores simulados por la herramienta *rcis*). Se trata del **RoboComp Monitor** (*RCMonitor*). Consiste en una interfaz de usuario muy sencilla, en la que aparecen listados los nombres de los motores[38], su velocidad de giro y su valor angular. Mediante las opciones de la interfaz podemos modificar tanto el valor angular del motor como su velocidad de giro. También podemos parar su movimiento, habilitarlos y des-habilitarlos.

El sistema de cinemática inversa hará uso de los componentes HAL para mover el efector final desde la posición de partida hasta la posición target, mediante el cálculo de los valores angulares de cada motor.

5.5 Componente básico de cinemática: IK

El nuevo sistema de cinemática inversa parte del antiguo componente software desarrollado en el Trabajo Fin de Grado *Cinemática inversa en robots Sociales*[8], denominado **inverseKinematicsComp**. Para este proyecto se ha tomado este componente y se le han aplicado una serie de modificaciones con el objetivo de mejorar su funcionamiento y para crear una base sólida sobre la que construir la nueva arquitectura software.

5.5.1 Diseño de la nueva interfaz

Comenzaremos describiendo la nueva interfaz diseñada, *InverseKinematics.idsl*, que ha reemplazado a la antigua *BodyInverseKinematics.idsl*. Esta interfaz es compartida por los tres componentes que forman el nuevo sistema de cinemática inversa, IK, GIK y VIK, y define una serie de estructuras de datos a utilizar por los diferentes métodos implementados.

Los elementos de la interfaz aparecen en el fichero, *InverseKinematics.idsl* en el siguiente orden:

[38] Si conectamos el *RCMonitor* al componente *DynamixelComp*, la aplicación sólo listará aquellos motores que sean Dynamixel. Lo mismo pasaría si lo conectamos al *FaulhaberComp*. Sin embargo, si conectamos el *RCMonitor* al *commonjoint* aparecerán ambos tipos de motores.

1. En primer lugar se define la excepción *IKException*, que se lanzará cuando los componentes cinemáticos encuentren algún problema o no puedan resolver el target.

2. También se define la estructura *Pose6D*. Al igual que en la interfaz antigua, esta estructura contiene las seis componentes que definen una pose en el espacio, $[x, y, z, rx, ry, rz]$.

3. Se mantiene la estructura *WeightVector*, que contiene los factores de escala de las componentes de error, es decir, los valores $[x, y, z, rx, ry, rz]$ que formarán la matriz de peso W_e del algoritmo de Levenberg-Marquardt.

4. La estructura *Axis* no ha sufrido ningún cambio en esta nueva versión, almacenando los tres ejes del espacio euclidiano $[x, y, z]$.

5. Se añade una nueva estructura, *Motor*, que contiene el nombre del motor, *name* y su valor angular asociado, *angle*. Con esta estructura se crea una lista de motores, *MotorList*.

6. La estructura *TargetState* ha sido modificada. A sus tres valores originales, *finish* (flag que indica si el componente IK ha terminado de ejecutar el target), *state* (estado final del target) y *elapsedTime* (tiempo de ejecución en el componente), se han añadido *errorT* (indica el error de traslación final alcanzado), *errorR* (indica el error de rotación final alcanzado) y la lista de motores *motors* (almacena la cadena cinemática del robot). La ampliación de esta estructura persigue el objetivo de devolver toda la información posible generada por el componente IK, desde el estado de finalización hasta los valores angulares calculados.

En cuanto a los métodos de la interfaz se han mantenido sin muchos cambios:

- *TargetState* **getTargetState** (*string bodyPart, int targetID*): devuelve el estado del target con identificador *targetID* (añadido) perteneciente a la parte del cuerpo *bodyPart*.

- *int* **setTargetPose6D** (*string bodyPart, Pose6D target, WeightVector weights*): establece un target objetivo de tipo **Pose6D** para una parte *bodyPart*[39] del cuerpo del robot, con un vector de pesos asociado *weights*, que atenúe o no sus componentes de traslación y rotación. En esta versión del software, este método devuelve el identificador numérico del target.

- *int* **setTargetAlignaxis** (*string bodyPart, Pose6D target, Axis ax*): establece un target de tipo **ALINGAXIS** a la parte *bodyPart* del robot para que la cinemática oriente el efector final correspondiente alineándolo con el eje que *ax* le indique. Devuelve el identificador numérico del target

- *int* **setTargetAdvanceAxis** (*string bodyPart, Axis ax, float dist*): este método recibe targets del tipo **ADVANCEAXIS**. Se encarga de que el efector de la parte *bodyPart* avance una distancia *dist* a lo largo del vector *ax* que se le indique. También devuelve el identificador numérico del target

- *bool* **getPartState** (*string bodyPart*): este método indica si la parte *bodyPart* del cuerpo tiene asignados targets a resolver.

- *void* **goHome** (*string bodyPart*): este método lleva la parte del robot que se le indique a una posición de inicio o home.

- *void* **stop** (*string bodyPart*): este método aborta la ejecución del algoritmo de Levenberg-Marquardt. Es un método de seguridad cuyo objetivo es detener el procesamiento y la ejecución de un target o de una lista de targets.

- *void* **setJoint** (*string joint, float angle, float maxSpeed*): este método modifica el valor angular de un joint determinado.

- *void* **setFingers** (*float d*): este método se encarga de abrir y cerrar las pinzas de las manos del robot, dejando entre ambas una distancia *d*.

[39] RIGHTARM, LEFTARM o HEAD

5.5.2 Funcionamiento del componente IK

Una vez vista la interfaz, podemos definir el comportamiento del componente IK. Este componente, además de las clases *SpecificWorker* (que implementa los métodos de *InverseKinematics.idsl*) y *SpecificMonitor* (ver imagen 37), cuenta con otras tres clases:

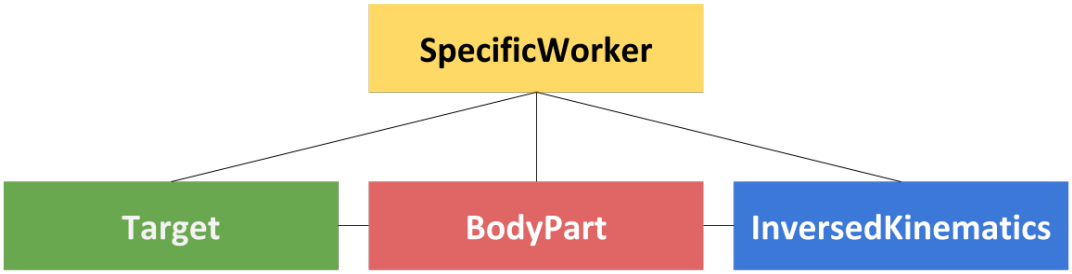


Figure 47: El componente IK cuenta con tres clases auxiliares: a) *Target* representa las poses objetivos, b) *BodyPart* representa una cadena cinemática del cuerpo del robot, y c) *InversedKinematics* contiene toda la lógica del algoritmo de Levenberg-Marquardt.

Clase *Target* Esta clase se encarga de definir los atributos y métodos más importantes de un punto objetivo o punto de destino. Al igual que en el software de cinemática antiguo, en esta nueva versión se mantienen los tres tipos de targets (*TargetType*): los targets *POSE6D*, los targets *ALINGAXIS* y los targets *ADVANCEAXIS* (ver sección 3.2.3).

Los targets *POSE6D* y *ALINGAXIS* tienen asociado un vector 6D que recoge las coordenadas de traslación y rotación que definen a la posición objetivo (*pose*), así como un vector de pesos para cada componente de la *pose* (*weight*). Además, los *ALINGAXIS* cuentan con un vector 3D que marca el eje de rotación con el que el efector final debe alinearse (*axis*). Luego tenemos el target *ADVANCE AXIS* que cuenta solamente con un vector director que marca la recta por la que el efector final debe avanzar (*axis*) y la longitud del paso o distancia que debe avanzar sobre dicha recta (*step*).

Todos los targets se caracterizan por tener un identificador numérico que los distingue unos de otros (*identifier*) dentro del *BodyPart* que les corresponda. También cuentan

con un estado (*TargetState*) que indica si están a la espera de ser ejecutados por el algoritmo de Levenberg-Marquardt (*IDLE*), si están siendo ejecutados en ese preciso momento (*IN_PROCESS*) o si han sido ya resueltos (*FINISH*).
Una vez que el target alcanza el estado *FINISH*, se indica el resultado final alcanzado por el algoritmo de Levenberg-Marquardt en su resolución (*TargetFinalState*), que puede ser *LOW_ERROR* cuando el algoritmo ha alcanzado una solución que satisface la ecuación de error, *KMAX* cuando el algoritmo agota todas las iteraciones que puede ejecutar, *LOW_INCS* cuando el algoritmo se estanca en un mínimo local y no puede descender por la función de error, o *NAN_INCS* cuando el algoritmo no puede calcular una nueva configuración angular para el *BodyPart* correspondiente.
También, cuando el target ha sido ejecutado, se le asigna un vector 6D que recoge el error en traslación y rotación alcanzado entre el efector final y la posición objetivo (*errorvector*[40]), el vector de configuración del *BodyPart* al que pertenece el target (*finalangles*) que contiene los ángulos de cada motor de la cadena cinemática con la que el efector final puede alcanzar la posición objetivo, y el tiempo que ha tardado el algoritmo en computar el target (*runTime*).
En cuanto a los métodos de esta clase, se reducen a los típicos *seters & geters*, es decir, métodos que consultan o modifican el valor de los atributos antes vistos.

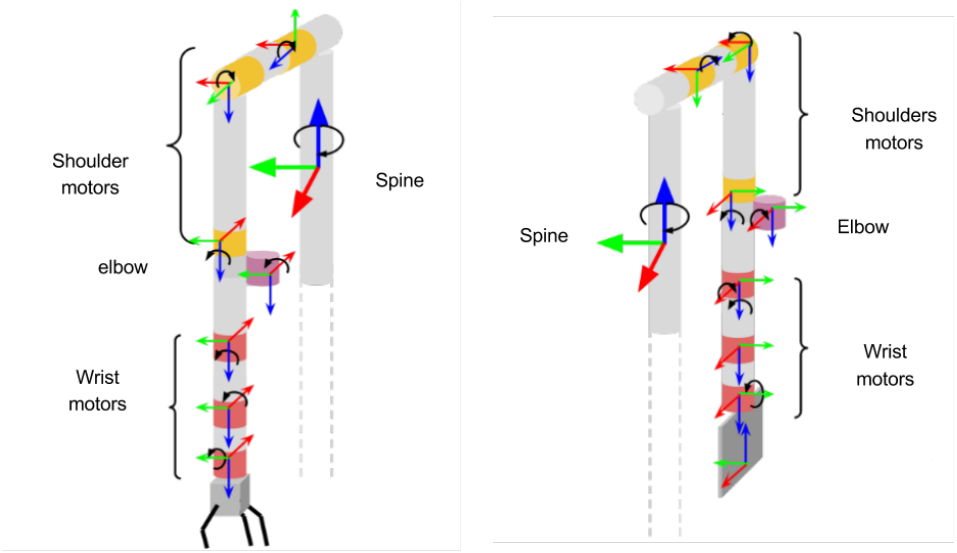
Clase *BodyPart* Convencionalmente, hemos dividido la estructura de Shelly en tres partes:

1. El brazo derecho: denominado RIGHTARM. Esta parte guarda la lista de joints que forman la cadena cinemática del brazo derecho del robot (*rightShoulder1*, *rightShoulder2*, *rightShoulder3*, *rightElbow*, *rightForeArm*, *rightWrist1* y *rightWrist2*), así como su efector final (*grabPositionHandR*).

2. El brazo izquierdo: llamado LEFTARM, el cual cuenta con su propia lista de joints, formada por los motores del brazo izquierdo del robot (*leftShoulder1*,

[40] Aunque se trata de un vector 6D para los tres tipos de targets, no todos ellos lo utilizan de igual forma. Por ejemplo, para los targets de tipo *ALINGAXIS* sólo se marcarán las componentes de rotación y no las de traslación.

leftShoulder2, *leftShoulder3*, *leftElbow*, *leftForeArm*), y su correspondiente efector final (*grabPositionHandL*).



(a) Brazo derecho de Shelly (b) Brazo izquierdo de Shelly

Figure 48: Cadena cinemática de los dos brazos de Shelly

3. La cabeza: etiquetada como HEAD. Al igual que las anteriores partes, cuenta con su lista de joints que forman el cuello del robot (*head_yaw_joint* y *head_pitch_joint*) y el tip o efector final situado en la cabeza del robot (*rgbd_transform*).

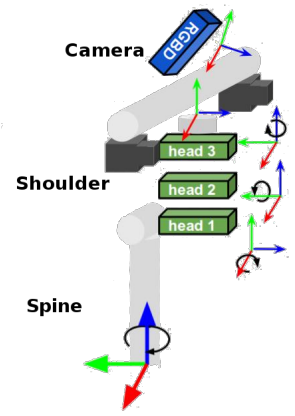


Figure 49: Cabeza de Shelly

La clase *BodyPart* representa estas partes del robot. Para saber qué parte del robot codifica cuenta con un identificador que lo define como RIGHTARM, LEFTARM o HEAD (*partName*). Almacena en la lista *motorList* el nombre de todos los joints de la cadena cinemática que lo compone, y en la variable *tipName*, el nombre del efector final. También dispone de un contador de targets (*counter*) y la lista de targets que debe alcanzar su efector final (*targetList*).
Al igual que la clase *Target*, *BodyPart* sólo cuenta con los métodos típicos *seters & geters*, para consultar y modificar sus atributos y gestionar la lista de targets.

Clase *InversedKinematics* Esta clase contiene toda la lógica del algoritmo Levenberg-Marquardt. Su funcionamiento es el siguiente:

- A través del método *void* **solveTarget** *(BodyPart *bodypart_, InnerModel *innermodel_)*, la clase recibe una parte corporal de Shelly junto con la representación interna del estado actual del robot. En este método, la clase *InversedKinematics* obtiene el target que debe resolver consultando la lista de targets del *BodyPart* y lo envía al método *void* **levenbergMarquardt** *(Target &target)*

- El método *void* **levenbergMarquardt** *(Target &target)* recibe el target y se encarga de resolverlo siguiendo la estructura vista en la sección 3.1.3[14]. Para ello se vale de cuatro métodos muy importantes:

    1. El método *QVec* **computeAngles** *()* le devolverá la lista con los valores angulares de cada motor de la cadena cinemática. Para ello consulta la representación interna del robot, *innermodel*, actualizada en todo momento.

    2. El método *void* **updateAngles** *(QVec new_angles)*, que irá actualizando el modelo interno del robot con cada incremento calculado por Levenberg-Marquardt.

3. El método *QMat* **jacobian** *(QVec motors)* calculará la matriz jacobiana necesaria para el correcto funcionamiento del algoritmo. Para ello introduce las mejoras del software antiguo de cinemática inversa, a saber, el bloqueo de motores gracias al método *bool* **outLimits** *(QVec &angles, QVec &motors)* y el cálculo del jacobiano para motores prismáticos.

4. El método *QVec* **computeErrorVector** *(Target &target)*, que calculará el vector de error del target. Para su correcto funcionamiento, este método consulta el tipo de target que está manejando el algoritmo y aplica la función de error que le corresponda. Por ejemplo, para los targets *ALINGAXIS* aplicará la **Fórmula de Rodrigues** aplicada a rotaciones[32], mientras que para los tipos *POSE6D* y *ADVANCEAXIS* calculará el error obteniendo la pose del efector final en el sistema de referencia del target.

- Una vez finalizado el método *void* **levenbergMarquardt** *(Target &target)*, la clase *InversedKinematics* llamará al método *bool* **deleteTarget** *()*, que se encarga de comprobar el tiempo de cómputo que ha supuesto resolver el target y el umbral de error alcanzado. Esto servirá para: 1) eliminar el target si se supera el tiempo máximo de ejecución o si se alcanza un error aceptable, 2) repetir la ejecución del target si el error es alto y si aún no se ha superado el tiempo máximo de ejecución.

El funcionamiento del método *void* **levenbergMarquardt** *(Target &target)* lo desarrollaremos más profundamente en la siguiente sección.

Clase principal *SpecificWorker* Esta clase inicializa y almacena las tres partes del robot en un mapa clave-valor (*bodyParts*) tal que la clave será el nombre de la parte y el valor una instancia de *BodyPart*. Cada *bodyPart* es inicializado con la información almacenada en el fichero de configuración con el que ejecutamos el componente:

```
CommonBehavior.Endpoints=tcp -p 12207
InverseKinematics.Endpoints=tcp -p 10240
#-----------------------------------------------------------
# Fichero modelo interno del robot: innermodel
```

```
#------------------------------------------------------------
InnerModel=/home/robocomp/robocomp/components/robocomp-shelly/etcSim/shelly.xml
#------------------------------------------------------------
#------------------------------------------------------------
# Sistema de cadenas cinematicas
#------------------------------------------------------------
RIGHTARM=rightShoulder1;rightShoulder2;rightShoulder3;rightElbow;rightForeArm;
        rightWrist1;rightWrist2
RIGHTTIP=grabPositionHandR

LEFTARM=leftShoulder1;leftShoulder2;leftShoulder3;leftElbow;leftForeArm
LEFTARM=leftShoulder1;leftShoulder2;leftShoulder3;leftElbow;leftForeArm;
       leftWrist1;leftWrist2
LEFTTIP=grabPositionHandL

HEAD=head_yaw_joint;head_pitch_joint
HEADTIP=rgbd_transform
#------------------------------------------------------------

#------------------------------------------------------------
# COMPONENTES/INTERFACES REQUERIDAS
#------------------------------------------------------------
# Proxies for required interfaces
JointMotorProxy = jointmotor:tcp -h localhost -p 20000
#------------------------------------------------------------

This property is used by the clients to connect to IceStorm.
TopicManager.Proxy=IceStorm/TopicManager:default -p 9999

Ice.Warn.Connections=0
Ice.Trace.Network=0
Ice.Trace.Protocol=0
Ice.ACM.Client=10
Ice.ACM.Server=10
```

Esto supone una gran mejora con respecto al software anterior, en el que las cadenas cinemáticas estaban *hardcodeadas* dentro del antiguo componente IK, limitando su uso al robot Shelly. Con esta mejora basta sólo con cambiar el fichero de configuración para poder utilizar el nuevo software componente de cinemática con cualquier cadena cinemática.

SpecificWorker también almacena el modelo interno del robot, *innermodel*, con el que la cinemática inversa (*inversedkinematic*) hará todos los cálculos necesarios.

El funcionamiento de esta clase es fácil de comprender, aunque su codificación entraña ciertas dificultades. Se trata de un bucle infinito que comprueba periódicamente la existencia de algún target por resolver en alguna parte del cuerpo de Shelly. En

el momento en que exista un target con estado IDLE en un *bodyPart*, la clase *SpecificWorker* lo envía a *inversedkinematic* para que lo resuelva. Una vez que *inversedkinematic* termina de ejecutar el target, éste es almacenado en una lista de targets resueltos (*targetsSolved*), para poder acceder a él y consultarlo en el momento que sea necesario.

Los targets llegarán por los tres métodos de entrada de la interfaz, y serán automáticamente almacenados con estado IDLE dentro de la lista de targets de cada *bodyPart*, devolviendo cada método el identificador del target para que los componentes de nivel superior puedan acceder a su estado cuando lo necesiten.

Una vez repasado el funcionamiento del componente IK, vamos a introducir las nuevas mejoras que se han implementado en esta última versión de la cinemática inversa.

5.5.3 Refactorización de Levenberg-Marquardt

El nuevo componente de cinemática inversa, IK, mantiene el algoritmo de Levenberg-Marquardt con la misma estructura que la versión anterior. Pero en este caso todos los umbrales ε han sido eliminados.

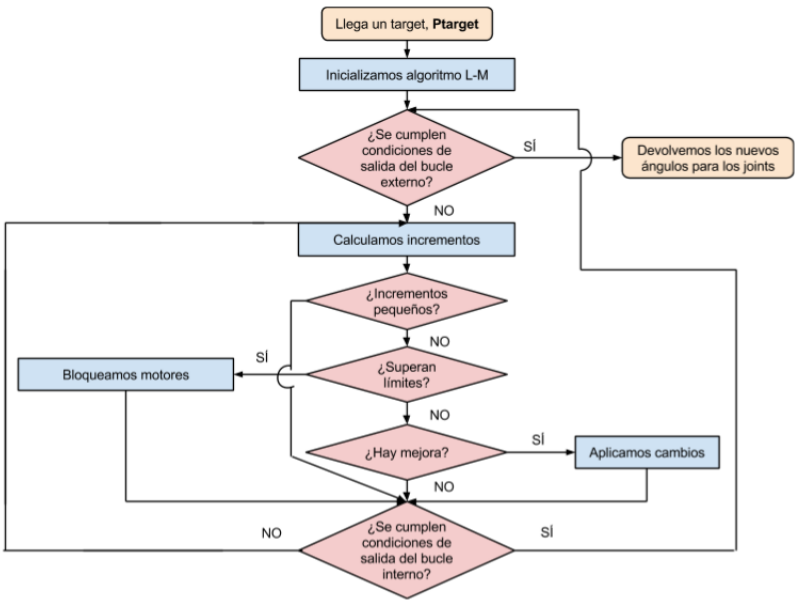


Figure 50: Diagrama de flujo general del algoritmo de Levenberg-Marquardt

En la estructura se mantienen las tres adaptaciones realizadas en el anterior software de cinemática: la matriz de pesos, el bloqueo de motores y el cálculo de la matriz Jacobiana para motores prismáticos y no prismáticos. Al igual que en la primera implementación, la estructura del algoritmo de Levenberg-Marquardt se divide en varias fases:

1. **Fase de inicialización**: en esta fase se inicializan las distintas variables del algoritmo, el parámetro de amortiguación μ, el vector de motores bloqueados L_b, el vector de ángulos iniciales x_0, el vector de error inicial e, las matrices Jacobiana y Hessiana, J y H, y el descenso de gradiente g. Los flags de parada se inicializan a falso. Las únicas constantes serán la matriz identidad I, la matriz de pesos W_e y el número máximo de iteraciones K_{max}, eliminándose, como dijimos, los distintos umbrales ε.

2. **Fase de comprobación del bucle externo**: se trata del primer bucle *while*, donde se comprueba si se ha superado el número de iteraciones (**KMAX**) si el error alcanzado es bajo (**LOW_ERROR**), si los incrementos que se calculan son despreciables (**LOW_INC**) o si directamente son nulos (**NAN_INC**). En cualquiera de los cuatro casos, el algoritmo de Levenberg-Marquardt finaliza.

3. **Cálculo de los incrementos**: dentro del bucle interno *do-while* se calculan los incrementos Δx de los ángulos y se llevan a cabo una serie de controles.

   - Que los incrementos calculados no sean nulos.
   - Que los incrementos calculados no sean despreciables.
   - Si los incrementos son aceptables se suman a los valores angulares de la cadena cinemática y se comprueba que no se superen las restricciones de los motores. Si se superan, se bloquea el motor y se recalculan las matrices Jacobiana y Hessiana J y H.

4. **Aceptación o rechazo de los incrementos**: una vez sumados los incrementos se comprueba si el error desciende o aumenta. Si desciende, se desbloquean

los motores y se aceptan los cambios, por lo que hay que recalcular J, H y g. Si no desciende, los cambios no son aplicados y se aumenta el parámetro de amortiguación *mu*.

5. **Fase final**: una vez que termina el algoritmo de Levenberg-Marquardt, se marca el target con el estado *FINISH* y se almacena, junto a su estado de finalización (LOW_ERRORS, LOW_INC, NAN_IC o KMAX), su vector de error final y los ángulos calculados.

5.5.4 División y repetición del target

Eliminadas las dependencias causadas por los distintos umbrales ε, se observó que el algoritmo tenía tendencia a corregir antes y mejor la rotación que la traslación en los targets de tipo Pose6D, lo que ralentizaba el movimiento del brazo robótico a la par que aumentaba el error de traslación.

Para evitar ese efecto se añadió un sistema de repetición de targets. Ese sistema consiste en que, cuando llega un target T a través del método *setTargetPose6D* con restricciones de rotación, el algoritmo añade dos *subtargets* a la lista de posiciones de destino del *bodyPart*, T_t y T_r. El primero, T_t sólo tendrá restricción en su traslación, mientras que el segundo, T_r, tendrá ambas restricciones, la de rotación y la de traslación.

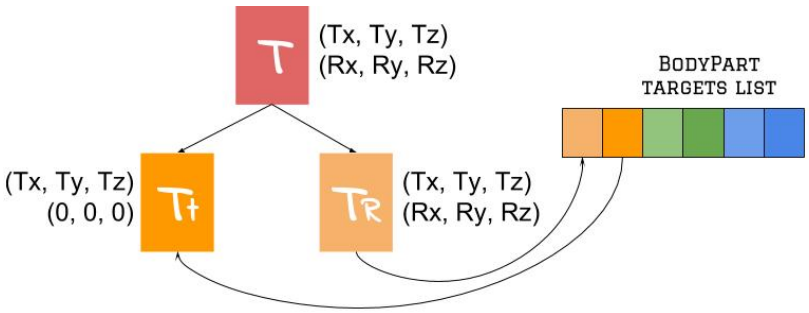


Figure 51: Método de repetición de targets

Con este método, el componente de IK nuevo realiza una primera aproximación hacia la pose objetivo sin restricciones de ángulos, lo que permite un movimiento más

rápido y fluido, de tal manera que, cuando se ejecute el segundo *subtarget* el efector final estará en una posición muy cercana y apenas tendrá que corregir la rotación.

5.5.5 Sistema de referencia. Longitud v.s. ángulos

Un problema que presentaba el antiguo software de cinemática era el cambio de unidad de medida de milímetros a metros. Esto suponía un inconveniente, ya que RoboComp utiliza el milímetro como unidad de medida de longitud, por lo que, internamente en el componente, había que pasar todo el modelo interno de milímetros a metros. En esta nueva versión se ha modificado el funcionamiento del método de Levenberg-Marquardt para que trabaje con milímetros, adaptando los umbrales que el algoritmo utiliza.

Otra dificultad era que no existía una forma de relacionar los metros con los radianes en cuanto a peso e importancia, por lo que la matriz de pesos W_e, que en un principio era binaria (0 y 1 en la diagonal) para eliminar ciertas componentes del error, tuvo que ser utilizada como factor de compensación entre metros y radianes.
Este problema deriva directamente del cálculo de la función de error que utiliza Levenberg-Marquardt: $e = P_{target} - F(x)$. Mientras que el máximo error de rotación alcanzaba los 2π radianes, los errores en traslación apenas llegaban a los centímetros. Esto se traducía en que las rotaciones tenían mayor peso en el algoritmo que las traslaciones[41]. Para solucionar esta situación, se ha añadido un factor que compensa y hace compatible la unidad de longitud con los radianes. Según [21] este factor se calcularía como:

$$f = \frac{360}{2\pi \cdot d}$$

[41] En el caso de la traslación, al estar el sistema de referencia en metros, los errores obtenidos eran del orden de centímetros, $e \cdot 10^{-2}m$, o incluso milímetros $e \cdot 10^{-3}m$, demasiado pequeños comparados con la magnitud del error de rotación, $e \cdot 1rad$.
Al cambiar el sistema a milímetros, se daba justo el caso contrario: los errores de traslación eran de milímetros, $e \cdot 1mm$, o centímetros $e \cdot 10mm$, más grandes que los errores de rotación. El algoritmo corregía la traslación a costa de empeorar la rotación.

Donde d es el ángulo en grados y f el factor de corrección entre grados y metros. En el caso de RoboComp, al usar radianes y milímetros la ecuación quedaría como:

$$f = \frac{1000}{\frac{360}{2 \cdot \pi \cdot \frac{180 \cdot e}{\pi}}};$$

Donde e es el error de traslación entre el efector y el target.
Esta mejora más las dos primeras modificaciones del componente IK solventaron los problemas derivados del algoritmo de Levenberg-Marquardt: la dependencia de los umbrales y la tendencia a corregir la rotación a cambio de sacrificar la traslación. Sin embargo, esto aún es insuficiente para resolver los problemas de calibración y de holguras que presenta el robot social Shelly.

5.6 Graph of free C-Space: GIK

En el primer sistema de cinemática, el componente IK era el encargado de:

1. Ejecutar el algoritmo de Levenberg-Marquardt, para calcular la configuración óptima de los motores que permitiera alcanzar el target.

2. Mover los motores de Shelly, enviando al *commonjoint* todos los incrementos calculados por Levenberg-Marquardt, Δx, consumiendo mucho tiempo de cómputo y exponiendo el brazo a posibles colisiones cuando el componente IK fallaba en el cálculo de los incrementos.

Una segunda aproximación eliminó los movimientos innecesarios de cada incremento calculado, Δx, de tal forma que el brazo robótico sólo se movía cuando el algoritmo de Levenberg-Marquardt había terminado de calcular los ángulos finales, x^{final}. Esto ahorraba tiempo de cómputo y evitaba el sobrecalentamiento de los motores.
Sin embargo, debido a las holguras del brazo, la posición inicial de partida afectaba mucho al algoritmo. Se daba el caso de que las distintas soluciones para un mismo target con diferentes punto de partida, fluctuaban entre un error mínimo y un error máximo, lo que lo hacía inviable.

Para evitar estas fluctuaciones se diseñó el segundo módulo de cinemática inversa, el *ikGraphGenerator* o **GIK**. La idea inicial de este componente era evitar, en la medida de lo posible, el uso de la IK básica, de tal forma que el brazo robótico pudiera moverse a través de posiciones conocidas dentro de su rango de trabajo utilizando cinemática directa. El planteamiento es sencillo: el GIK mueve el efector final a través de posiciones previamente calculadas y alcanzables, que lo vayan acercando a la posición objetivo, para que sólo al final, cuando el efector esté lo más cerca posible del target, se ejecute la IK, aumentando así el porcentaje de aciertos, la robustez del sistema y disminuyendo el tiempo de cómputo.

Con este planteamiento, el componente GIK genera una malla 3D virtual de posiciones dentro del volumen espacial en el que se mueve el brazo robótico. Cada nodo de la malla codifica una posición euclidiana 6D (las tres componentes de traslación más las tres componentes de rotación) del efector final robótico, junto al correspondiente conjunto de valores angulares de la cadena cinemática.

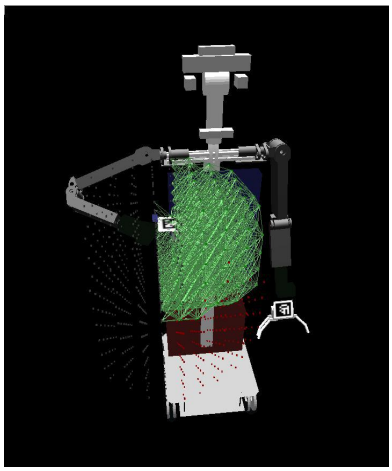


Figure 52: Malla 3D generada por el GIK

El componente GIK tiene pues dos modos de ejecución:

1. El primer modo es crear y almacenar en un fichero, la malla o grafo de posiciones (graph of free C-Space, llamado *ConnectivityGraph*). Lo primero que hace GIK es generar una serie de puntos contenidos en el cubo que forma el espacio de

trabajo del efector final, formando una malla regular cúbica. Cada punto será un nodo del *ConnectivityGraph*, que almacenará:

- un identificador numérico (*id*),
- la posición del target (*pose*),
- la posición del codo del robot (*poseElbow*),
- su validez (*valid*),
- si está libre en el espacio o está ocupado por algún objeto externo al robot (*state*),
- y la configuración del brazo del robot (*configurations*).

2. El segundo modo sólo podrá ejecutarse si antes se ha generado un *ConnectivityGraph*. En este caso, cuando el componente GIK reciba un target *POSE6D* a través del correspondiente método de la interfaz, éste calculará primero el camino óptimo y libre de colisiones que una la posición inicial del efector con la posición más cercana al target, mediante el **algoritmo de Dijkstra**. Después, cuando haya terminado de mover el efector a través del grafo, llamará al componente IK[18] y ejecutará los ángulos finales que éste le devuelva.

   Para los otros dos tipos de targets, llama directamente al componente IK y ejecuta los ángulos que éste último calcule.

Algorithm 3 Búsqueda del camino óptimo

procedure PATH PLANNING
  $r \leftarrow$ Closest(effector, G_B)
  $rg \leftarrow$ Plan(effector, r)
  $t \leftarrow$ Closest(target, G_B)
  $tg \leftarrow$ Plan(target, t)
  $path \leftarrow$ Dijkstra(r, t, G_B)
  $path \leftarrow$ Smooth(rg, path, tg)
end procedure

De esta forma, el algoritmo de IK sólo se empleará cuando el efector esté muy cerca. Una vez que la IK calcule los ángulos para la cadena cinemática, el GIK los tomará y

los enviará al componente *commonjoint* para que los ejecute. Así, la IK básica ya no se preocupa de mover los motores, lo que permite reducir el código y simplificar las secuencias de control.

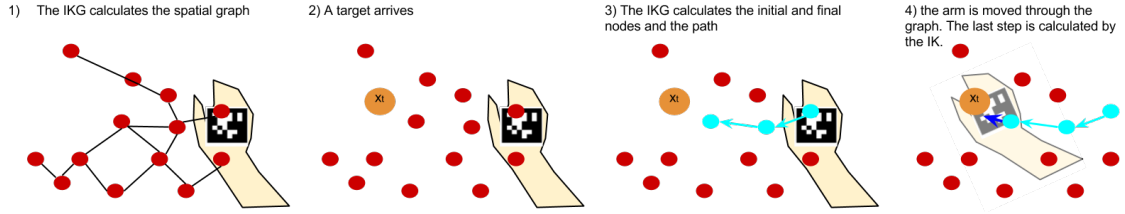


Figure 53: Movimiento del efector final a través de la malla

Aprovechando la existencia de la malla 3D generada por el GIK, se ha implementado dentro del componente un método para detectar y evitar colisiones con objetos externos al robot Shelly: *bool* **delete_collision_points** *()*. Este método se basa en la **librería PCL**[42] para crear las estructuras con las que detectar colisiones.
El componente está conectado a la cámara RGBD del robot, de la que obtiene su nube de puntos (*point_cloud*). Ésta contiene las coordenadas espaciales de los objetos que la cámara ve:

```
RoboCompRGBD::PointSeq point_cloud;
RoboCompJointMotor::MotorStateMap hState;
RoboCompDifferentialRobot::TBaseState bState;

rgbd_proxy->getXYZ(point_cloud, hState, bState);
...
```

GIK diezma la muestra (por cada 10 puntos toma 1) y la filtra, pasand los puntos del sistema de referencia de la cámara, al sistema de referencia del robot. Por último, se filtran guardando en la nube de puntos PCL *full_cloud* sólo aquellos puntos que se encuentren dentro del rango de trabajo del efector final. Para hacer este procedimiento más rápido, se paraleliza el bucle:

```
...
QVec center = QVec::vec3(0, 900, 640);
xrange = std::pair<float, float>( center(0)-200, center(0)+200 + 1);
yrange = std::pair<float, float>( center(1)-200, center(1)+200 + 1);
zrange = std::pair<float, float>( center(2)-200, center(2)+200 + 1);
```

[42]Documentación y descarga disponible en http://pointclouds.org/

```
QMat mat = innerModel->getTransformationMatrix("robot", "rgbd");
int32_t usedPoints = 0;

int num_hilos = 3;
omp_set_num_threads(num_hilos);

#pragma omp parallel for shared(usedPoints) ordered schedule(static,1)
for (uint32_t i=0; i<point_cloud.size(); i=i+10)
{
        QVec v = (mat * QVec::vec4(point_cloud[i].x, point_cloud[i].y,
                                                    point_cloud[i].z, point_cloud[i].w)
                                                    ).fromHomogeneousCoordinates();
        if (v(0)>=xrange.first and v(0)<=xrange.second and
                v(1)>=yrange.first and v(1)<=yrange.second and
            v(2)>=zrange.first and v(2)<=zrange.second)
        {
                #pragma omp ordered
                {
                        full_cloud->points[usedPoints].x = v(0);
                        full_cloud->points[usedPoints].y = v(1);
                        full_cloud->points[usedPoints].z = v(2);
                        usedPoints++;
                }
        }
}
full_cloud->width = usedPoints;
full_cloud->height = 1;
full_cloud->points.resize(usedPoints);
```

Si se han detectado puntos dentro del rango de trabajo del brazo sólo puede significar dos cosas: 1) que la cámara está viendo un objeto externo al robot, y por lo tanto, es un obstáculo a evitar, 2) que la cámara está viendo el brazo del propio robot, por lo que hay que ignorarlos.

Para detectar estas dos situaciones, GIK hace un segundo filtrado de la nube de puntos *full_cloud*, con el objetivo de quitar densidad, y la pasa a la estructura KdTree de PCL:

```
if (usedPoints>0)
{
        pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
        sor.setInputCloud (full_cloud);
        sor.setMeanK (50);
        sor.setStddevMulThresh (1.0);
        sor.filter (*cloud_filtered);

        pcl::KdTreeFLANN<pcl::PointXYZ> kdtree;
        kdtree.setInputCloud (cloud_filtered);
        ...
```

Por último, va recorriendo todo el grafo, comparando cada nodo con los puntos

almacenados en el *kdtree*. Para comprobar que no sea el propio brazo el que esté obstaculizando la visión de la cámara, utiliza las colisiones del modelo interno del robot, *innermodel*, de tal forma que, si el punto detectado por la cámara se corresponde con una posición del brazo, será inmediatamente descartado, mientras que si el punto no pertenece al robot será porque pertenece a un objeto externo, y por lo tanto se marca como bloqueado.

```
...
pcl::PointXYZ searchPoint;
std::vector<int> pointIdxRadiusSearch;
std::vector<float> pointRadiusSquaredDistance;

int free_nodes=0;
bool free;

for (uint i=0; i<graph->vertices.size(); i++)
{
        if (graph->vertices[i].valid)
        {
                free = true;

                searchPoint.x = graph->vertices[i].pose[0];
                searchPoint.y = graph->vertices[i].pose[1];
                searchPoint.z = graph->vertices[i].pose[2];

                if(kdtree.radiusSearch(searchPoint,
                                       collision_threshold,
                                       pointIdxRadiusSearch,
                                       pointRadiusSquaredDistance)>0)
                {
                        innerModel->updateTransformValues("my_mesh",
                                                          searchPoint.x,
                                                          searchPoint.y,
                                                          searchPoint.z,
                                                          0,0,0, "root");
                        for (auto mesh: meshes)
                        {
                                if (!innerModel->collide(a, "my_mesh"))
                                {
                                        free = false;
                                        break;
                                }
                        }
                }
                if(free)
                {
                        graph->vertices[i].state=
                                ConnectivityGraph::VertexState::FREE_NODE;
                        free_nodes++;
```

```
                        }
                        else
                                graph->vertices[i].state=
                                        ConnectivityGraph::VertexState::LOCKED_NODE;

                        paintNode(free, i);
                }
        }
}
```

Al marcar el nodo como LOCKED_NODE, el GIK lo descarta a la hora de calcular el camino con Dijkstra, evitando entonces que el efector final choque conra el objeto. Esta ampliación ha sido probada en el nuevo brazo de Shelly, como veremos en el apartado de Resultados y Discusión. Por supuesto tiene sus limitaciones, como por ejemplo: *¿Qué pasa cuando el brazo tapa al obstáculo?*. En este caso, el algoritmo marcará como FREE_NODE aquellos nodos que coincidan con el brazo, por lo que los objetos que estén tapados no se tendrán en cuenta y el efector podrá chocar contra ellos, cuando GIK calcule el camino.

5.7 Sistema de realimentación visual: VIK

Hasta este momento, los dos módulos de cinemática inversa estudiados han basado su funcionamiento en el feedback que les proporciona los encoders de los motores[43]. Pero esta información no será suficiente si existen problemas de calibración y holguras en el sistema robótico.

Precisamente y para solucionar este problema se ha propuesto esta estrategia en dos etapas. La primera etapa esta compuesta por el IK y el GIK. En ella el brazo robótico se mueve a través del path calculado por el GIK hasta la posición de destino mediante la ayuda del IK. En esta etapa el sistema se basa exclusivamente en la información que ofrecen los joints de la cadena. Sin embargo, en la segunda etapa, se cierra el bucle mediante la información visual adquirida por una cámara. Es ésta última etapa la que puede eliminar los errores producidos por la mala calibración y las holguras de

[43]El GIK también utiliza información visual para detectar objetos y podar su grafo. Pero su objetivo es evitar posibles colisiones, no comprueba si el efector final se está acercando de forma correcta al target.

la estructura hardware[18].

En este proyecto se han utilizado un dispositivo Asus Prime Sense colocado sobre la cabeza del robot Shelly, y una marca AprilTag[33][44] colocada en el efector final del robot. Para sacar el máximo partido a este tipo de marcas se utilizará el componente **apriltagscomp**, que, mediante el método de publicación-suscripción, envía a los componentes suscritos a él la siguiente información:

- El vector de traslación desde la cámara hasta el punto central de la marca (el cero del sistema de referencia de AprilTag).
- Los tres ángulos de Euler que codifican la orientación del sistema de referencia de la marca con respecto al sistema de referencia de la cámara.

A este componente se conectará el tercer módulo de cinemática inversa del sistema, *VisualIK* (**VIK**). Este último componente será el encargado de aplicar la realimentación visual que ofrece la cámara dentro del sistema de cinemática inversa propuesto.

Antes de explicar el funcionamiento del software VIK, debemos tener claros tres conceptos fundamentales:

- p_v: es la pose $[tx, ty, tz, rx, ry, rz]$ del efector final dada por el feedback visual que proporciona la cámara. Gracias a *apriltagscomp*, esta pose es actualizada en tiempo real, cada vez que la marca de AprilTag del efector se encuentra dentro del campo de visión de la cámara.
- p_i: es la pose $[tx, ty, tz, rx, ry, rz]$ del efector final dada por el feedback que proporcionan los encoders de los joints. Es la pose interna del efector y cumple que $p_i \neq p_v$, debido, insistimos, a los errores de calibración y a las holguras.
- p_t: es la pose $[tx, ty, tz, rx, ry, rz]$ del target.

[44] **AprilTags** es un sistema de realidad aumentada de etiquetas desarrollado por E. Olson en la Universidad de Michigan, EE.UU. Son marcas sintéticas de código binario definido por dos colores: blanco y negro

Con estos tres elementos, lo que persigue el componente VIK es reducir el error de traslación e_T y de rotación e_R[45] existente entre p_v y p_t.

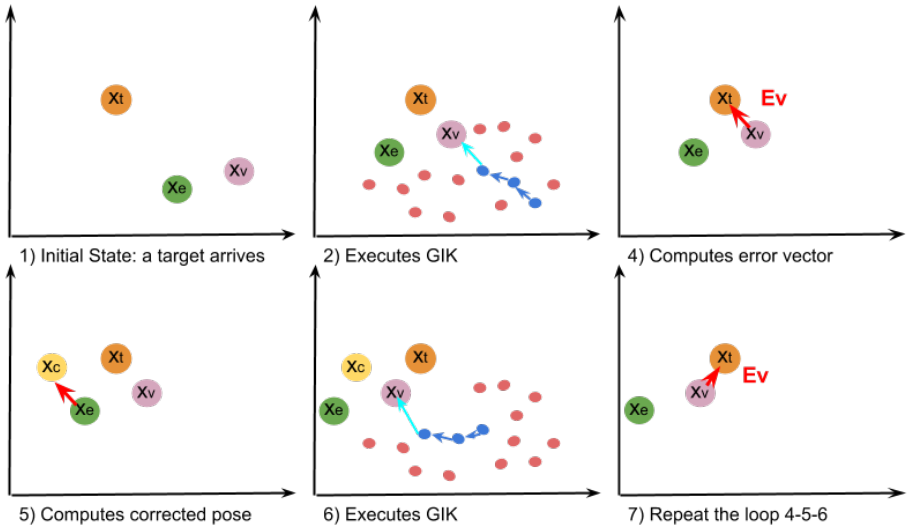


Figure 54: Esquema gráfico del funcionamiento de los tres módulos de cinemática inversa

El funcionamiento del algoritmo que implementa VIK es relativamente sencillo:

1. Cuando el componente VIK recibe un target p_t, lo envía al GIK para realizar una primera aproximación. El GIK mueve el efector final por el grafo de posiciones, desde la posición inicial p_i^0 hasta la posición p_i^n del grafo más cercana a p_t. En ese momento, el GIK llama al IK básico, que le devuelve la configuración de ángulos Θ para que el efector final alcance el target. El GIK ejecuta Θ a través del *commonjoint* y termina.

2. En tiempo real, el VIK mantiene actualizada la pose visual del efector final p_v. Cuando el GIK termina la primera aproximación, VIK calcula los errores e_T y e_R existentes entre p_t y p_v:

   - Si e_T y e_R son menores que los umbrales impuestos por el algoritmo (25mm y 0.18rad), la ejecución del VIK termina satisfactoriamente y el target finaliza con estado **RESOLVED**.

[45] e_T y e_R son las componentes de traslación y rotación del error E

- Si e_T y e_R superan los umbrales de error del algoritmo, el VIK calcula una nueva posición p_c que los corrija:

$$p_c = -^{p_v} p_t$$

$$^{p_c}R_{p_t} = {^{p_v}R_{p_t}}^{-1}$$

Donde $^{p_v}R_{p_t}$ es la matriz de rotación del target en el sistema de referencia de la pose visual del efector. Esta nueva pose es enviada al GIK, repitiéndose el proceso del punto 1.

Algorithm 4 VIK main algorithm

procedure VISUAL CALIBRATION
  $p_t \leftarrow targetArrives()$
  $sendTargetToGIK(p_t)$
  $p_v \leftarrow getAprilTagPose(robot)$
  while $(E = p_v - p_t) > threshold \wedge \neg timeOut()$ **do**
    $p_v \leftarrow getAprilTagPose(robot)$
    $p_c \leftarrow -^{p_v} p_t$
    $^{p_c}R_{p_t} \leftarrow {^{p_v}R_{p_t}}^{-1}$
    $sendTargetToGIK(p_c)$
  end while
end procedure

El componente VIK termina la ejecución de un target cuando el error alcanzado es aceptable, en cuyo caso el target finaliza con estado **RESOLVED**, o cuando el tiempo de ejecución consumido por el target supera el valor de un *timer*, en cuyo caso el target no ha podido ser resuelto con un error bajo y finaliza con estado **NOT_RESOLVED**.

De esta forma se implementa un sistema de cinemática inversa mucho más robusto, aplicable a cualquier tipo de robot y capaz de amortiguar los efectos negativos de la mala calibración, así como el juego de holguras presentes en los componentes de la estructura cinemática.

La estructura final de esta arquitectura software la podemos ver en el grafo **??**. En él están representados todos los componentes necesarios para levantar el sistema de

cinemática inversa. Esto es importante tenerlo en cuenta, en especial a la hora de codificar el fichero de configuración asociado a cada componente cinemático (IK, GIK y VIK).

También resulta llamativo contrastarlo con la antigua arquitectura software (ver grafo **??**), tan simple en comparación con la actual.

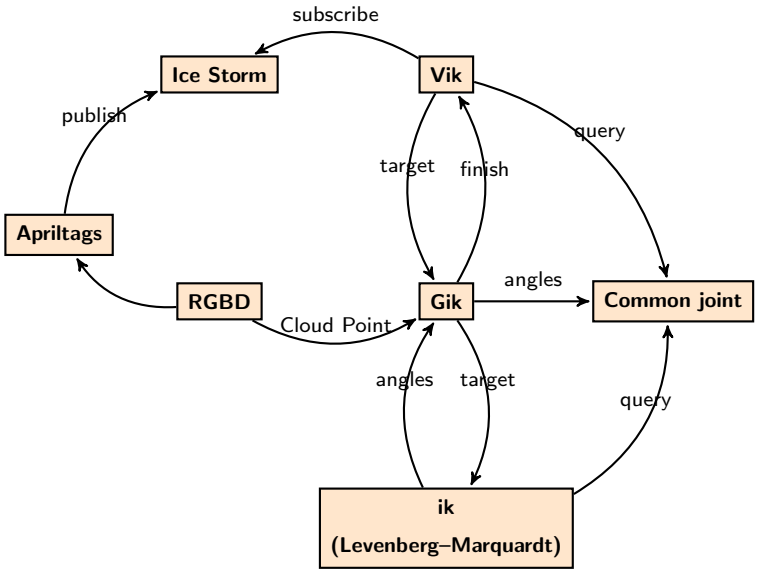


Figure 55: Visión esquemática del sistema de cinemática propuesto

Sin embargo esto es sólo una pequeña parte en todo el sistema de componentes que se ejecutan en el robot Shelly[34]. Podemos decir que esta arquitectura es sólo el nivel inferior de una arquitectura de agentes mucho mayor, en la cual destacan tres agentes:

1. El **Mission Agent**: a grandes rasgos, este agente se encarga de activar la misión del robot, por ejemplo *Coger la taza que está sobre la mesa*, y permite monitorizar el desarrollo de la misma.

2. El **Executive Agent**: este agente recibe la misión del *Mission Agent*, deduce cómo tiene que llegar al estado final deseado (en este caso, el estado final es que el robot coja la taza) y genera un plan, compuesto por una serie de reglas o acciones que son publicadas al resto de agentes, por ejemplo *cambiar de habitación*, *acercarse a la mesa*, *coger la taza*. También es el encargado de

coordinar las acciones del resto de agentes, para que sean coherentes con la misión y el estado del robot.

3. El **Grasping Agent**: este agente se encarga de coordinar a los componentes que forman el sistema de cinamática inversa visto. Es el nivel superior. Recibe las acciones a llevar a cabo del *Executive*, en este caso *coger la taza*. Para ello utiliza la información geométrica para calcular la distancia entre la posición del robot y la posición de la taza, hasta que estén lo suficientemente cerca como para lanzar la ejecución de la cinemática inversa, llamando al VIK.

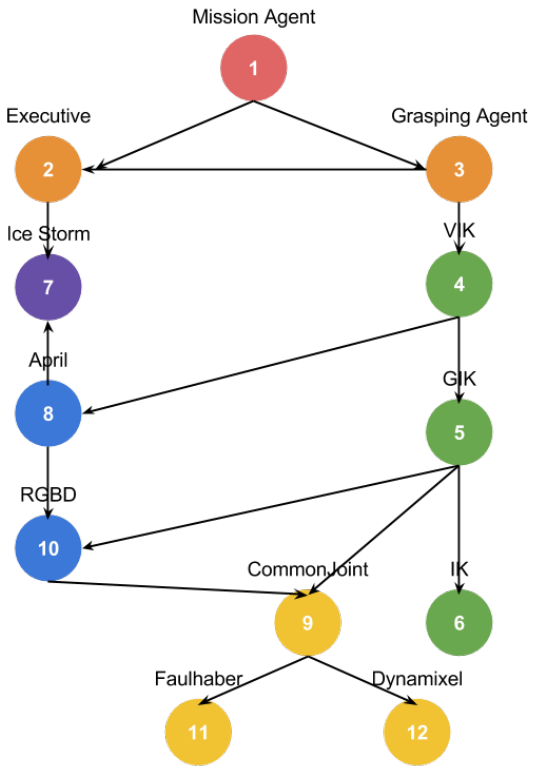


Figure 56: Arquitectura completa de la cinemática

6 Resultados y discusión

Para validar el nuevo sistema de cinemática inversa, así como para contrastar sus resultados con el antiguo componente de IK, se han realizado una serie de experimentos sobre el robot real.

Para llevar a cabo estos experimentos y obtener unos resultados concluyentes se ha tenido presente el **Teorema Central del Límite**. Este teorema indica que, si la población testeada P no sigue una distribución normal (como es el caso de las distintas posiciones target) pero el número de muestras tomadas es mayor que 30, se puede aproximar la distribución de medias de P a una normal, minimizándose el error a medida que se añaden muestras a la población.

Por esta razón, en cada experimento probado se han generado 50 escenarios (targets) posibles. De estos experimentos se han obtenido las medias de error de traslación y rotación, junto a otros parámetros interesantes para la validación, como por ejemplo intervalos de confianza.

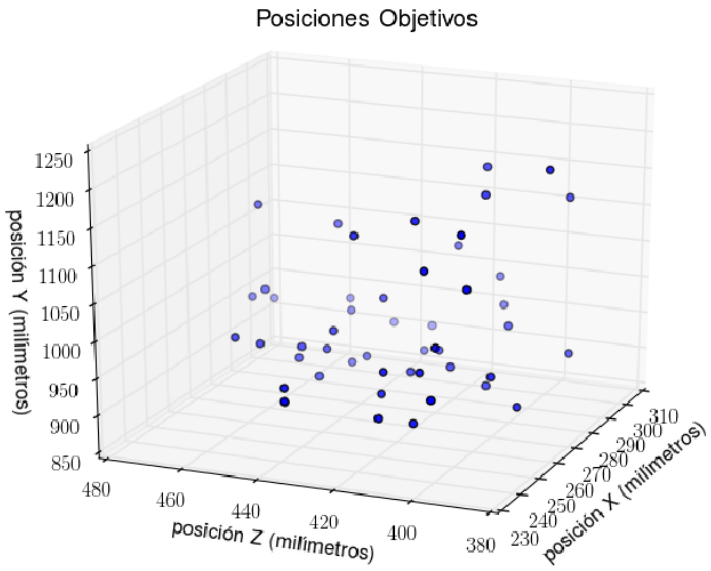


Figure 57: Poses experimentales utilizadas

Para probar el funcionamiento del VIK y del GIK se han planteado dos experimentos:

- Primero pondremos a prueba el funcionamiento del nuevo sistema cinemático con realimentación visual y sin ella. Así pues, comprobaremos el funcionamiento del GIK sin la detección de colisiones y sin la información del VIK. Este experimento nos mostrará el problema de las holguras y la mala calibración del brazo derecho de Shelly, dando como resultado la precisión del sistema cinemático en el robot real sin realimentación visual.

  Después introduciremos el VIK, y podremos comprobar como los resultados obtenidos por el sistema de cinemática mejoran de forma espectacular.

- El segundo experimento pondrá a prueba el sistema de detección de colisiones del GIK. Al ser ésta la última mejora añadida al sistema, no se ha podido probar sobre el robot real, debido a que está en mantenimiento. Por lo que mostraremos las pruebas realizadas sobre el simulador.

6.1 Realimentación visual, el gran salto

Para estos experimentos se generaron 50 targets cuyas componentes de traslación se encontraban entre $140 \leq X \leq 300$, $780 \leq Y \leq 800$ y $300 \leq Z \leq 390$, mientras que las componentes de rotación estaban fijas a 0 radianes en X e Y, y $\frac{\pi}{2}$ en Z.
Comencemos con el error de traslación: Si ejecutamos el experimento sólo con los componentes GIK e IK obtenemos una media de 196.7079mm de error en la traslación. Sin embargo, al incluir el nivel de VIK, ese error baja a 21.7998mm, lo que supone una reducción del 11.08%.

Datos	GIK	VIK
Media	196.7079 mm	21.7998 mm
Desviación típica	82.7402 mm	7.8722 mm
Máximo error	438.5210 mm	36.8398 mm
Mínimo error	58.3679 mm	5.3661 mm

Table 5: Error de traslación con GIK y con VIK

Podemos ver la gran diferencia que existe entre la ejecución del GIK y la del VIK en la siguiente gráfica 58. Si bien la gráfica une los errores con una línea, debemos aclarar que es más por cuestión de estética y visualización. En verdad los errores son independientes unos de otros por lo que sería más correctos señalarlos sólo con puntos aislados.

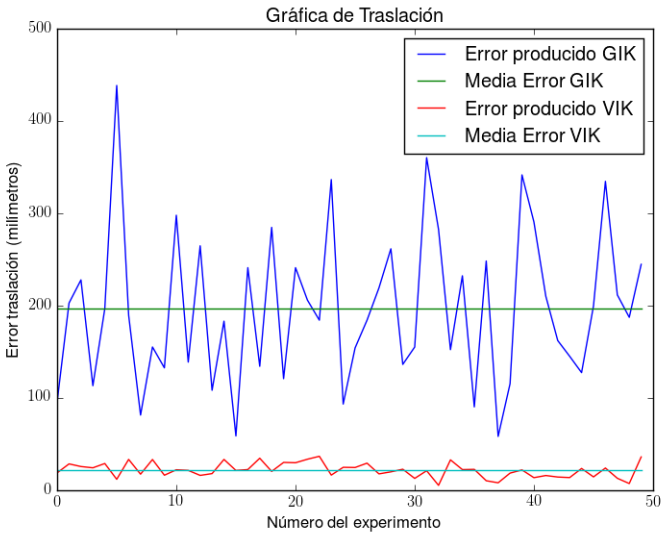


Figure 58: Comparativa del error de traslación entre GIK y VIK

Por otra parte los errores de rotación también son mayores al ejecutar el sistema cinemático sólo con los componentes GIK e IK, sin realimentación visual. Así nos encontramos con una media de 1.3204 radianes de error en el GIK, mientras que con el VIK la media no supera los 0.0738 radianes, una reducción del 5.589%.

Datos	GIK	VIK
Media	1.3204 rad	0.0738 rad
Desviación típica	1.0471 rad	0.047 rad
Máximo error	3.2000 rad	0.1593 rad
Mínimo error	0.0080 rad	0.0116 rad

Table 6: Error de rotación con GIK y con VIK

Al igual que en la gráfica superior, unimos los errores de rotación con una línea contínua por pura estética.

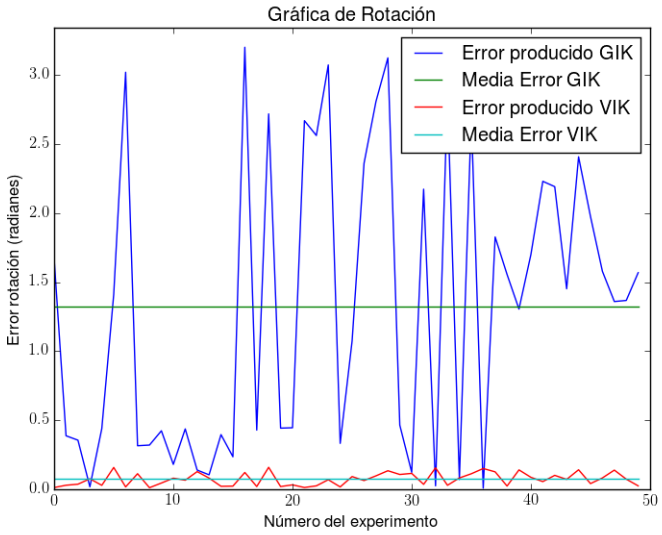


Figure 59: Comparativa del error de rotación entre GIK y VIK

Con estos datos podemos corroborar la gran disminución en cuanto a errores se refiere: de 300 mm en el IK antigua, hemos pasado a 22 mm con el VIK. Y de 1.95 radianes a 0.07 radianes.

Para finalizar, la inclusión del GIK (sin detección de colisiones) y del VIK en el sistema cinemático apenas ha supuesto un gran aumento del tiempo de ejecución, estando la media entorno a 5-7 segundos en un i3.

6.2 Prevención de colisiones

Por último, vamos a exponer brevemente el funcionamiento del método *delete_collision_points*, ya que no se dispone del robot real para probarlo, por causas de fuerza mayor, y tampoco hay suficiente tiempo como para desarrollar más a fondo esta nueva vía de mejora.

El experimento consistirá en una simulación simple sobre el nuevo brazo de Shelly. Colocaremos en el simulador una pequeña caja, dentro del rango de acción del efector final:

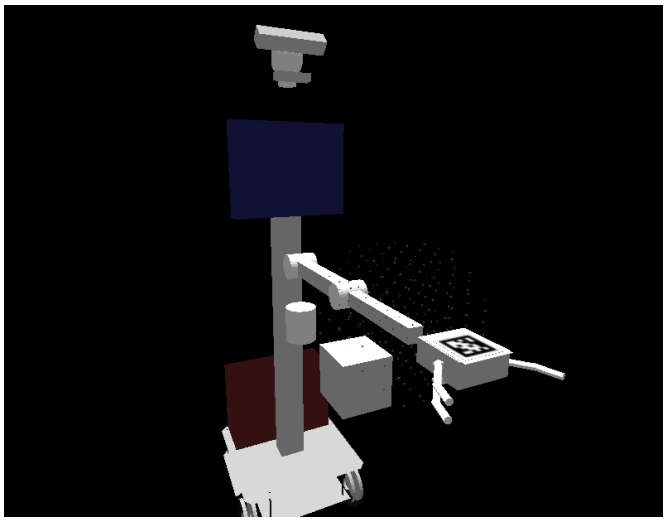


Figure 60: Colocación del obstáculo dentro del rango de acción del efector

Hemos probado a mandar el efector final a poses cercanas al obstáculo[46], obligando al efector final a atravesarlo en su camino. Como resultado hemos obtenido un comportamiento bastante favorable, en el que el GIK anula los nodos que colisionan con el cubo, obligando al efector a dar una especie de salto:

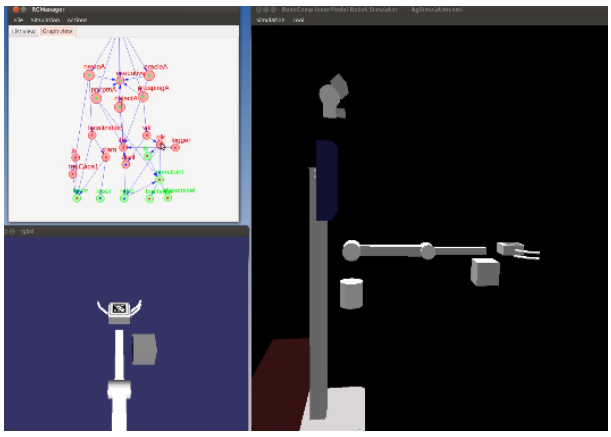
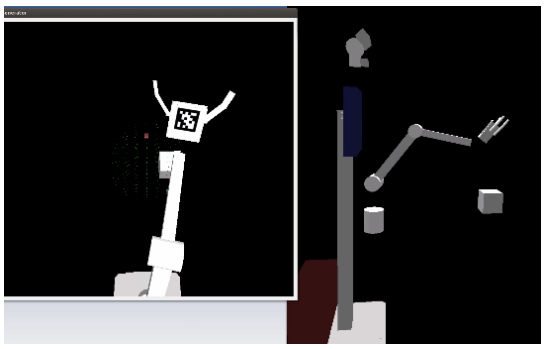


Figure 61: Posición de partida del brazo robótico.

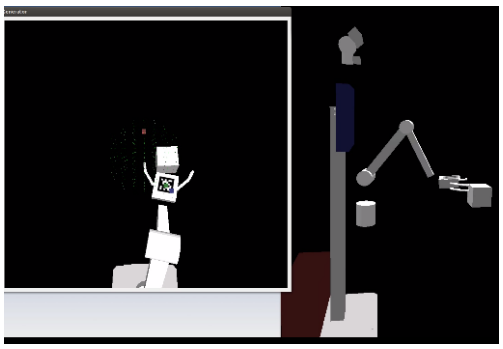
En el primer experimento, partimos de la posición (0, 900, 900) en el espacio. La RGBD ve perfectamente el brazo del robot y el obstáculo colocado a su derecha (un cubo). Con el GIK colocamos el target en el punto (100, 900, 500, 0, 0, 0).

[46]En el video anexo se puede observar como el brazo va alas posiciones (100, 900, 900) y (100, 900, 500). También se ha probado con 50 puntos comprendidos en el rango de trabajo del efector final.

En las siguientes imágenes podemos ver cómo Shelly levanta el brazo para pasar por encima del cubo, esquivándolo. Finalmente llega al target.



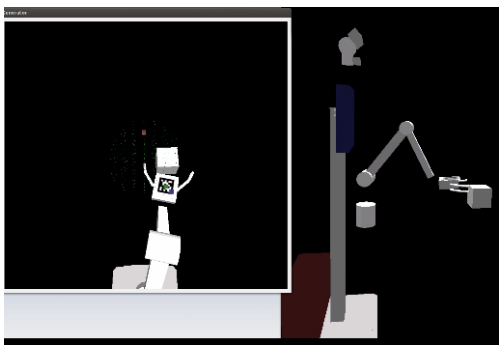
(a) El robot esquiva el objeto levantando el brazo



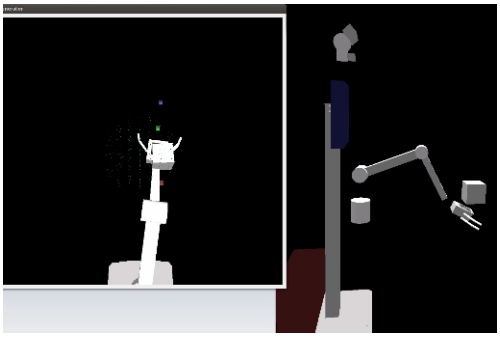
(b) El efector alcanza el target

Al ejecutar el experimento con el VIK, los datos de error no han sufrido grandes variaciones, siendo el único punto negativo la carga computacional del método, que tarda en ejecutarse de media 4.5 segundos en un i3.

También se han observado los comportamientos anómalos mencionados en la sección 5.6: cuando el brazo tapa parte del objeto, el GIK calcula un camino que lo atraviesa justo por donde la RGBD no puede verlo:



(c) Partimos de la posición target anterior



(d) El efector atraviesa ligeramente la parte baja del cubo

Figure 62: En este caso podemos observar cómo el brazo está tapando parte del cubo (la parte inferior del cubo, en concreto). Por esta razón, al mandar el efector final al nuevo target, el GIK atraviesa ligeramente la zona oculta del cubo.

7 Conclusiones

Este trabajo se planteó como una revisión y mejora del antiguo software de cinemática que se propuso en el TFG anterior [8]. Como resultado, ha derivado en un proyecto de gran envergadura en el que se ha tocado dos de los mayores problemas de la cinemática: por una parte las inconsistencias provocadas por las holguras y la mala calibración, y por otra, la detección de obstáculos y la prevención de colisiones. Partiendo desde el algoritmo original, se ha estudiado y se ha probado cada situación problemática para la cinemática, dando como resultado un sistema software más completo y robusto.

Por supuesto hay que seguir refinando el sistema, para optimizar su funcionamiento lo máximo posible. Se ha planteado actualmente compilar todo el software con una precisión de *double* en vez de la actual precisión de *float*, para lo cual se ha modificado completamente la librería matemática de RoboComp, **QMat**, convirtiéndola en un template que admita tanto *float* como *double*. Sin embargo esta modificación aún no ha podido ser probada, debido a los grandes cambios que implica realizar en casi todos los componentes de RoboComp, quedando su completa inclusión en el sistema en pausa, por ahora.

También hay que retocar el método *delete_collision_points* del GIK, para hacerlo más rápido computacionalmente hablando. Esto supone un punto muy importante, ya que el robot debe trabajar en tiempo real y para ello los tiempos de computación deben ser más bajos de lo que son ahora.

Otro punto de mejora es incluir una especie de memoria dentro del GIK, que permita recordar al componente la localización de los obstáculos que la cámara detecta. De esta forma evitaríamos los comportamientos erróneos que actualmente se están dando.

También se propone revisar la autodetección del brazo, mejorando el método que actualmente ofrece la clase *InnerModel*. Llegados a este punto, hay que señalar que es necesario de forma urgente actualizar las clases que emplean la librería **FCL**, la cual ha sufrido una actualización. Actualmente, si se quiere compilar RoboComp con

soporte para FCL, es necesario descargar una versión antigua de la librería.

Por otra parte, con la introducción del nuevo brazo de Shelly, más preciso y sin holguras, habrá que testear de nuevo todo el sistema, para adaptarlo a la nueva cadena cinemática. En principio, el software que hemos desarrollado es genérico y puede emplearse en cualquier robot. Tan sólo se necesita modificar la cadena cinemática con la que trabajan los componentes IK, GIK y VIK, modificando los ficheros de configuración, y crear el nuevo modelo interno o representación interna del robot, preparando un nuevo fichero de innermodel.

References

[1] Antonio Barrientos, Luis Felipe Peñín, Carlos Balaguer, and Rafael Aracil. *Fundamentos de Robótica*. McGraw-Hill/Interamericada de España, S.A., 1997 edition, 1997.

[2] Encyclopædia Britannica Company Merriam-Webster. Full definition of robot.

[3] Luis Basañez Villaluenga Emmanuel Nuño Ortega. Teleoperación: técnicas, aplicaciones, entorno sensorial y teleoperación inteligente. Technical report, Universidad Politécnica de Cataluña, Instituto de Organización y Control de Sistemas Industriales, 2014.

[4] J.G.C. Devol. Programmed article transfer, June 13 1961. US Patent 2,988,237.

[5] J M Haut, M E Paoletti, P Bustos, and N García. Code2bot, a social robot for the classroom. *CAEPIA, Albacete*, 2015.

[6] Adrián Romero-Garcés, Luis Vicente Calderita, Jesús Martínez-Gómez, Juan Pedro Bandera, Rebeca Marfil, Luis J Manso, Antonio Bandera, and Pablo Bustos. Testing a fully autonomous robotic salesman in real scenarios. In *IEEE International Conference on Autonomous Robots Systems and Competitions*, pages 1–7, 2015.

[7] F Fernández, M Martínez, I García-Varea, J Martínez-Gómez, J M Pérez-Lorenzo, R Viciana, P Bustos, L J Manso, L Calderita, M Gutiérrez, P Núñez, A Bandera, A Romero-Garcés, J P Bandera, and R Marfil. Gualzru's path to the advertisement world. *IROS Fine-R Workshop*, 2015.

[8] Mercedes Paoletti Ávila. Cinemática inversa en robots sociales. 2014.

[9] Luis Manso, Pilar Bachiller, Pablo Bustos, and Luis Calderita. Robocomp: a tool-based robotics framework. *Lecture Notes in Computer Science. Simulation, Modeling and Programming in Autonomous Robots*, 6472:251–262, 2010.

[10] P Gavin. The levenberg-marquardt method for nonlinear least squares curve-fitting problems. 2015.

[11] Ananth Ranganathan. The levenberg-marquardt algorithm. 2004.

[12] Pedro Yuste Gallego. Algorithms for efficient computation of generalized inverse kinematics in humanoid robots. 2014.

[13] L. E. Scales. *Introduction to Non-Linear Optimization*. 1985.

[14] A. Argyros M. I. Lourakis. Sba: A software package for generic sparse bundle adjustment. *ACM Transactions on Mathematical Software*, 36(1):1–30, 2009.

[15] P Bustos, L J Manso, L V Calderita, S Leal, and C Parra. Ursus: a robotic assistant for training of children with motor impairments. In D. Torricelli J.L Pons and Marta Pajaro, editors, *Converging Clinical and Engineering Research on Neurorehabilitation, Springer series on BioSystems and BioRobotics*, chapter Ursus: A R, pages 249–254. Springer, 2012.

[16] I. García-Varea, A Jiménez-Picazo, J. Martínez, A Revuelta, L. Rodríguez, P Bustos, and Pedro Núñez. Apedros: Asistencia a personas con discapacidad mediante robots sociales. In *IBERDISCAP*, 2011.

[17] L.V. Calderita, P. Bustos, C. Suárez Mejías, F. Fernández, R. Viciana, and a. Bandera. Asistente robótico socialmente interactivo para terapias de rehabilitación motriz con pacientes de pediatría. *Revista Iberoamericana de Automática e Informática Industrial RIAI*, 12(1):99–110, 2015.

[18] Mario Haut, Luis Manso, Daniel Gallego, Mercedes Paoletti, Pablo Bustos, Antonio Bandera, and Adrián Romero-Garcés. A navigation agent for mobile manipulators. *WAF, ROBOTS*, 2015.

[19] P Bustos, L J Manso, M A Gutiérrez, M Haut, M Paoletti, I García-Varea, J Martínez-Gómez, L Rodríguez-Ruíz, A Romero-Garcés, and R Marfil. Ursus team. *Rockin 2015, Lisboa*, 2015.

[20] T. Sugihara. Solvability-unconcerned inverse kinematics by the levenberg–marquardt method. *Robotics, IEEE Transactions on*, 27(5):984 – 991, 2011.

[21] Jianmin Zhao and Norman I Badler. Real time inverse kinematics with joint limits and spatial constraints. *ScholarlyCommons*, 1989.

[22] Paolo Baerlocher and Ronan Boulic. An inverse kinematics architecture enforcing an arbitrary number of strict priority levels. *The Visual Computer*, 20(6):402–417, 2008.

[23] Antonio Barrientos, Luis Felipe Peñín, Carlos Balaguer, and Rafael Aracil. *Fundamentos de Robótica*. McGraw-Hill/Interamericana de de España, S.A.U, 2 edition, 2007.

[24] Jesús Ángel de Pedro Sangrós, Roberto Jinménez Pacheco, and Jorge Santolaria Mazo. *Técnica de alineación de sistemas de referencia de sensores de medida a bordo para calibración de robots industriales. Aplicación a palpadores autocentrantes de cinemática paralela*. PhD thesis, 2012.

[25] R Bernard and S. L. Albright. *Robot Calibration*. Chapman & Hall, 1 edition, 1993.

[26] Ángel O. Rodríguez Vázques and Eduardo Castillo Castañeda. *Localización del elemento efector del robot paralelo de tres grados de libertad PARALLIX*. PhD thesis, 2010.

[27] Roberto Conde Ojeda. Aplicación del algoritmo rrt a la planificación de caminos para robots móviles en configuración ackerman.

[28] D López, F Gómez-Bravo, F Cuesta, and A Ollero. Planificación de trayectorias con el algoritmo rrt. aplicación a robots no holónomos. *Revista Iberoamericana de Automática en Informática Industrial (RIAI)*, 3(3):56–67, 2006.

[29] M. A. Gutiérrez, A. Romero-Garcés, P. Bustos, and J. Martínez. Progress in robocomp. In *Workshop of Physical Agents*, 2012.

[30] Carlos Soria, Flavio Roberti, Ricardo Carelli, and José M. Sebastián. Control servo visual de un robot manipulador tipo scara basado en pasividad. *Revista Iberoamericana de Automática e Informática Industrial (RIAI)*, 5(4):54–61, 2008.

[31] Seth Hutchinson, Gregory D. Hager, and Peter I Corke. A tutorial on visual servo control. *IEEE Transactions on robotics and automation*, 12(5):651–670, 1996.

[32] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. 2003.

[33] E. Olson. Apriltag: a robust and flexible visual fiducial system. *Robotics and Automation (ICRA), IEEE International Conference on*, pages 3400–3407, 2011.

[34] Luis Vicente Calderita. *Deep State Representation: an Unified Internal Representation for the Robotics Cognitive Architecture CORTEX*. PhD thesis, Escuela Politécnica de Cáceres, Universidad de Extremadura, 2016.