UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Cinemática Inversa en Robots Sociales

Mercedes Paoletti Ávila

Julio 2014


Escuela Politécnica

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Cinemática Inversa en Robots Sociales

Autor/es: Mercedes Paoletti Ávila
Fdo.:

Director/es: Pablo Bustos García de Castro
Luis Calderita Estévez
Fdo.:

Tribunal Calificador

Presidente: Pilar Bachiller Burgos
Fdo.:

Secretario: Antonio Plaza Miguel
Fdo.:

Vocal: Carmen Ortiz Caraballo
Fdo.:

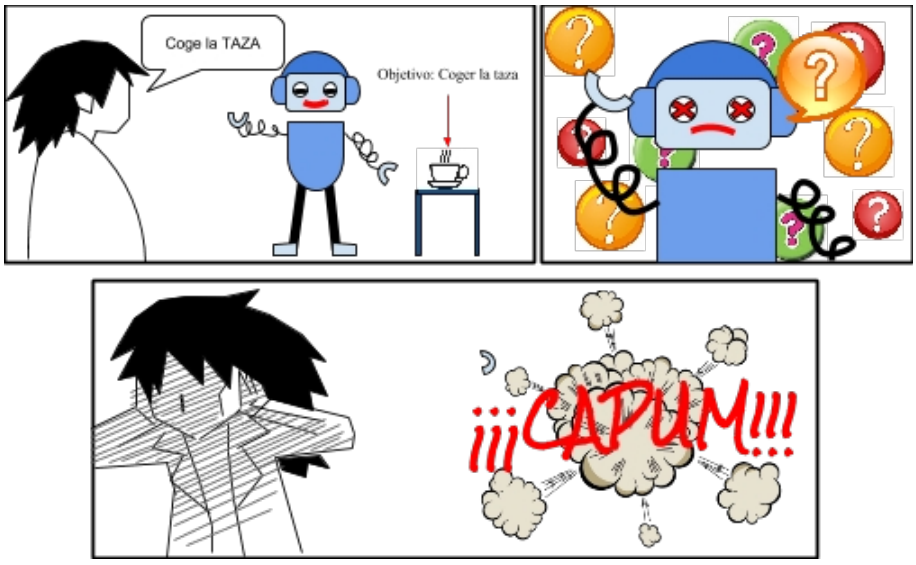
CALIFICACIÓN:
FECHA:

Índice

1. Introducción

Desde los inicios de la robótica, los investigadores se han planteado de manera reiterada dos objetivos, a saber: 1) dotar al robot de cierta **inteligencia** o capacidad de raciocinio; y 2) darle **autonomía**, en cuanto al movimiento se refiere. De esta forma podríamos afirmar que en el campo de la robótica existen dos grandes y persistentes problemáticas basadas, una, en la *inteligencia artificial* y, otra, en la *cinemática del robot*. Con este trabajo hemos pretendido aclarar la complejidad del problema que rodea a la cinemática de los robots, más concretamente en el campo de la cinemática inversa (IK) aplicada al entorno de los robots sociales. Para ello, nos centraremos en un ejemplo práctico, como es la trayectoria de un brazo robótico desde una situación inicial hasta un punto objetivo.

El problema es conocido: ¿cómo conseguir que un robot mueva su brazo para alcanzar una taza, un lápiz o cualquier objeto que se le marque como objetivo, en un tiempo y con una carga de cómputo razonable? O de manera más simplificada, si se prefiere: dado una pose (tx, ty, tz, rx, ry, rz) objetivo en el espacio tridimensional ¿cómo llevamos la mano del robot a esa pose?



Para solucionar este típico problema de cinemática inversa se ha estudiado el algoritmo de búsqueda local de **Levenberg-Marquardt** aplicado sobre el robot social Ursus 3.0,

desarrollado por el laboratorio de robótica de la Escuela Politécnica de Cáceres, **Robolab**, en la Universidad de Extremadura.

Por otra parte, y con el fin de programar dos componentes, *inverseKinematicsComp* e *ineverseKinematicTesterComp*, para que planteen y resuelvan problemas de cinemática inversa presentados al robot Ursus, nos apoyaremos en el framework de código abierto para robótica **Robocomp** (desarrollado también por el equipo de Robolab), el cual proporciona una serie de librerías, componentes y herramientas[1] software que permiten desarrollar programas para robots de forma fácil, eficaz y distribuida. El lenguaje que se utilizará para desarrollar el proyecto será C++, realizando algunas pruebas de verificación sobre MATLAB.

En el siguiente apartado introduciremos al lector en algunas cuestiones previas que consideramos oportunas, antes de abordar nuestro objetivo principal, como son los conceptos de robots sociales y el problema de la cinemática inversa aplicado a ellos, para después introducir la metodología empleada en la solución del problema anteriormente enunciado y los resultados que se han obtenido a lo largo de toda la investigación.

[1] En los siguientes apartados iremos explicando detenidamente los distintos elementos de Robocomp que necesitaremos a lo largo del desarrollo de este proyecto, a medida que vayan surgiendo.

2. Robots Sociales y de Servicios

2.1 Definición de *robot*

Para definir un robot quizás debamos preguntarnos primero ¿qué es un robot? Hay múltiples y variadas formas de definir y explicar en qué consisten este tipo de máquinas. Por ejemplo, según el **Instituto de Robots de América** (Robot Institute of America, RIA), 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.*"

Pero 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*."

Con tantos matices y acepciones, resulta complicado definir exactamente qué es un robot, cuáles son sus principales características y en que se basa. Lo que parece claro es que los robots son dispositivos pensados y diseñados para realizar ciertas tareas y/u ofrecer ciertos servicios.

Nosotros definiremos los robots como máquinas controladas por un computador, diseñadas principalmente para sustituir y/o ayudar al humano en tareas repetitivas (por ejemplo las actividades típicas de una cadena de montaje como soldar, ensamblar, pintar...), reemplazarlo en acciones peligrosas (como supervisar el interior del reactor nuclear de Chernobyl, investigar los restos del Titanic...), o simplemente inviables para una persona (como es la exploración del planeta Marte), trabajando siempre bajo supervisión humana, ya sea de forma autónoma o teleoperados.

2.2 Historia y evolución de los robots

Antes de que aparecieran los primeros robots industriales, ciertas tareas peligrosas, como era el manejo de sustancias radiactivas en una central nuclear, eran llevadas a cabo por los **teleoperadores** o telemanipuladores. Estas máquinas, del tipo maestro-esclavo, consistían en extensiones mecánicas que un operario humano controlaba, reproduciendo con cierto grado de exactitud los movimientos del trabajador.

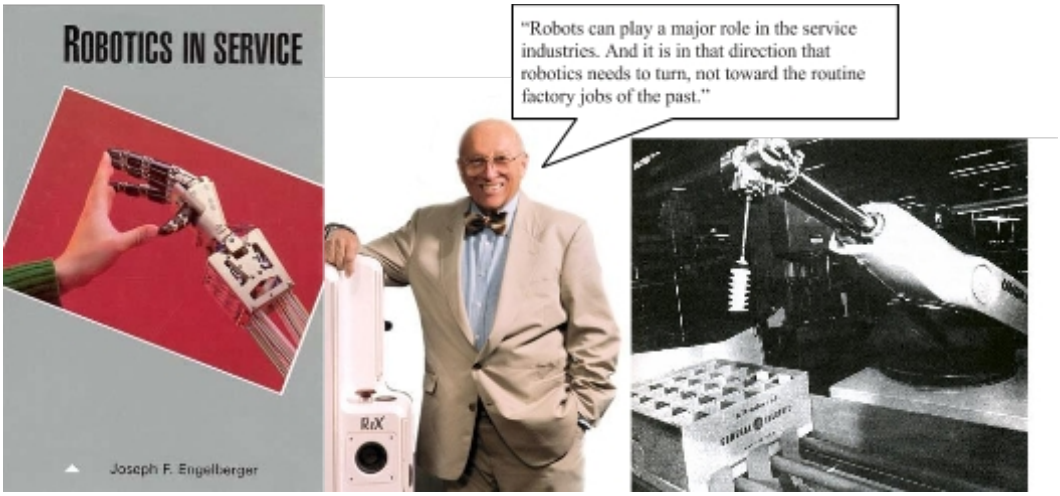
Los primeros robots aparecieron en la década de los 50 (aunque la idea de autómatas y seres humanoides artificiales se remonta a la antigua Grecia[2]), con el desarrollo del primer manipulador programable, diseñado por el estadounidense George Devol como una máquina flexible, adaptable y fácil de manejar. Este hito marcó el inicio de la **robótica industrial**. A partir de entonces se intenta sustituir al operador humano, que controla los movimientos y acciones de los teleoperadores, por un programa informático, obteniéndose los primeros **robots industriales** propiamente dichos. Éstos se caracterizan por ser manipuladores re-programables y multifuncionales, disponen de varias articulaciones o grados de libertad (lo que les permite moverse en un espacio o rango de trabajo) y pinzas o efectores, diseñados para manejar herramientas, piezas y materiales, desarrollando tareas propiamente industriales, como puede ser el ensamblado de las piezas de un coche.

En los años 60, los robots comienzan a utilizar sensores, lo que les permite tener cierto conocimiento de su entorno. Aparecen los **robots perceptores**. De esta forma, el controlador que ejecuta las instrucciones del programa informático, utiliza la información aportada por los distintos sensores del robot para monitorizar el correcto funcionamiento de éste. Es a finales de esta década, cuando los laboratorios empiezan a interesarse por la inteligencia artificial, que alcanzará su auge en los años 80.

Es justamente en la década de los 80 cuando se avanza en técnicas de reconocimiento de voz y de objetos, y se desarrollan los primeros robots con fines militares, de seguridad y de rehabilitación. Este desarrollo de los robots en actividades fuera de los campos de la

[2] En la mitología griega se dice que Hefesto, el dios de la forja, fabricó una serie de autómatas artificiales, las llamadas doncellas doradas o Kourai Khryseai, dotadas de inteligencia y cierta autonomía, que le ayudaban en su fragua.

industria y la investigación dio como resultado lo que hoy en día conocemos como **robots de servicios**. Este concepto, tan de moda en nuestros días, no es nuevo. Fue acuñado por vez primera en 1989, cuando el ingeniero físico estadounidense Joseph Engelberger (considerado el padre de la robótica y desarrollador, junto a George Devol, del primer robot industrial para una cadena de montaje, *Unimate*) publicó su libro "*Robotics in Service*".



Los robots de servicio operan de forma "semi" o completamente autónoma, realizando múltiples tareas útiles a los humanos y a otros equipos (robots, computadores...), como de limpieza, de inspección, de construcción y/o demolición, acciones de rescate...

Dentro del grupo de los robots de servicio se encuentran los **robots sociales**. Son robots autónomos o semi-autónomos[3], que interactúan y se comunican con los seres humanos u otros agentes físicos autónomos (como otros robots), siguiendo una serie de comportamientos o patrones sociales y unas normas vinculadas a su función. Podemos destacar la existencia de una gran variedad de robots sociales:

- Aquellos dedicados al ocio y el entretenimiento.
- Los que se dedican al campo de la educación.
- Los que desarrollan sus actividades en el campo de la vigilancia y la seguridad.

[3] Para algunos autores, una característica esencial de un robot social es la completa autonomía, que el robot sea capaz de tomar un gran número de decisiones por sí mismo. Otros autores permiten un grado menor de autonomía para considerar al robot como tipo social.

- Aquellos que ofrecen servicios sanitarios, como terapia y rehabilitación, o simplemente como compañía.
- Otros que ofrecen algún tipo de información o sirven como organizadores (de agenda, contactos, tareas...).

En cualquier caso y a pesar de la gran variedad de robots sociales, todos ellos tienen en común el objetivo fundamental de satisfacer el **bienestar** del ser humano, utilizándose como una herramienta al servicio de la persona. Así pues, este trabajo se centrará en aquellos servicios en los que se requiera que el robot interactúe y se comunique con las personas de su entorno, manipulando objetos cotidianos de una casa.

3. El problema de la cinemática inversa

Hemos visto que los robots industriales y los de servicios tienen que *moverse* hacia poses de destino y *manipular* cierto tipo de objetos. A este tipo de robots se les llama **manipuladores** y en ellos se da un interesante problema: ¿cómo saben a dónde moverse para poder manipular los objetos? Existen dos formas de resolver este problema:

- **Pose enseñada**: normalmente, los robots industriales se mueven hacia poses objetivo que les han sido enseñadas con anterioridad. Primero se entrena al robot para que lleve su efector hacia ciertos puntos de destino de tal forma que, cuando los alcanzan, se leen los sensores de posicionamiento de cada motor y se almacenan sus valores angulares. Así, cuando el robot deba ir a una pose leerá los ángulos correspondientes que debe asignar a cada motor.
- **Pose calculada**: en estos casos, al robot se le indica una posición y una orientación de destino en el espacio, sin existir un entrenamiento previo. Es el robot el que se encarga de calcular los valores angulares que deben adoptar sus motores para alcanzar la pose objetivo. Esta forma de resolver el problema permite mover el efector del robot a todas poses que se encuentren dentro de su rago de trabajo, algo que no puede hacer el método anterior, donde es obligatorio enseñar primero al robot cómo llegar a la pose.

En ambos casos se está haciendo uso de la **cinemática del robot**, aunque aplicada de dos formas distintas.

Cuando hablamos de la cinemática de un robot hacemos referencia al estudio de su movimiento sobre un sistema de referencia.

Este campo será justamente nuestro objeto de investigación y análisis a lo largo del presente proyecto. Pero para poder profundizar en el estudio de la cinemática, los métodos, usos y las aplicaciones que tiene (así como el desarrollo del componente diseñado expresamente para resolver este tipo de problemas) conviene primero introducir al lector en el mundo de los robots manipuladores, de la geometría 3D y de los sistemas de referencia.

3.1 Robots manipuladores

Como explicamos en el apartado anterior, los primeros robots manipuladores que aparecieron alrededor de los años 70 se diseñaron para trabajar en entornos industriales, por ejemplo para manejar elementos radioactivos en una central nuclear, y a partir de ahí se han ido desarrollando hasta ocupar otros sectores, como el de servicios.

Un robot manipulador se caracteriza por ser un tipo de robot flexible (en el sentido de adaptable) y re-programable, dotado de una o varias articulaciones, que le proporcionan suficientes grados de libertad para realizar de forma automática o semi-automática ciertos procesos que conllevan la manipulación de algún objeto u objetos externos al robot, de una forma precisa. Existen varios criterios para clasificar los distintos tipos de robots manipuladores. Uno de ellos es el seguido por la *Asociación Francesa de Robótica Industrial* (AFRI), en la que los robots manipuladores se clasifican en cuatro categorías:

1. Tipo A o **telemanipuladores**: este tipo de robots no tienen capacidad de decisión. Suele componerse de una estructura esclava (toda la parte manipuladora, como el brazo y el efector final) y una parte maestra controlada por un operario humano.
2. Tipo B o **pre-reglados**: son robots automáticos cuyos movimientos están regulados mediante topes.
3. Tipo C o **programables**: son robots que no tienen conciencia de su entorno y cuyas trayectorias, continuas o punto a punto, se pueden reprogramar. Cuentan con la estructura mecánica, unos sensores digitales y/o analógicos y los elementos de programación. Se utilizan en tareas repetitivas.
4. Tipo D o **inteligentes**: es un robot más avanzado, capaz de aprender de su entorno y readaptar su comportamiento. Cuentan con elementos de programación bastante más avanzados, como componentes software para la percepción del entorno o la toma de decisiones, además de la estructura mecánica y de los sensores digitales y/o analógicos.

Existen más formas de clasificar los robots manipuladores[4], pero a pesar de los múltiples nombres y características de cada una de ellas, todas coinciden en que los robots se componen de segmentos unidos a motores que le proporcionan la capacidad de movimiento. La cadena formada por esos motores y segmentos se denomina **cadena cinemática**:

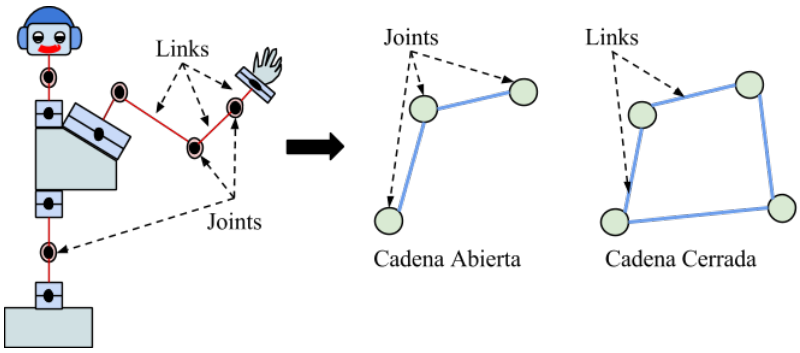


Illustration 1: Cadenas cinemáticas en robots (abiertas y cerradas)

Formalmente, la cadena cinemática se compone de eslabones (links o segmentos) unidos por conectores móviles (joints o articulaciones). Las cadenas cinemáticas se clasifican a su vez en:

- **Cadenas abiertas**, si existen unos extremos inicial (una base) y final (un efector). En este tipo de cadenas todos los joints son activos, cada uno tiene capacidad para moverse. Los robots seriales utilizan estas cadenas en sus estructuras.
- **Cadenas cerradas**, si no existen extremos. En este tipo de 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.

[4] La **Federación Internacional de Robótica** (IFR) los clasifica en *robots secuenciales*, *robots de trayectoria controlable*, *robots adaptativos* y *robots tele-manipulados*.

Según **T. M. Kansel** los robots se clasifican en cinco generaciones: *pick and place* (sin grados de movilidad, es usado en manipulación y servicio de maquinas), *servo* (controlado por servo-control, con trayectoria continua y programación condicional, se usa en soldadura y pintura.), *ensamblado* (controlado con servos de precisión, visión y tacto, guiado por vía, se usa en ensamblado), *móvil* (controlado por sensores inteligentes, usa patas o ruedas en su movimiento y se usa en construcción y movimiento), y *especiales* (controlados por técnica de inteligencia artificial y se usan con fines militares y espaciales).

3.2 Geometría 3D y sistemas de referencia

Uno de los problemas que tienen los robots manipuladores es conocer dónde están dispuestos sus elementos estructurales en el espacio en el que se mueven. Por ello necesitamos un sistema de referencia que nos sitúe o posicione los elementos del robot en el espacio de trabajo.

Un **sistema** o **marco de referencia** de un sistema físico es un conjunto de acuerdos o convenciones usados por un ente observador para poder medir posiciones, rotaciones y demás magnitudes físicas del sistema estudiado. Explicado de una forma más simple, 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.

Existen varios tipos de coordenadas (cartesianas, polares, cilíndricas, esféricas, geográficas, curvilíneas generales, curvilíneas ortogonales...), a lo largo de este proyecto utilizaremos marcos de referencia basados en ejes y coordenadas **cartesianos**. Este sistema de coordenadas está definido por uno, dos o tres ejes ortogonales o 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.

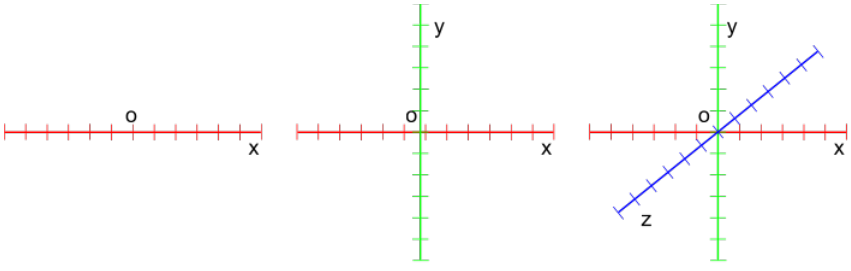


Illustration 2: Ejes cartesiabos en 1D, 2D y 3D

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 del que

"cuelga") 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 3 y queremos calcular las coordenadas de la muñeca W(wx, wy, wz) en el sistema del hombro, habría que calcular las matrices de transformación para pasar de la muñeca al codo y del codo al hombro: $W_S = M_{ES} \cdot M_{WE} \cdot W_w$.

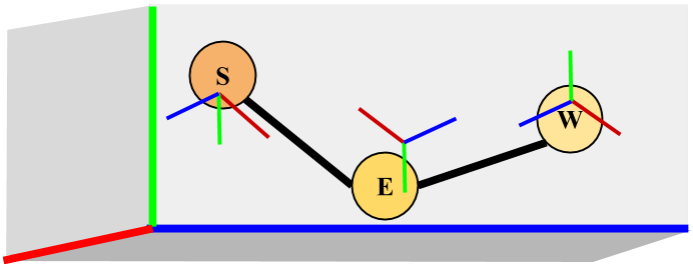


Illustration 3: Ejemplo de brazo robótico en 3D

Estas matrices de transformación están compuestas por la matriz de rotación 3x3 y por el vector de traslación de 3 elementos, de tal forma que para trasladar las coordenadas de la muñeca al sistema de referencia del hombro nos quedaría la siguiente operación:

$$\begin{pmatrix} W_x \\ W_y \\ W_z \\ 1 \end{pmatrix}_S = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix}_{ES} \cdot \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix}_{WE} \cdot \begin{pmatrix} W_x \\ W_y \\ W_z \\ 1 \end{pmatrix}_W = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}_{ES} \cdot \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}_{WE} \cdot \begin{pmatrix} W_x \\ W_y \\ W_z \\ 1 \end{pmatrix}_W$$

Equation 1: Transformación de un punto de un sistema de referencia a otro mediante el producto de matrices de transformación

Por otro lado, este proyecto también hará uso de la la geometría 3D, que se encarga de estudiar las figuras o cuerpos en el espacio tridimensional, como puntos, rectas, planos, curvas... En particular haremos uso de la **geometría analítica**, que estudiará los cuerpos geométricos del plano o del espacio desde un punto de vista matemático. Dicho de otra forma, la geometría analítica define los cuerpos geométricos con ecuaciones o fórmulas del tipo F(x)=y, siendo F una función continua o no en todo el plano (2D) o el espacio (3D).

3.3 Cinemática Directa

En robótica, cuando se trabaja con robots móviles y/o manipuladores, existen dos problemas cinemáticos fundamentales a tener en cuenta:

- El problema de la **cinemática directa.**
- El problema de la **cinemática inversa**.

Supongamos que tenemos 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 4. Esta cadena se compone de:

- Dos joints: el **hombro**, que forma un ángulo α con el eje X, y el **codo**, que forma un ángulo β con el segmento que une el hombro con el codo.
- Dos links: el **brazo**, que une el hombro con el codo y tiene una longitud fija L1; y el **antebrazo**, que une el codo con el efector final del brazo, la mano, y tiene una longitud fija L2.
- El extremo de la cadena o efector final rígido, la **mano**.

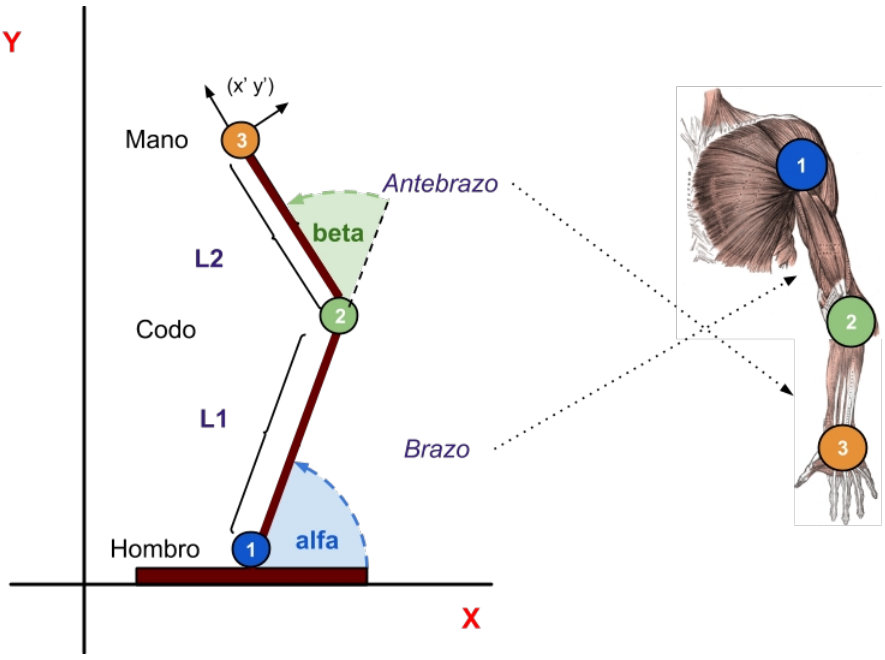


Illustration 4: Esquema de brazo robótico en un sistema 2D

El problema de la **cinemática directa** consiste en determinar la posición y la orientación de la mano una vez conocidos los ángulos que forman las articulaciones del hombro y del codo, así como las longitudes del brazo y del antebrazo. Lo que busca la cinemática directa es una función F que devuelva las coordenadas de la mano dados los ángulos de los joints y las longitudes de los links:

$$\boldsymbol{p}_{mano}(\boldsymbol{p}_x, \boldsymbol{p}_y) = \boldsymbol{F}(\theta, \boldsymbol{L}) \quad \theta = [\alpha, \beta] \;\; y \;\; L = [L_1, L_2]$$

Equation 2: Función de cinemática directa

En el ejemplo de la figura 4, la función de cinemática directa se formula como sigue:

- $p_x = L_1 \cdot \cos(\alpha) + L_2 \cdot \cos(\alpha + \beta)$
- $p_y = L_1 \cdot \sin(\alpha) + L_2 \cdot \sin(\alpha + \beta)$

En este proyecto, toda la parte de cinemática directa está resuelta gracias a la herramienta de Robocomp, ***InnerModel***, cuya función es la de ayudar al programador a resolver las tareas algebraicas más comunes en los problemas de robótica, como por ejemplo, los cálculos para trasladarse de un sistema de referencia a otro o los cálculos de matrices de transformación entre las uniones de las articulaciones del robot en la cadena cinemática. Cabe destacar que *InnerModel* se sustenta en dos pilares: por una parte tiene un lenguaje basado en XML que utiliza etiquetas propias para definir el mundo del robot, por ejemplo, para definir una articulación se usa la etiqueta <***joint*** *"nombre de la articulación" tx="" ty="" tz="" rx="" ry="" rz=""*> donde se define el nombre de la articulación, sus traslaciones y sus rotaciones; y por otra parte cuenta con una clase que lo mantiene actualizado y con la que se realizan las consultas.

3.4 Cinemática Inversa

El problema en las cadenas cinemáticas abiertas se complica (y bastante) cuando hablamos de **cinemática inversa** (IK). La cinemática inversa permite trasladar y rotar un nodo o efector de la cadena cinemática, de un punto inicial A a un punto objetivo B, una vez que se conocen el punto objetivo, el número de articulaciones (que no sus ángulos, pues éstos son las incógnitas a resolver) y la longitud de los segmentos que unen esas articulaciones. La cinemática inversa se encarga de resolver la secuencia de ángulos de

los joints para llevar el efector final desde el punto inicial A hasta el punto final u objetivo B, como se muestra en la siguiente figura:

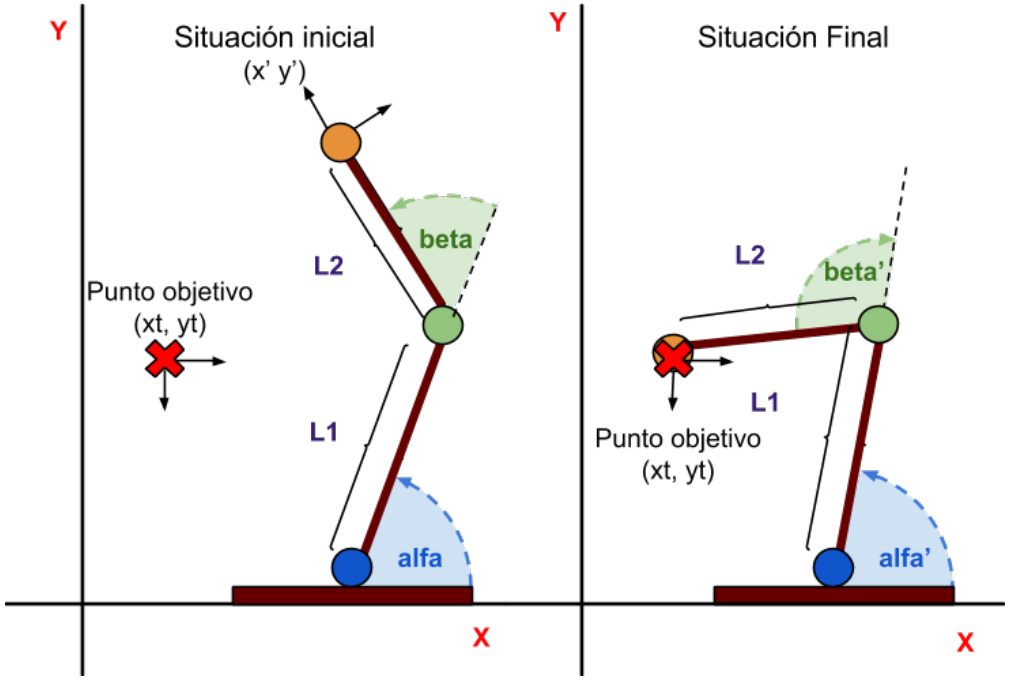


Illustration 5: Ejemplo de cinemática inversa

En el ejemplo anterior, al introducir un punto objetivo referenciado en el sistema de la base y la restricción de que la mano debe estar sobre ese punto, creamos un bucle o polígono "mal" cerrado. El bucle estará abierto mientras la mano no alcance su objetivo y el error o "apertura" será la distancia entre ambos.

Esto es lo que sucede en la situación inicial del sistema que vemos en la figura 5, el polígono no se cierra: la mano del robot está colocada en una posición y el punto objetivo (comúnmente llamado *target*) está colocado en otra, formando un bucle que no cierra (en la figura 6 podemos ver el bucle que se forma entre el *target,* el cero del sistema y la mano en rojo, como también el vector de error de cierre entre la mano y el punto objetivo, marcado en rosa).

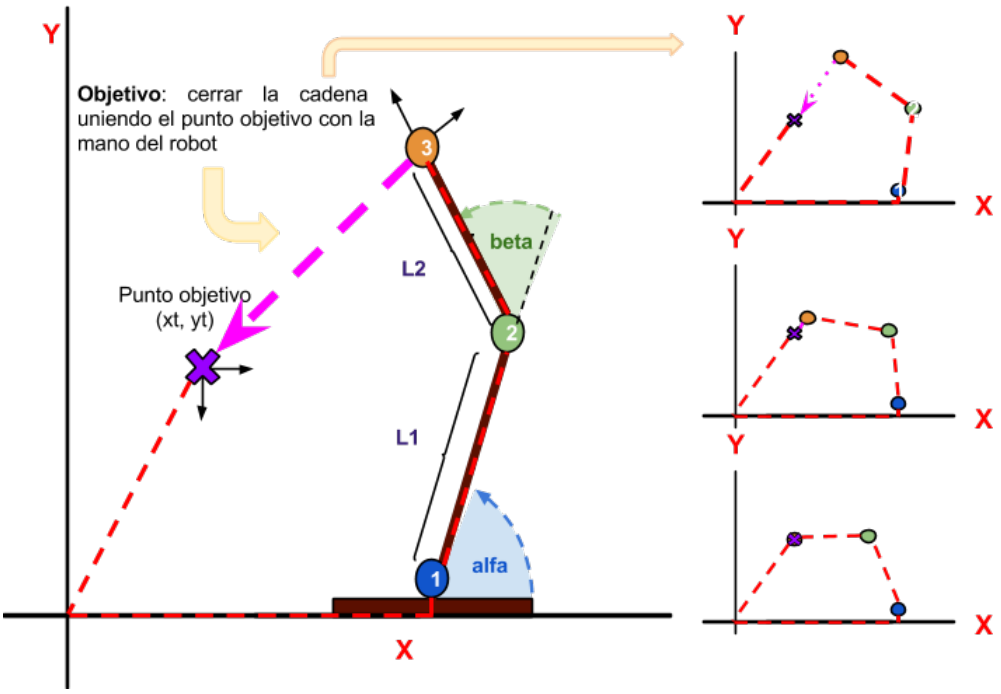


Illustration 6: Cadena cinemática cerrada utilizada por la IK

En la herramienta *InnerModel* los bucles no se pueden representar de forma explícita porque la estructura de datos subyacente es un **árbol cinemático**[5], como veremos en apartados posteriores. Sin embargo, podemos imaginar que se crean bucles cuando se perciben elementos desde dos nodos del robot, como por ejemplo una cámara que mira un objeto y una mano que intenta alcanzarlo o que lo está sosteniendo. En ese momento aparecería un bucle que va desde la mano del robot hasta la cámara, pasando por todos los joints intermedios, como pueden ser la muñeca, el codo o el cuello, y que vuelve a la mano, cerrándose. Otro ejemplo sería cuando el robot junta las manos para sostener algún objeto. Estos bucles normalmente no son geométricamente perfectos, es decir, no cierran con exactitud. Ese error de cierre se interpreta como lo que debe corregir el robot en su percepción tanto del mundo como de sí mismo para poder recalibrarse. Esta recalibración implica un movimiento que en la cinemática inversa tiene como objetivo anular ese error de cierre

[5] En un árbol los bucles no tienen cabida. Los nodos (a excepción del primer nodo del que "arranca" el árbol) siempre cuelgan de un único nodo padre. La existencia de un bucle en un árbol implicaría que un nodo tiene dos padres.

La cinemática inversa, se encarga de encontrar los valores de los ángulos que forman los joints para que el extremo del brazo se sitúe sobre el punto objetivo y se oriente correctamente. Lo que busca es la función inversa a la de cinemática directa, una F^{-1} que devuelva los ángulos de los joints, dadas las coordenadas del punto objetivo $P_{target}(t_x, t_y)$ y las longitudes de los segmentos del brazo:

$$\theta = F^{-1}(P_{target}, L) \quad \theta = [\alpha', \beta'], \; P_{target}(x_t, y_t,) \; y \; L = [L_1, L_2]$$

Equation 3: Función de cinemática inversa

Pero, volviendo al ejemplo de la figura 4, se nos plantea un problema: ¿cómo calculamos la inversa de las funciones de cinemática directa que hemos visto antes, $p_x = L_1 \cdot \cos(\alpha) + L_2 \cdot \cos(\alpha+\beta)$ y $p_y = L_1 \cdot \sin(\alpha) + L_2 \cdot \sin(\alpha+\beta)$? No podemos hacerlo de una forma simple. Existen una serie de problemas que dificultan su solución:

- La solución no es sistemática, puesto que las ecuaciones no son lineales.
- La solución depende de la configuración del robot. Además pueden darse infinitas soluciones, dados unos datos iniciales.
- No siempre existe una solución cerrada.

Por todo ello, nos vemos obligados a recurrir a métodos matemáticos genéricos, que intentan acercarse a una solución óptima de forma iterativa en un tiempo razonable.

En este documento se estudiará el **método iterativo de Levenberg-Marquardt** para la resolución general del problema de la cinemática inversa aplicada a los brazos del robot social Ursus.

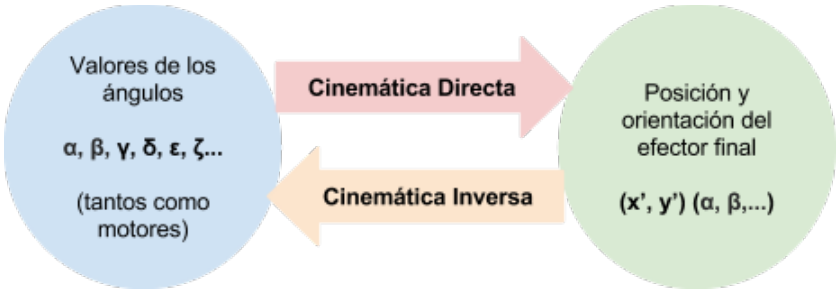


Illustration 7: Relación entre la cinemática directa y la inversa

4. Solución general de la IK con Levenberg-Marquardt

4.1 Introducción a los métodos de IK

Como hemos visto, no es trivial encontrar la solución al problema de cinemática inversa. Existen múltiples métodos que estudian la forma de llegar a una solución:

- Métodos geométricos: se caracterizan por usar relaciones trigonométricas y geométricas para calcular una posible solución a la IK. Estos métodos suelen ser suficientes para robots con pocos grados de libertad.
- Métodos a partir de matrices de transformación homogéneas: trabajan con matrices de transformación entre los diferentes sistemas de referencia con los que trabaja el robot.
- Métodos de desacoplamiento cinemático, utilizado sólo en cierto tipo de robots con seis grados de libertad. 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.

Para resolver el problema de cinemática inversa se ha optado por un método iterativo conocido como el **algoritmo de Levenberg-Marquardt** o el algoritmo de **mínimos cuadrados amortiguados**. Este método se utiliza para resolver problemas no lineales de mínimos cuadrados en los que se busca una solución que disminuya una función de error.

En este apartado se explicará en qué se fundamenta y cómo trabaja[6] sobre un sistema 2D teniendo en cuenta sólo las traslaciones, para simplificar su explicación. Pero antes es necesario introducir una serie de conceptos matemáticos que se van a utilizar.

[6] Explicaremos el algoritmo tomando como principal referencia el documento "*SBA: A Software Package for Generic Sparse Bundle Adjustment*", escrito por los profesores de la Foundation for Research and Technology de Hellas (Grecia), Manolis I. A. Lourakis y Antonis A. Argyros.

4.2 Teorema de Taylor

Este teorema permite aproximar una función derivable **G(x)** en el entorno de un punto **a**, (a-ε, a+ε), mediante un polinomio **P**, cuyos coeficientes dependen de las derivadas de la función en ese punto.

Lo que quiere decir es que si existe un punto **x** cercano al punto **a** (**x** pertenece al entorno del punto **a**, (a-ε, a+ε)) donde la función **G(x)** es medible y derivable **n** veces, entonces existe un polinomio **P** de orden **n** de la siguiente forma:

$$P_n(x)=G(a)+G'(a)(x-a)+\frac{G''(a)}{2!}(x-a)^2+\frac{G'''(a)}{3!}(x-a)^3+\ldots+\frac{G^n(a)}{n!}(x-a)^n$$

Equation 4: Polinomio de Taylor de grado n para función de una sola variable

Este polinomio cumple con $G(x)\simeq P_n(x)$ si $x\in(a-\varepsilon,a+\varepsilon)$. Es decir, la solución del polinomio en el punto **x** dentro del entorno del punto **a** se aproxima a la solución de la función **G** en ese punto **x**. Entonces se puede decir que el valor de la función en el punto **x**, es igual que el polinomio de Taylor de orden **n** de **x** más un resto o error **Rn+1(x)**:

$$G(x)=P_n(x)+R_{n+1}(x) \quad .$$

Equation 5: Aproximación a la función G(x) mediante el polinomio de Taylor

Un caso interesante (y que se va a usar más adelante) es el polinomio de primer orden, que responde a la siguiente forma:

$$P_1(x)=G(a)+G'(a)(x-a),\forall\, x\in(a-\varepsilon,a+\varepsilon)$$

Equation 6: Polinomio de Taylor de grado 1

Con esta última fórmula, se demuestra que el polinomio de Taylor de orden 1 de una función **G(x)**, es la recta tangente a dicha función en el punto **x=a**, con fórmula $y=n+m\cdot x$, donde $y=P_1(x)$, $n=G(a)$ y $m\cdot x=G'(a)\cdot(x-a)$.

Esto es para funiones con una única variable. Pero el teorema de Taylor se puede generalizar para funciones multivariantes, de tal forma que el polinomio **P** de orden 1 de una función con dos variables **H(x,y)** en un punto **b**(bx, by), perteneciente al entorno del punto **a**(ax, ay) (es decir, que $b_x\in(a_x-\varepsilon,a_x+\varepsilon)$ y $b_y\in(a_y-\varepsilon,a_y+\varepsilon)$), sería:

$$P_1(b_x, b_y) = H(a_x, a_y) + \left(\frac{\partial H}{\partial b_x}\right)_b (b_x - a_x) + \left(\frac{\partial H}{\partial b_y}\right)_b (b_y - a_y)$$

Equation 7: Polinomio de grado1 para funciones multivariantes

Donde $\frac{\partial H}{\partial b_x}$ es la derivada parcial de la función **H** con respecto a la coordenada en X del punto b, y $\frac{\partial H}{\partial b_y}$ es la derivada parcial de la función **H** con respecto a la coordenada en Y del punto b, by. Este polinomio define el plano tangente a la función **H(x,y)** en el punto **b(bx, by).**

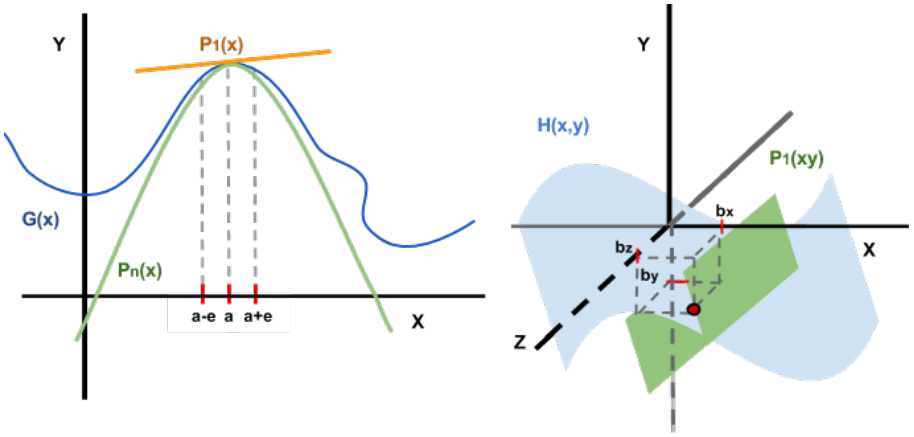


Illustration 8: Ejemplos gráficos de polinomios de Taylor

En la figura 8 se muestran varios ejemplos gráficos de polinomios de Taylor. Para la función de una sola variable G(x), en azul sobre un sistema 2D, se han representado el polinomio de grado n, Pn(x), en verde, y el polinomio de primer grado, P1(x), en naranja, en el entorno de un punto **a**. También se ha representado en el sistema 3D una función de dos variables, H(x,y), en azul, a la que se le ha pintado el polinomio de primer grado, P1(x,y), como el plano tangente al punto **b**(bx,by,bz), pintado en rojo.

4.3 Métodos numéricos en los que se basa Levenberg-Marquardt

El algoritmo de Levenberg-Marquardt combina el método de **Gauss-Newton** con el de **gradiente**. Para entender correctamente el funcionamiento del Levenberg-Marquardt hace falta explicar estos dos métodos y los principios en los que se basan.

4.3.1 El método del gradiente

Este método se incluye entre los algoritmos generales de descenso. Es conocido como método del gradiente, método del descenso más pronunciado o método de Cauchy, y es uno de los más usados para minimizar una función diferenciable multivariante.

El método del gradiente comienza en un punto inicial x^0 de una función multivariante $F(x)=F(x_1, x_2 .., x_n)$ y va iterando, determinando en cada iteración una **dirección de movimiento** sobre la función *F(x)*. Esa dirección de movimiento viene determinada por el **gradiente** de la función, ∇F , que es el campo vectorial asociado al campo escalar *F* y se calcula mediante las derivadas parciales de éste:

$$\nabla F(x)=\left(\frac{\partial F(x)}{\partial x_1}, \frac{\partial F(x)}{\partial x_2}, \ldots \frac{\partial F(x)}{\partial x_n}\right) \quad .$$

Equation 8: Gradiente de una función

El gradiente de la función en un punto $\nabla F(x)$ indica la dirección en la que el campo escalar *F* varía de una forma rápida, esto es, apunta en la dirección en la que la función *F* varía de una forma más pronunciada[7]. Además, el gradiente se hace cero en los puntos máximos, mínimos y de inflexión de la función *F*.

En la figura 9 podemos ver el gradiente expresado de forma geométrica como un vector tangente a la función *F* en el punto x:

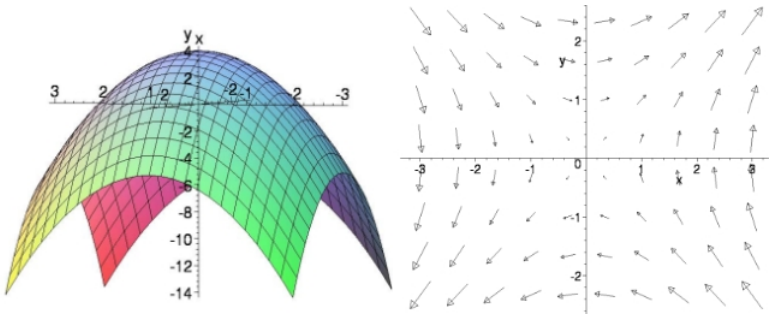


Illustration 9: Gradiente de descenso de una superficie curva

[7] El gradiente de un punto tiene la dirección y el sentido en los que la derivada direccional o la tasa de cambio de la función en ese punto es máxima.

El método de descenso rápido utiliza el gradiente negativo para buscar la **dirección de descenso** de la función *F*. Dicho de otra forma, en cada iteración el método escoge aquella dirección en la que la función *F* decrece más rápidamente.

La dirección de descenso de la función *F* en el punto **x** será el vector **v** si existe un **δ > 0** tal que $F(x+\lambda \cdot v) < F(x)$ para todo λ perteneciente al intervalo positivo (0, **δ**) y coincide justamente con la dirección contraria a $\nabla F(x)$, $-\nabla F(x)$. El método del gradiente se mueve a lo largo de esta dirección hasta localizar un punto con gradiente nulo, es decir, este método va descendiendo por la función *F* hasta encontrar un mínimo.

4.3.2 El método de Gauss-Newton

Se utiliza para resolver problemas de **mínimos cuadrados no lineales**. En este tipo de problemas, el objetivo es encontrar el mínimo de una función no lineal que es la suma de los cuadrados de otras funciones. Por ejemplo, si se tienen **m** funciones F1, F2..., Fm cada una con **n** parámetros o variables $x=[x_1, x_2 ..., x_n]$, el problema de los mínimos cuadrados no lineales será encontrar el mínimo de la función **S**(x) definida como:

$$S(x)=\sum_{i=1}^{m}(F_i(x))^2=\|F(x)\|^2 \quad con\ F=[F_1, F_{2...}, F_m] \ \ y \ \ x=[x_1, x_{2...}, x_n]$$

Equation 9: Función a minimizar en Gauss-Newton

El método de Gauss-Newton se basa en una simplificación del **método de optimización de Newton** que sólo utiliza las primeras derivadas de la función S(x) para encontrar sus raíces (puntos máximos y mínimos) y su idea principal es aproximar linealmente la función S(x) mediante un **desarrollo de Taylor** de primer orden. Así, partiendo de la función S(x) en un punto x=**a**, se considera que para pequeñas perturbaciones Δa (ya sean incrementos o decrementos, dependiendo del signo de Δa) se obtiene una serie de Taylor tal que $S(a\pm\Delta a)=S(a)+\epsilon$, lo que quiere decir que la función en el punto ligeramente perturbado $a\pm\Delta a$ tiene el mismo valor que la función en el punto exacto **a** más un error, como se describe en el apartado "*Teorema de Taylor*".

Por otra parte, para minimizar la función S(x), lo que hace el método de Gauss-Newton es buscar su dirección de descenso hasta encontrar un punto donde el gradiente se haga

cero, lo que indica que se ha llegado a un mínimo. El vector gradiente de S(x), llamado g(x), se calcula mediante la expresión $g(x)=\nabla(S(x))=J(x)^t\cdot F(x)$, siendo F(x) el vector de funciones $[F_1, F_2 \ldots, F_m]$ y $J(x)^t$ la matriz jacobiana transpuesta de ese vector F(x), cuyas filas son las derivadas parciales de cada función $F_i(x)$ entre cada uno de los parámetros del vector $x=[x_1, x_2 \ldots, x_n]$, que no son otra cosa que los gradientes de cada función, $F_i(x)$:

$$J(x)=\begin{pmatrix} \frac{\partial F_1}{x_1} & \frac{\partial F_1}{x_2} & \ldots & \frac{\partial F_1}{x_n} \\ \ldots & \ldots & \ldots & \ldots \\ \frac{\partial F_m}{x_1} & \frac{\partial F_m}{x_2} & \ldots & \frac{\partial F_m}{x_n} \end{pmatrix}=[\nabla F_1, \nabla F_2 \ldots, \nabla F_m]^t$$

Equation 10: El jacobiano de F son los gradientes de cada Fi

Si queremos buscar la dirección de descenso entonces el cálculo del gradiente será $g(x)=-\nabla(S(x))=-J(x)^t\cdot F(x)$. Además, al no utilizar las segundas derivadas, el método de Gauss-Newton aproxima el hessiano, la matriz de segundas derivadas parciales de S(x), mediante el cálculo $H(x)=J(x)^t\cdot J(x)$ [8].

Lo primero que hace el método de Gauss-Newton es expresar la función a minimizar S(x) en forma matricial, de tal forma que S(x)=y es lo mismo que tener una matriz de coeficientes que multiplica un vector de parámetros o variables:

$$S(x)=y \rightarrow \begin{pmatrix} S_{11} & \ldots & S_{1n} \\ \ldots & \ldots & \ldots \\ S_{m1} & \ldots & S_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \ldots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \ldots \\ y_n \end{pmatrix}$$

El objetivo es bajar por el gradiente de la función S(x) hasta que sea cero, $g(x)=-\nabla S(x)=0$. Hemos visto que esta expresión del gradiente de descenso es idéntica a $-J(x)^t\cdot F(x)=0$. Si expresamos esta ecuación de forma matricial obtendremos:

[8] En verdad, el método de Gauss- Newton sólo calcula la primera parte del hessiano, siendo la aproximación completa calculada de la siguiente forma: $J^t\cdot J+\sum_{i=1}^{m} F_i(x)\cdot\nabla^2 F_i(x)$

$$-J(x)^t \cdot F(x) = \begin{pmatrix} \frac{\partial F_1}{x_1} & \frac{\partial F_1}{x_2} & \cdots & \frac{\partial F_1}{x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial F_m}{x_1} & \frac{\partial F_m}{x_2} & \cdots & \frac{\partial F_m}{x_n} \end{pmatrix}^t \cdot \begin{pmatrix} F_{11} & F_{12} & \cdots & F_{1n} \\ \cdots & \cdots & \cdots & \cdots \\ F_{m1} & F_{m2} & \cdots & F_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \cdots \\ x_n \end{pmatrix} = 0$$

Ya tenemos una parte de la ecuación, nos falta "descubrir" la otra. Tenemos una pista: el hessiano nos devuelve las segundas derivadas parciales y cuando las segundas derivadas parciales se hacen cero es porque existe un mínimo o un máximo. Así que ya tenemos la segunda parte de la ecuación: igualamos el hessiano a cero, $H(x)=0$, que es lo mismo que $J(x)\cdot J(x)^t=0$. Esto indica que tanto el gradiente como el hessiano deben ser cero, por lo que la ecuación final sería $-J(x)^t \cdot F(x) = J(x)\cdot J(x)^t = 0$, que matricialmente es:

$$\begin{pmatrix} \frac{\partial F_1}{x_1} & \cdots & \frac{\partial F_1}{x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial F_m}{x_1} & \cdots & \frac{\partial F_m}{x_n} \end{pmatrix}^t \cdot \begin{pmatrix} F_{11} & \cdots & F_{1n} \\ \cdots & \cdots & \cdots \\ F_{m1} & \cdots & F_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \cdots \\ x_n \end{pmatrix} = \begin{pmatrix} \frac{\partial F_1}{x_1} & \cdots & \frac{\partial F_1}{x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial F_m}{x_1} & \cdots & \frac{\partial F_m}{x_n} \end{pmatrix}^t \cdot \begin{pmatrix} \frac{\partial F_1}{x_1} & \cdots & \frac{\partial F_1}{x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial F_m}{x_1} & \cdots & \frac{\partial F_m}{x_n} \end{pmatrix}$$

Simplificando, esta expresión quedaría $-J(x)^t \cdot F(x) \cdot \boldsymbol{x} = J(x)\cdot J(x)^t$, donde las J(x) son las matrices jacobianas cuando el vector de parámetros es x, F(x) ahora es la matriz de coeficientes del vector de funciones F(x)=[F1, F2..., Fm] y x es el vector de parámetros $x=[x_1, x_2 \ldots, x_n]$.

Teniendo ya la ecuación completa, sólo debemos estimar un vector de parámetros inicial x^0 para poder utilizar el método de Gauss-Newton en la búsqueda de una configuración para el vector de parámetros $x=[x_1, x_2 \ldots, x_n]$ que minimice la función S(x). En las iteraciones sucesivas, el vector de parámetros x se re-calculará siguiendo la siguiente expresión:

$$x^{k+1} = x^k - \left(J(x^k)^t \cdot J(x^k)\right)^{-1} \cdot J(x^k)^t \cdot F(x^k) \quad .$$

Equation 11: Actualización del vector de parámetros x

Lo que significa que el vector de parámetros para la siguiente iteración del método k+1, x^{k+1} , es el resultado de restar al vector de parámetros de la iteración anterior k, x^k toda la expresión $(J(x^k)^t \cdot J(x^k))^{-1} \cdot J(x^k)^t \cdot F(x^k)$, donde $J(x^k)$ es el jacobiano de F(x) cuando el vector de parámetros vale x^k . Dividamos esta expresión en dos partes:

1. $(J(x^k)^t \cdot J(x^k))^{-1}$ está calculando la inversa de la aproximación al hessiano. ¿Y por qué multiplica el jacobiano por su transpuesto? No se puede asegurar que el jacobiano siempre sea una matriz cuadrada y una matriz que no es cuadrada no se puede invertir. Multiplicando el jacobiano por su transpuesto nos aseguramos que la matriz es cuadrada e invertible si tiene rango completo.
2. $J(x^k)^t \cdot F(x^k)$ es el gradiente de descenso de la función S(x).

A pesar de asegurar que la matriz $J^t \cdot J$ es cuadrada, el cálculo de su inversa[9] puede dar problemas si ésta es singular (no tiene rango completo, por lo que su determinante siempre es cero). Por otra parte hemos dicho que una matriz que no es cuadrada no se puede invertir, pero eso no es del todo cierto. Existen matrices rectangulares que sí tienen inversa. Por ejemplo para una matriz A_{mn} con m>n (matriz alta) puede existir una matriz L_{nm} tal que $L_{nm} \cdot A_{mn} = I$, entonces se dice que L es la matriz inversa de A por la izquierda. Del mismo modo, para una matriz A_{mn} con m<n (matriz ancha) puede existir una matriz R_{nm} tal que $A_{mn} \cdot R_{nm} = I_{nn}$, entonces se dice que R es la matriz inversa de A por la derecha. Esas inversas se llaman pseudoinversas o inversas generalizadas.

Así pues, generalmente se utiliza otra expresión en la que se sustituye el término $(J(x^k)^t \cdot J(x^k))^{-1} \cdot J(x^k)^t \cdot F(x^k)$ por un incremento δ^k quedando la ecuación como $x^{k+1} = x^k + \delta^k$. El objetivo es resolver ese δ^k , así que igualamos las dos expresiones:

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

$$\delta^k = x^k - (J(x^k)^t \cdot J(x^k))^{-1} \cdot J(x^k)^t \cdot F(x^k) - x^k$$

[9] Para calcular la inversa se divide la matriz adjunta transpuesta por el determinante de la matriz: $A^{-1} = \frac{1}{|A|} \cdot (A^{adj})^t$. Si la matriz no tiene rango completo, el determinante se hace cero. No se puede dividir algo entre cero.

$$\delta^k = -1 \cdot \left(J(x^k)^t \cdot J(x^k)\right)^{-1} \cdot J(x^k)^t \cdot F(x^k)$$

$$\left(\left(J(x^k)^t \cdot J(x^k)\right)^{-1}\right)^{-1} \cdot \delta^k = -1 \cdot J(x^k)^t \cdot F(x^k)$$

Finalmente obtenemos las ecuaciones normales:

$$J(x^k)^t \cdot J(x^k) \cdot \delta^k = -J(x^k)^t \cdot F(x^k) \quad .$$

Equation 12: Ecuaciones normales

Esta ecuación permite despejar δ^k para obtener el incremento óptimo que se debe sumar a la siguiente iteración. Este incremento es el que conduciría directamente, de un salto, al mínimo de la función si ésta fuese cuadrática. En el caso general de funciones no cuadráticas, las ecuaciones normales se resuelven para cada iteración sobre la función F, proporcionando la solución óptima suponiendo que localmente es cuadrática.

4.4 Método de Levenberg-Marquardt

Como hemos comentado, el algoritmo de Levenberg-Marquardt combina el método de **Gauss-Newton** con el **de gradiente**. Cuando la solución actual está lejos de un mínimo local, el algoritmo Levenberg-Marquardt se comporta como un método de descenso rápido. 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. Presenta el problema de que depende demasiado de las condiciones iniciales de ejecución, pudiendo quedar atrapado en mínimos locales.

4.4.1 ¿Cómo aplicamos el algoritmo de Levenberg-Marquardt?

Para explicar de forma sencilla cómo funciona el algoritmo de Levenberg-Marquardt tomaremos como ejemplo el sistema de la figura 4 del apartado 3.3. El algoritmo de Levenberg-Marquardt necesita una situación de partida para poder arrancar, por eso digamos que la mano o efector final de la figura 4 está situado inicialmente en el punto 2D $P_{mano}(P_x, P_y)$, medido por la función de cinemática directa F(x) (donde x hace referencia al vector de variables angulares del brazo robótico, que en este ejemplo es $x=[\alpha, \beta]$) a la que se le da unos ángulos iniciales $x^0=[\alpha^0, \beta^0]$:

$$P_{mano}(P_x, P_y) = F(x^0), \quad x^0 = [\alpha^0, \beta^0] \qquad .$$

Equation 13: Función de cinemática directa

El segundo paso será marcar la meta: llevar la mano al punto objetivo $P_{target}(T_x, T_y)$. Para ello se necesita encontrar unos ángulos $x^t = [\alpha^t, \beta^t]$ que cumplan con la ecuación $F(x^t) = P_{target}(T_x, T_y)$. Dicho de otra forma, se necesita encontrar unos ángulos $x^t = [\alpha^t, \beta^t]$ que minimicen una función de error, definida como:

$$\epsilon(\epsilon_x, \epsilon_y) = P_{target}(T_x, T_y) - P_{mano}(P_x, P_y) = (T_x - P_x, T_y - P_y)$$

Equation 14: Función de error a minimizar

Esta expresión del error se corresponde con $\epsilon = P_{target} - F(x)$.

Para el siguiente paso aplicaremos la base del algoritmo de Levenberg-Marquardt. Según éste, para incrementos pequeños en los ángulos x=[α, β] (que los denominaremos como Δx=[Δα, Δβ]), la función F(x+Δx)=F([α+Δα, β+Δβ]) se aproxima a la suma de la función de los ángulos originales, F(x), más el producto de la matriz Jacobiana de dicha función por los incrementos en los ángulos:

$$F(x + \Delta x) \simeq F(x) + J \cdot \Delta x \qquad .$$

Equation 15: Base de Levenberg-Marquardt

Si observamos adecuadamente nos daremos cuenta de que, además de ser ésta una expresión idéntica a la ecuación $S(a \pm \Delta a) = S(a) + \epsilon$ del método de Gauss-Newton, también hemos obtenido una fórmula igual a la del polinomio de Taylor de primer grado, $P_1(x) = G(a) + G'(a) \cdot (x - a)$ que para funciones multivariantes, donde x es el vector $x = [x_1, x_2, \ldots x_n]$, quedaría como:

$$P_1(x_1, x_2 \ldots, x_n) = G(a_1, a_2, \ldots a_n) + \left(\frac{\partial G}{\partial x_1}\right)(x_1 - a_1) + \left(\frac{\partial G}{\partial x_2}\right)(x_2 - a_2) + \ldots + \left(\frac{\partial G}{\partial x_n}\right)(x_n - a_n)$$

De ésta forma, nuestra función multivariante F(x+Δx)=F([α+Δα, β+Δβ]) se corresponde con el polinomio $P_1(x = [x_1, x_2])$, siendo $x_1 = \alpha + \Delta\alpha$ y $x_2 = \beta + \Delta\beta$; la función de los ángulos de partida, F(x=[α, β]) se corresponde con la función original multivariante $G(x = [a_1, a_2])$, siendo $a_1 = \alpha$ y $a_2 = \beta$; la matriz jacobiana de la función F(x=[α,

β]), $J=\left(\frac{\partial F}{\partial \alpha} \quad \frac{\partial F}{\partial \beta}\right)$, se corresponde con $\left(\frac{\partial H}{\partial b_x} \quad \frac{\partial H}{\partial b_y}\right)$, las primeras derivadas parciales de la función $G(x=[x_{1,}x_2])$ en forma matricial; y el vector de incrementos Δx=[Δα, Δβ] se corresponde con las diferencias (x_1-a_1) y (x_2-a_2), como podemos comprobar:

- Si $b_x=\alpha+\Delta\alpha$ y $a_1=\alpha$, entonces $(x_1-a_1)=((\alpha+\Delta\alpha)-\alpha)=\Delta\alpha$
- Si $x_2=\beta+\Delta\beta$ y $a_2=\beta$, entonces $(x_2-a_2)=((\beta+\Delta\beta)-\beta)=\Delta\beta$

El objetivo del algoritmo de Levenberg-Marquardt será conseguir un vector de incrementos Δx=[Δα, Δβ] que minimice la función de error y aproxime la mano del robot al punto objetivo Ptarget. Esto se representa matemáticamente como:

$$\|P_{target}-F(x+\Delta x)\|\simeq\|P_{target}-F(x)-J\cdot\Delta x\|$$

Equation 16: Aproximación a la función de error

Donde $P_{target}-F(x)$ es la función de error que hemos definido en el segundo paso, por lo que la fórmula queda finalmente $\|\epsilon-J\cdot\Delta x\|$

Para minimizar el error debemos conseguir que la expresión $J\cdot\Delta x-\epsilon$ sea ortogonal a la columna espacio de la matriz J, es decir, que su producto escalar sea 0. Para ello aplicamos la fórmula $J^t(J\cdot\Delta x-\epsilon)=0$. Al despejar el vector de incrementos Δx=[Δα, Δβ] de ésta última expresión, obtenemos la ecuación final que vamos a utilizar: $J^t\cdot(J\cdot\Delta x-\epsilon)=0$ es lo mismo que $J^t\cdot J\cdot\Delta x-J^t\cdot\epsilon=0$. Pasamos el error a la derecha: $J^t\cdot J\cdot\Delta x=J^t\cdot\epsilon$. Finalmente obtenemos las ecuaciones normales:

$$\Delta x=(J^t\cdot J)^{-1}\cdot J^t\cdot\epsilon \quad .$$

Equation 17: Cálculo de los incrementos.

Siendo $J^t\cdot J$ la aproximación al hessiano, la matriz de segunda derivadas parciales, que propone el método de Gauss-Newton: $H=J^t\cdot J$, cuadrada y con posibilidad de invertirse si tiene rango completo, y $J^t\cdot\epsilon$ la dirección de descenso de la función: $J^t\cdot(P_{target}-F(x))=J^t\cdot P_{target}-\boldsymbol{J^t\cdot F(x)}=J^t\cdot P_{target}-\boldsymbol{g}(x)$

4.4.2 Estructura del algoritmo de Levenberg-Marquardt

La estructura codificada del algoritmo de Levenberg-Marquardt se basa en la descrita en el documento "*SBA: A Software Package for Generic Sparse Bundle Adjustment*", de Manolis I. A. Lourakis y Antonis A. Argyros, con ligeras variaciones al adaptarla al código C++ del componente:

Entrada: función vectorial de cinemática directa $f: R^m \rightarrow R^n$ con $n \geq m$, un vector de coordenadas objetivo $X_{target} \in R^n$ y un vector de ángulos iniciales estimados de las articulaciones $p_0 \in R^m$

Salida: Un vector de ángulos para las articulaciones $p^{final} \in R^m$ que minimiza la función de error $\|X_{target} - f(p)\|^2$

Algoritmo:

```
Constantes ε1, ε2, ε3, ε4, kmax y τ.
  k:=0;   ν:=2   p:=p0
  H:=J^t·J;   ε_p:=X_target−f(p);   g:=J^t·ε_p;
  stop:=(‖g‖∞≤ε1);   μ:=τ·max_{i=1,...m}(H_ii);
while(not stop) and (k<kmax)
      k:=k+1
    repeat
          SOLVE  (H+μ·I)·δ_p=g;
          if  ‖δ_p‖≤ε2·(‖p‖+ε2)
            //Si los incrementos son despreciables los ignoramos
              stop:=true;
          else
            //Paso intermedio para poder deshacer cambios
              p_new:=p+δ_p;
            //ρ  se  calcula  como  la  diferencia  de  las  normas
            //entre el error antiguo y el error nuevo:
              ρ:=(‖ε_p‖²−‖X_target−f(p_new)‖²);
            if  ρ>0
              /*Si ρ>0 → error nuevo < error antiguo, hay mejora*/
                stop:=(‖ε_p‖−‖X_target−F(p_new)‖)<ε4·‖ε_p‖;
                p=p_new;  //aplicamos cambios
                H:=J^t·J;   ε_p:=X_target−f(p);   g:=J^t·ε_p;
                stop:=(stop)or(‖g‖∞≤ε1);
                μ:=μ·max(1/3, 1−(2·ρ−1)³);   ν:=2
            else
              //Si ρ<0 → error nuevo > error antiguo, no mejora
                μ:=μ·ν   ν:=2·ν
            endif
          endif
    until( ρ>0 ) or (stop)
      stop:=(‖ε_p‖≤ε3);
endwhile
  p^final=p;
```

Donde ϵ_p es el vector de error, ρ es la relación de ganancia que nos indica si estamos acercándonos a un mínimo o no, μ es el parámetro o factor de amortiguación, g es el gradiente de descenso, y los distintos ε y τ son umbrales de cuyos valores dependerá el encontrar una solución mejor[10]:

- ε_1 es el umbral para medir la magnitud del gradiente de descenso, g.
- ε_2 es el umbral para considerar si el incremento δ_p es despreciable o no.
- ε_3 es el umbral utilizado para aceptar o no un error calculado, ϵ_p .
- ε_4 es el umbral que utiliza el algoritmo para controlar la reducción del error después de aplicar un incremento δ_p .

Pero ¿cómo se resuelve la parte `SOLVE` $(H+\mu\cdot I)\cdot\delta_p=g;$? Lo que nos interesa es encontrar el vector de incrementos δ_p por lo que al invertir la ecuación nos quedaría:

$$\delta_p=(H+\mu\cdot I)^{-1}\cdot g;$$

Donde I es la matriz identidad cuadrada, con las mismas dimensiones del hessiano. Así podemos ver que cuando el algoritmo no encuentra unos incrementos que mejoren el error, la μ aumenta dándole más peso a la diagonal del hessiano:

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

De este modo nos aseguramos que la matriz H es invertible y que descendemos hacia el mínimo. Este algoritmo es bastante completo al aplicar dos controles importantes:

- El primero de ellos es la sentencia de control `if` $\|\delta_p\|\leqslant\varepsilon_2\cdot(\|p\|+\varepsilon_2)$. Esta se encarga de comprobar si el incremento δ_p calculado es lo suficientemente grande como para ser aplicado. Despreciamos los incrementos demasiado pequeños, ahorrando tiempo de cómputo, tan necesario en robots que trabajan en tiempo real.

[10] Este algoritmo depende en cierta medida 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.

- El segundo es la sentencia `if` $\rho>0$. En el algoritmo vemos que ρ es el resultado de restar al error antiguo el error nuevo obtenido después de aplicar el incremento δ_p recien calculado: $\rho=\|\epsilon_p\|^2-\|X_{target}-f(p_{new})\|^2$. Dicho de forma más simple, esta expresión es lo mismo que $\rho=\epsilon_{old}-\epsilon_{new}$. Cuando ρ es mayor que cero significa que $\epsilon_{old}>\epsilon_{new}$ y que estamos descendiendo correctamente, por lo que se aplican los cambios y se recalculan todas las variables con las que trabaja el algoritmo. Por el contrario, si ρ es menor que cero es porque $\epsilon_{old}<\epsilon_{new}$, en este caso se desecha el incremento calculado, no se aplican los cambios y sólo se aumenta el valor del factor de amortiguación μ.

4.4.3 Ejemplo sencillo: Traslaciones

Una vez estudiadas las bases matemáticas del método, proponemos el estudio de un ejemplo sencillo en 2D, con el que poder mostrar el funcionamiento del algoritmo de Levenberg-Marquardt en una iteración, teniendo en cuenta sólo las traslaciones y olvidándonos de las rotaciones para poder simplificarlo.

Como vimos en la figura 4, nuestro brazo robótico tendrá tres nodos unidos por dos segmentos, el primero de longitud **L1**=1u. y el segundo de longitud **L2**=0.5u. Además el ángulo que forma el primer segmento con el eje X es **α** = π/4 rad. y el ángulo que forma el segundo segmento con el eje X es **σ**= π/2. Y *¿por qué definimos σ y no β, que es el ángulo que forma el segundo segmento con el primero?* se preguntará el lector. Llegados a este punto es importante dejar claro la relación que existe entre los ángulos α, β y σ antes de continuar.

4.4.3.1 Introducción a los ángulos

¿Cómo se relacionan los ángulos α y β que forman los dos segmentos del brazo robótico? Partimos de la premisa de que los ángulos se miden en sentido antihorario[11] y de que β se mide con respecto a α tal y como se muestra en la figura 10, como la

[11] En sentido horario el ángulo es negativo, y en sentido antihorario el ángulo es positivo. Esto cambiará al trabajar con las herramientas de Robocomp, donde se aplica siempre la regla de la **mano izquierda**, donde el sentido antihorario es negativo y el horario es positivo.

diferencia entre el ángulo que forma el segundo segmento con el eje X y el ángulo que forma el primer segmento con el eje X:

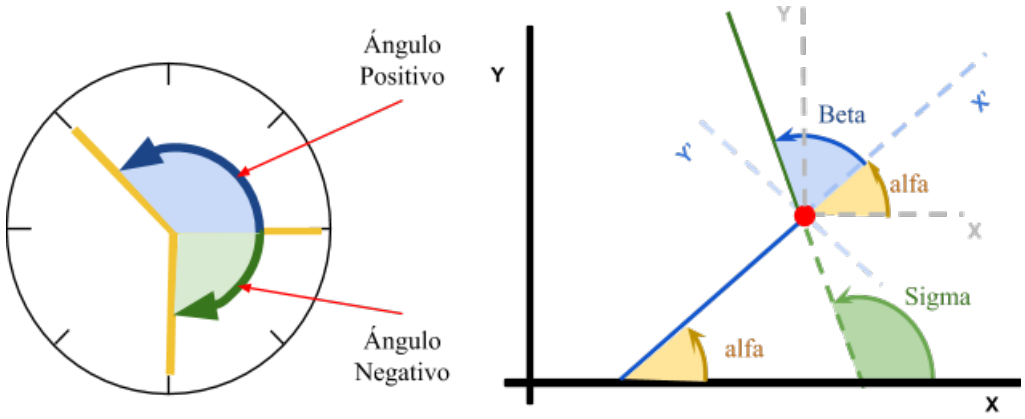


Illustration 10: Sentido de los ángulos y cálculo de β

Para medir el ángulo β lo que se hace es trasladar primero el sistema de referencia (X,Y) al nodo pintado en rojo, donde interseccionan el primer segmento azul con el segundo segmento en verde y rotarlo para alinear el eje X con la dirección del primer segmento.

De esta forma se pueden definir tres ángulos:

1. α: ángulo que forma el segmento azul con el eje X.
2. σ: ángulo que forma el segmento verde con el eje X.
3. β: ángulo que forma el segmento verde con el segmento azul.

Así obtenemos la expresión matemática que relaciona α con β: β = σ - α. Resuelto este punto continuamos con el problema de cinemática inversa.

Seguimos definiendo las características del brazo robótico. Tenemos L1, L2, α y σ. Obtenemos el valor del ángulo β que forma el segundo segmento con el primero según la fórmula que acabamos de ver: $\beta = \sigma - \alpha = \frac{\pi}{2} - \frac{\pi}{4} = \frac{\pi}{4}$ radianes.

Por otra parte situaremos las articulaciones que componen el sistema en los puntos (0, 0) para el hombro --nodo 0--, (0.7071, 07071) para el codo --nodo 1-- y (0.7071, 1.2071) para la mano --nodo 2--, usando las ecuaciones de cinemática directa:

- Para el nodo 1:
  - $x = L_1 \cdot \cos(\alpha) = 1 \cdot \cos(\frac{\pi}{4}) = 0.7071$
  - $y = L_1 \cdot \sin(\alpha) = 1 \cdot \sin(\frac{\pi}{4}) = 0.7071$
- Para el nodo 2:
  - $x = L_1 \cdot \cos(\alpha) + L_2 \cdot \cos(\alpha+\beta) = 1 \cdot \cos(\frac{\pi}{4}) + 0.5 \cdot \cos(\frac{\pi}{4}+\frac{\pi}{4}) = 0.7071$
  - $y = L_1 \cdot \sin(\alpha) + L_2 \cdot \sin(\alpha+\beta) = 1 \cdot \sin(\frac{\pi}{4}) + 0.5 \cdot \sin(\frac{\pi}{4}+\frac{\pi}{4}) = 1.2071$

Por último, el punto objetivo, Ptarget, se situará cerca del nodo 2, por ejemplo, en las coordenadas (0.6572, 1.2046). Al final nos quedaría un sistema con la situación inicial que muestra la figura 11:

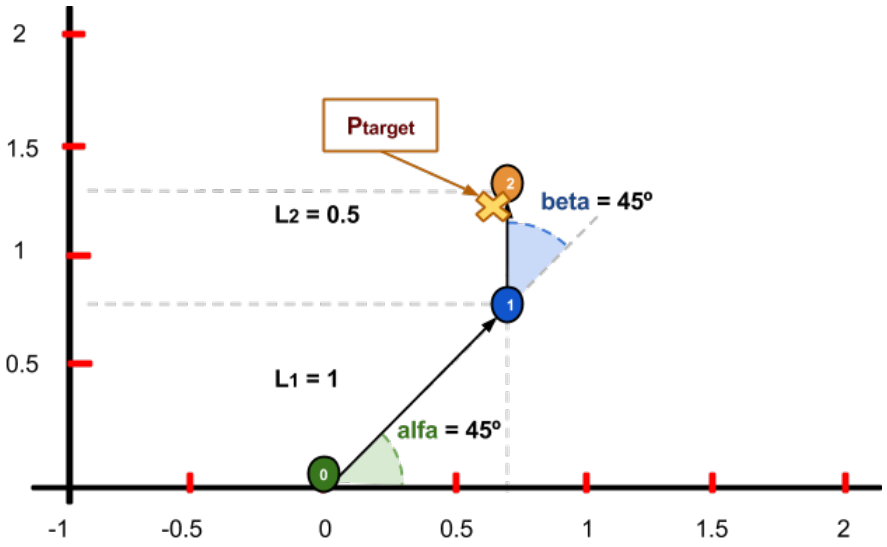


Illustration 11: Datos iniciales del problema

El objetivo es llevar el nodo 2 al punto Ptarget, por lo que nos interesa la función multivariante que nos devuelve la posición (X, Y) del nodo 2:

$$F(\alpha,\beta) = (L_1 \cdot \cos(\alpha) + L_2 \cdot \cos(\alpha+\beta),\ L_1 \cdot \sin(\alpha) + L_2 \cdot \sin(\alpha+\beta))$$

Una vez que tenemos la posición inicial de los joints, calculamos el error inicial, que será el vector diferencia entre el vector de posición del punto objetivo **Ptarget** (Tx, Ty), el punto al que queremos llegar, y el vector de posición del nodo 2, **Pnodo2** (Nx, Ny):

- $\epsilon_p = P_{target} - P_{nodo2} = P_{target} - F(\alpha, \beta)$:
  - $\epsilon_p(1) = T_x - (L_1 \cdot \cos(\alpha) + L_2 \cdot \cos(\alpha+\beta)) = 0.6572 - 0.7071 = -0.0499$
  - $\epsilon_p(2) = T_y - (L_1 \cdot \sin(\alpha) + L_2 \cdot \sin(\alpha+\beta)) = 1.2046 - 1.7071 = -0.0025$

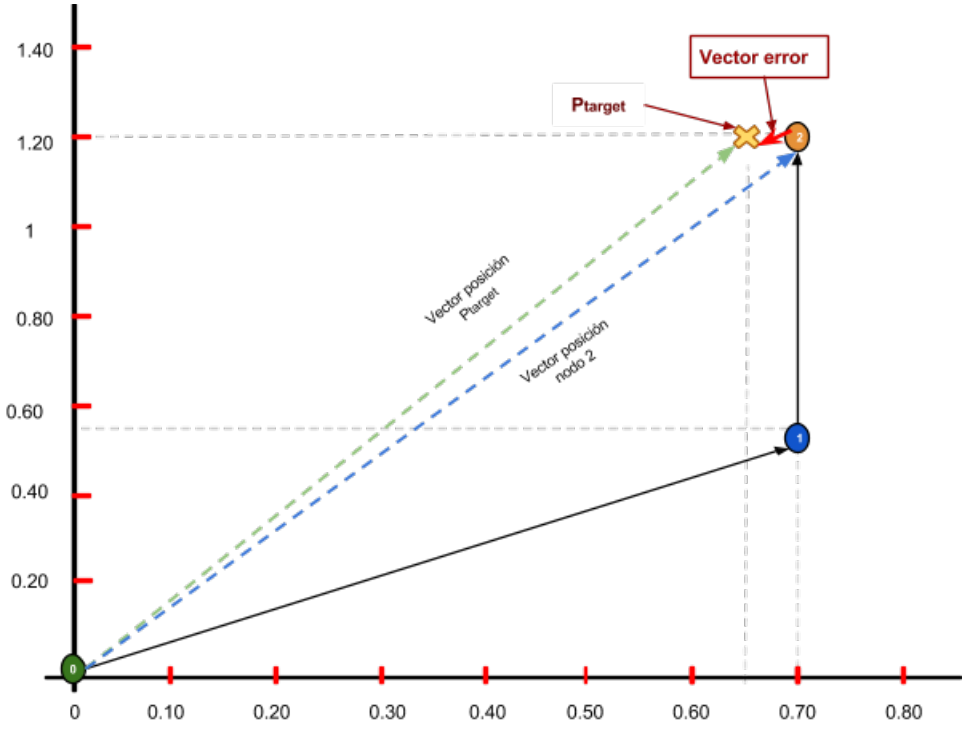


Illustration 12: Representación del vector de error entre el efector final y el target

En la figura 12 representamos gráficamente el error inicial (el error de cierre del bucle) como el vector que se corresponde con la diagonal menor del rombo formado por el vector posición del nodo 2 y el vector de posición del punto Ptarget.

Para llevar el nodo 2 al punto Ptarget debemos encontrar unos ángulos α y β que satisfagan las siguientes ecuaciones:

- $x = 1 \cdot \cos(\alpha) + 0.5 \cdot \cos(\alpha+\beta) = 0.6572$
- $y = 1 \cdot \sin(\alpha) + 0.5 \cdot \sin(\alpha+\beta) = 1.2046$
- $\epsilon_p = P_{target} - P_{nodo\,2} = P_{target} - F(\alpha, \beta) \simeq 0$

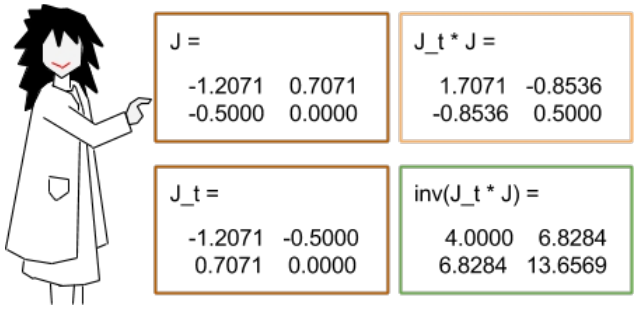
Para encontrar esos ángulos α y β aplicaremos el algoritmo iterativo de Levenberg-Marquardt, que se encargará de calcular pequeños incrementos para los ángulos

originales, α y β, hasta que la función de error sea cercana a cero. Estos incrementos se calculan como $[\Delta\alpha, \Delta\beta] = (J^t \cdot J)^{-1} \cdot J^t \cdot \epsilon_p$

Calculamos la matriz jacobiana de la función F(α, β), cuyos términos son las derivadas parciales de F en α y β:

$$\begin{pmatrix} -L_1 \cdot \sin(\alpha) - L_2 \cdot \sin(\alpha+\beta) & L_1 \cdot \cos(\alpha) + L_2 \cdot \cos(\alpha+\beta) \\ -L_2 \cdot \sin(\alpha+\beta) & L_2 \cdot \cos(\alpha+\beta) \end{pmatrix}$$

Para nuestro ejemplo, los valores de las matrices que vamos a necesitar (jacobiana, su traspuesta y la inversa del producto de la jacobiana traspuesta por la jacobiana) son:



Los incrementos se calculan del siguiente modo:

$$(J^t \cdot J)^{-1} \cdot J^t \cdot \epsilon_p = \begin{pmatrix} 4 & 6.8284 \\ 6.8284 & 13.65 \end{pmatrix} \cdot \begin{pmatrix} -1.2071 & -0.5 \\ 0.7071 & 0 \end{pmatrix} \cdot \begin{pmatrix} -0.0499 \\ -0.0025 \end{pmatrix}$$

Obteniéndose Δα=0.0050 y Δβ=-0.0621.

Actualizamos los ángulos para calcular el nuevo error y para la siguiente iteración:

- $\alpha = \alpha + \Delta\alpha = \frac{\pi}{4} + 0.005 = 0.7904\ rad.$
- $\beta = \beta + \Delta\beta = \frac{\pi}{4} - 0.0621 = 0.7233\ rad.$

El nuevo error quedaría:

- $\epsilon_p(1) = 0.6572 - (1 \cdot \cos(0.7904) + 0.5 \cdot \cos(0.7904 + 0.7233)) = -0.0749$
- $\epsilon_p(2) = 1.2046 - (1 \cdot \sin(0.7904) + 0.5 \cdot \sin(0.7904 + 0.7233)) = -0.0052$

Este sería el resultado de la primera iteración en el ejemplo propuesto. Deberíamos iterar varias veces más para alcanzar unos ángulos que acerquen el nodo 2 al *target*.

4.4.4 Pruebas en matlab

En este apartado se recogen algunas pruebas del algoritmo de Levenberg-Marquardt muy simplificado y probando sólo traslaciones en un sistema 2D, aplicadas al brazo robótico del ejemplo anterior, utilizando el entorno de desarrollo MATLAB.

El proyecto desarrollado para MATLAB se compone de dos ficheros[12]:

1. **Sistema2Nodos.m**: crea un brazo robótico parecido al de la figura 4, con dos segmentos de 1u. y 0.2u. de longitud respectivamente, formando un ángulo de $\pi/4$ rad. el primero con el eje X, y $\pi/4$ rad. el segundo segmento con el primero.
2. **CinemáticaDirecta.m**: es un fichero auxiliar que realiza el cálculo de la cinemática directa sobre un nodo. Recibe como parámetros de entrada: un **id** que indica el índice del nodo del que queremos calcular su posición dados unos ángulos, así el programa encuentra la ecuación de cinemática directa que corresponda en base a este índice y se puede extender para sistemas de más nodos y segmentos; dos ángulos, un ángulo **alfa** que forma el primer segmento con el eje X y un ángulo **beta** que forma el segundo segmento con el primero; y las longitudes de los segmentos, **L1** para el primer segmento y **L2** para el segundo segmento. Devuelve las coordenadas del punto donde se sitúa el nodo.

El algoritmo de Levenberg-Marquardt se ha simplificado lo máximo posible, suprimiendo la parte de control del descenso del error y de los incrementos pequeños (al no ser necesarias aquí), obteniendo la siguiente estructura:

Algoritmo:

$\epsilon_p := P_{target} - P_{nodo2}$;

while $(\|\epsilon_p\| > umbral)$

$$J := \begin{pmatrix} -L_1 \cdot \sin(\alpha) - L_2 \cdot \sin(\alpha+\beta) & L_1 \cdot \cos(\alpha) + L_2 \cdot \cos(\alpha+\beta) \\ -L_2 \cdot \sin(\alpha+\beta) & L_2 \cdot \cos(\alpha+\beta) \end{pmatrix};$$

$$\delta_p := (J^t \cdot J)^{-1} \cdot J^t \cdot \epsilon_p ;$$

[12] El código de estos ficheros se adjuntan como parte del proyecto al final del documento

```
        α := α+δ_p(1);
        β := β+δ_p(2);
        ε_p := P_target − CinematicaDirecta(2, α, β, L_1, L_2);
    endwhile
```

4.4.4.1 Prueba completa

Al ejecutar el programa, se simula un brazo robótico con dos segmentos de longitudes 1u. y 0.2u., formando los ángulos $\alpha^o = \beta^o = \pi/4$ rad. Los nodos del brazo se colocarán en los siguientes puntos:

- Nodo 0: se situará en el punto (0, 0).
- Nodo 1: se situará en el punto (0.707, 0.707)
  - $x_1 = 1 \cdot \cos(\frac{\pi}{4}) = 0.707106781$,
  - $y_1 = 1 \cdot \sin(\frac{\pi}{4}) = 0.707106781$.
- Nodo 2: se colocará en el punto (0.707, 0.907)
  - $x_2 = 1 \cdot \cos(\frac{\pi}{4}) + 0.2 \cdot \cos(\frac{\pi}{4} + \frac{\pi}{4}) = 0.707106781$
  - $y_2 = 1 \cdot \sin(\frac{\pi}{4}) + 0.2 \cdot \sin(\frac{\pi}{4} + \frac{\pi}{4}) = 0.907106781$

Para calcular un punto objetivo, Ptarget, cercano al nodo 2, se calculan las coordenadas (X, Y) del Ptarget mediante las fórmulas de cinemática directa para el nodo 2 añadiendo una ligera perturbación de 0.05 radianes a los ángulos originales α^o y β^o:

Ptarget = **CinematicaDirecta**(2, α^o+**incAlfa**, β^o+**incBeta**, L1, L2)

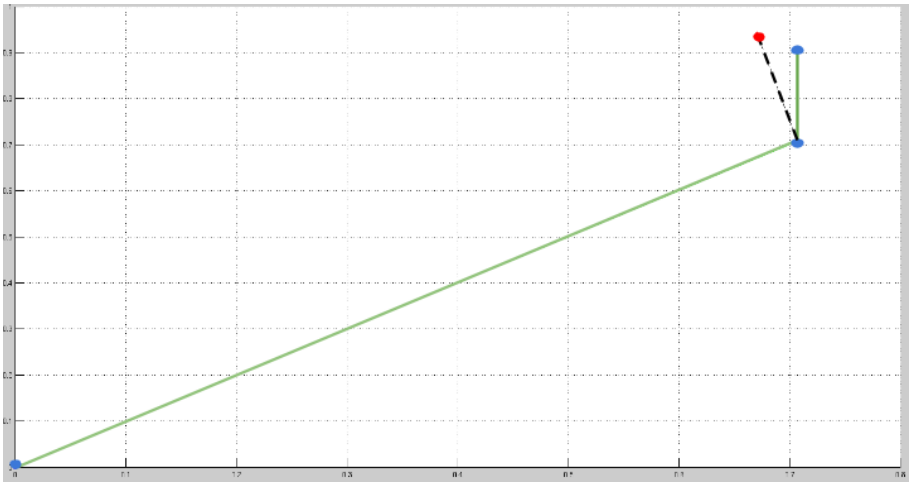
Así, los ángulos perturbados serán:

- α perturbado, α^p = π/4 + 0.05 = 0.835398163 radianes (47.86º).
- β perturbado, β^p = π/4 - 0.05 = 0.74 radianes (42.14º).

Si sustituimos los ángulos en la fórmula correspondiente de la cinemática directa, el Ptarget se sitúa en el punto (0.670, 0.942):

- $P_{target}(1) = 1 \cdot \cos(0.835398163) + 0.2 \cdot \cos(0.835398163 + 0.74) = 0.66996$
- $P_{target}(2) = 1 \cdot \sin(0.835398163) + 0.2 \cdot \sin(0.835398163 + 0.74) = 0.94156$

La situación de partida será la que muestra el primer gráfico de MATLAB, donde los puntos azules representan los nodos 0, 1 y 2, y el punto rojo es el Ptarget:



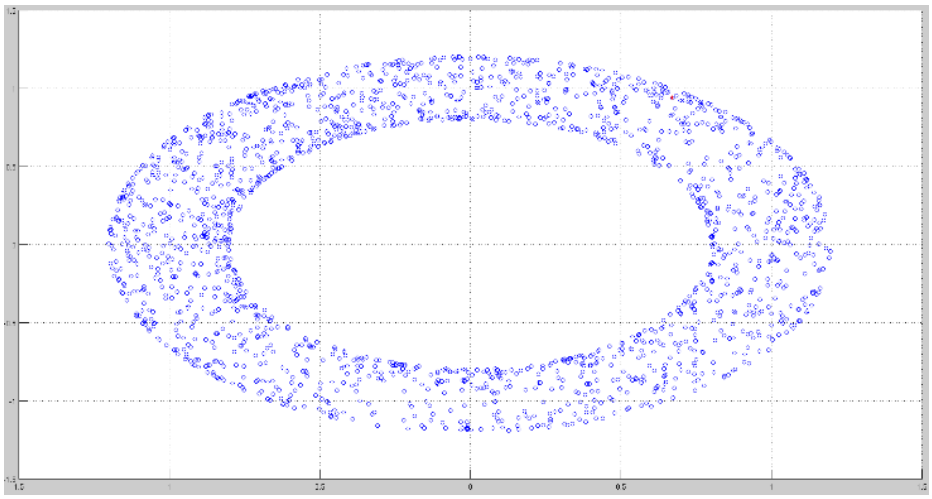
Graphic 1: Situación inicial del problema

El objetivo será igualar el α original al α perturbado, $\alpha^0 \rightarrow \alpha^p$, y el β original al β perturbado, $\beta^0 \rightarrow \beta^p$, así conseguiremos que el nodo 2 se sitúe en el mismo punto que Ptarget. Para ello necesitamos minimizar la función de error que nos calcula ϵ_p . Hasta que consigamos un error cuya norma sea menor a un umbral, que hemos fijado a 0.01, repetiremos el mismo proceso:

1. Crear la matriz jacobiana derivando la función de cinemática directa entre α y β, es decir, calculamos las expresiones de las derivadas parciales de la función de cinemática inversa derivando primero entre α y después entre β.
2. Calcular los incrementos siguiendo la fórmula del algoritmo de Levenberg-Marquardt.
3. Actualizar los ángulos, sumando al antiguo valor de éstos el nuevo incremento.
4. Calcular el nuevo error con los nuevos ángulos.

Tras 1 933 iteraciones, el nodo 2 ha recorrido o visitado todas las posiciones que vemos marcadas en azul en el gráfico 2 (donde también podemos ver el punto objetivo marcado en rojo), hasta encontrar en la iteración 1 934 una posición en la que el error es

aceptable, al estar por debajo del umbral impuesto (la norma del vector de error para la posición encontrada, $\epsilon_p = (-0.0033, 0.0050)$ es **0.00599**):



Graphic 2: Posiciones visitadas por el nodo 2

Los ángulos finales que alcanzan la posición objetivo son:

- $\alpha^f = 0.827939\,\text{rad.}$, unos 47.44º. El ángulo α perturbado, α^p , era de 0.835398163rad., por lo que tiene aproximadamente un error de 0.835398163 - 0.827939 = **0.007459163 radianes** (0.43º).
- $\beta^f = 0.758398\,rad.$ (43.45º). El ángulo β perturbado, β^p , era 0.74rad., por lo que tiene aproximadamente un error de 0.74-0.758398=**−0.018398 radianes** (-1.05º).

De esta forma, siguiendo los cálculos de cinemática directa, los nodos quedarán situados en las siguientes posiciones:

- Nodo 0: en el punto (0, 0).
- Nodo 1: en el punto...
  - $x_1 = 1 \cdot \cos(0.827939) = 0.676$
  - $y_1 = 1 \cdot \sin(0.827939) = 0.7365$
- Nodo 2: en el punto...
  - $x_2 = 1 \cdot \cos(0.827939) + 0.2 \cdot \cos(0.827939 + 0.758398) = 0.6733$

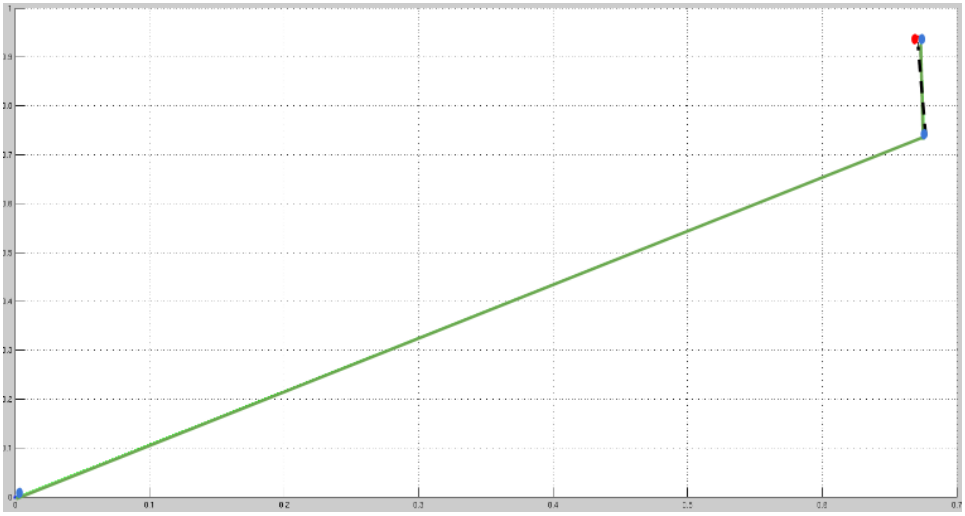
- $y_2 = 1 \cdot \sin(0.827939) + 0.2 \cdot \sin(0.827939 + 0.758398) = 0.9365$

Si calculamos el error manualmente siguiendo las ecuaciones con estos datos que hemos obtenido tendríamos $P_{target} - P_{nodo2} = (-0.003327193, 0.005045265)$, que coincide con el error calculado por MATLAB.

También podemos comprobar si las longitudes de los dos segmentos se mantienen estables con las nuevas posiciones calculadas:

- $L_1 = \sqrt{0.676395202^2 + 0.736538886^2} = 1\text{u}.$
- $L_2 = \sqrt{(0.673287193 - 0.676395202)^2 + (0.936514735 - 0.736538886)^2} = 0.2\text{u}.$

El resultado final se muestra en la gráfica 3, donde vemos que el nodo 2 se aproxima al Ptarget aunque no consigue cerrar el bucle completamente, tal y como esperábamos:



Graphic 3: Situación final después de ejecutar el programa

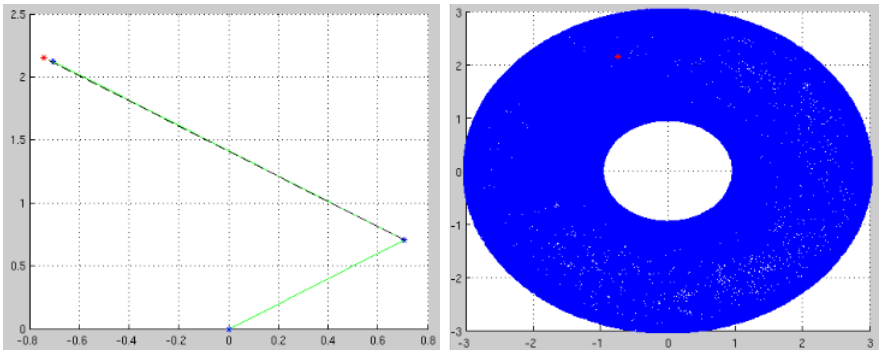
En esta gráfica podemos comprobar que, al no existir una lógica que controle el correcto descenso del error en el algoritmo aplicado en MATLAB (si miramos el algoritmo simplificado, siempre se calculan unos incrementos y siempre se aplican, aunque el error ascienda al alejarse el nodo 2 del Ptarget) el resultado no es todo lo preciso que desearíamos. Esto se arreglará más tarde al pasar el código a C++, donde añadiremos las dos restricciones originales del algoritmo de Levenberg-Marquardt: 1) desechar los incrementos pequeños y 2) aplicar sólo aquellos incrementos con los que el error

desciende, además de añadirle otras nuevas que explicaremos en los siguientes apartados.

4.4.4.2 Otras pruebas

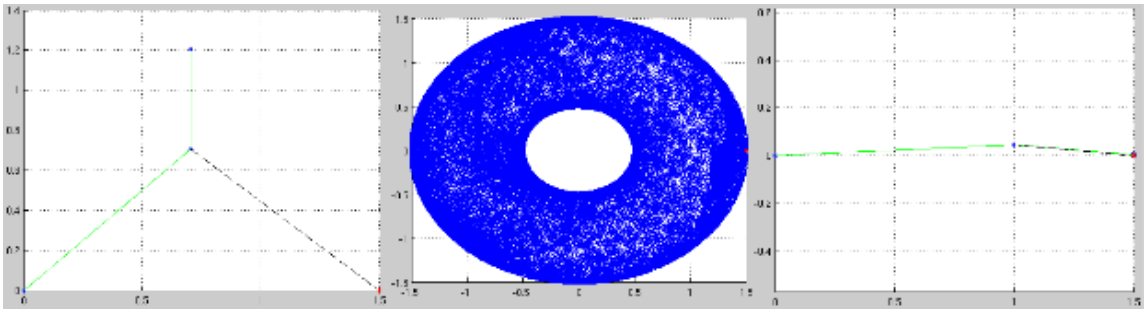
Se han ejecutado varias pruebas más para comprobar el funcionamiento del algoritmo en casos conflictivos, como aquellos en los que el punto objetivo está fuera del alcance, los segmentos del robot miden lo mismo o están alineados.

- Punto *target* dentro del alcance del brazo del robot, el cual consta de dos segmentos con longitudes L1=1u. y L2=2u., ángulos $\alpha=\pi/4$rad. y $\beta=\pi/2$rad.. Tras 56 221 iteraciones el algoritmo no fue capaz de encontrar una solución con un error cuya norma fuera menor o igual al umbral fijado en 0.01. La situación de partida y los puntos que visitó el nodo 2 son los mostrados por la gráfica 4:



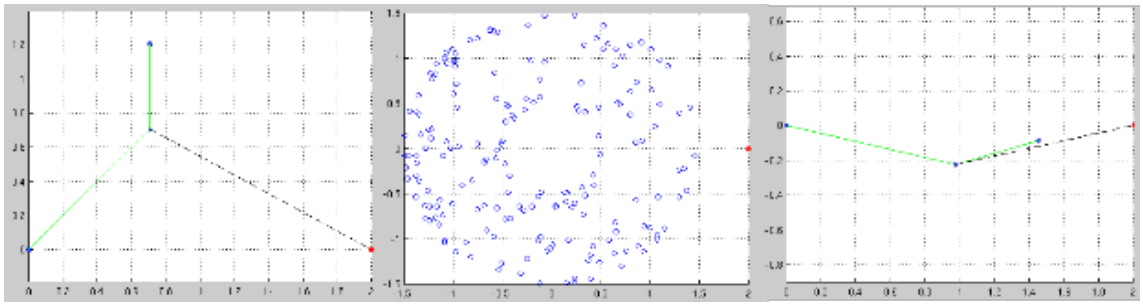
Graphic 4: Situación inicial y puntos visitados

- Brazo con dos segmentos de longitudes L1=1u. y L2=0.5u., ángulos $\alpha=\pi/4$ y $\beta=\pi/4$. El punto Ptarget se aleja más del nodo 2, al punto (1.5, 0). Después de 23 826 iteraciones alcanza una solución:
    - ÁNGULOS FINALES: α= 0.046446, β= 6.154121
    - ERROR FINAL: (0.002784, -0.005168)
    - A continuación se muestran la situacion de partida, los puntos visitados por el nodo 2 y la situación final:



Graphic 5: Situación inicial, puntos visitados y situación final

- Brazo con dos segmentos de longitud L1=1u. y L2=0.5u., ángulos $\alpha=\pi/4$ y $\beta=\pi/4$. El punto Ptarget se aleja fuera del alcance del nodo 2, al punto (2, 0). Con un umbral para el error de 0.01, no alcanza la solución después de 135.201 iteraciones. Pero si el umbral sube a 0.6 sí encuentra una solución después de 184 iteraciones:
    - ÁNGULOS FINALES: $\alpha = 6.059734$, $\beta= 0.499497$
    - ERROR FINAL: (0.543791, 0.085320)



Graphic 6: Situación de partida, puntos visitados y situación final

5. *InverseKinematicsComp*: un componente para la solución de problemas de cinemática inversa.

Después de estudiar el algoritmo de Levenberg-Marquardt y probarlo sobre MATLAB, pasaremos a explicar el desarrollo en C++ del componente *inverseKinematicsComp*, que, valiéndose de las ayudas proporcionadas por las herramientas de Robocomp, se encarga de aplicar la cinemática inversa a distintas partes del cuerpo de un robot, como los brazos izquierdo y derecho o la cabeza.

Este estudio se dividirá en varias fases (coincidiendo con las fases del desarrollo natural del proyecto) en las que se irá incrementando la dificultad del componente.

5.1 Introducción a Robocomp

Antes de continuar con la explicación de las fases del desarrollo del proyecto, conviene explicar en profundidad el frame de código abierto **Robocomp**, por ser éste el entorno donde se ha llevado a cabo toda la codificación en C++ del algoritmo de Levenberg-Marquardt, así como sus pruebas y ensayos.

Como dijimos en la introducción, Robocomp es un framework de código abierto que lleva desarrollándose por el departamento de robótica de la Universidad de Extremadura, **Robolab**, desde el 2005[13]. Está pensado para ser usado en el campo de la robótica y se apoya en el paradigma de la *Programación Orientada a Componentes*[14] (por sus siglas en inglés, COP) lo que le hace ser una herramienta fácil de entender y rápida a la hora de desarrollar nuevos códigos/programas. Además, Robocomp está basado en el middleware[15] *Internet Communications Engine* (**Ice**), una plataforma capaz de trabajar en entornos muy heterogéneos, ya sea con distintos lenguajes de programación, con

[13] Actualmente está siendo desarrollado en colaboración por las Universidades de Extremadura, Málaga, Jaén, Castilla La Macha y la Carlos III de Madrid además de la empresa Indra.

[14] La *programación orientada a componentes* es un paradigma a la hora de diseñar código dentro de la ingeniería software. Se caracteriza por descomponer el sistema programado en varios componentes lógicos, capaces de comunicarse entre sí intercambiando mensajes a través de unas interfaces bien definidas. Cada componente ofrece un determinado servicio.

[15] *Middleware*: software que actúa de soporte o ayuda a una aplicación a la hora de comunicarse o interactuar con otras aplicaciones. Es imprescindible en aplicaciones distribuidas.

distintas tecnologías de conexión de redes, con distintos sistemas operativos..., y que proporciona una serie de herramientas, APIs y soportes de librería para aplicaciones orientadas a objetos y distribuidas del tipo cliente-servidor.

Por otra parte, dentro de Robocomp existen dos elementos importantes: por un lado cuenta con una serie de herramientas y librerías de gran utilidad, que facilitan y agilizan enormemente la tarea de programar, y por otro tiene una gran variedad de componentes listos para ser integrados en nuevos proyectos de forma fácil. En los siguientes apartados explicaremos la estructura general de los componentes y nos centraremos en las herramientas y las clases más importantes de Robocomp que se han utilizado en este proyecto.

5.1.1 Estructura de los componentes: DSLEditor

Todos los componentes de Robocomp se caracterizan por seguir una misma estructura gracias al uso de la herramienta **DSLEditor**. Esta herramienta es la encargada de generar el esqueleto de los componentes y se basa en cinco tipos de ficheros (CDSL, IDSL, PDSL, DDSL e InnerModelDSL) de los cuales los que más nos interesan, por haberlos usado a lo largo del proyecto, son tres:

- **CDSL**: *Component Description Specific Language*, es un archivo DSL (*Domain Specific Language*[16]) que permite a los usuarios crear, mantener y/o modificar las descripciones de un componente. Estos ficheros contienen la definición básica de la estructura del componente, es decir los proxies de comunicación, el lenguaje de programación, las interfaces y los tópicos usados por el componente además de otras cosas como el soporte gráfico de Qt, dependencias con otras clases o librerías...

[16] Un DSL, o Lenguaje Específico del Dominio, es una especificación de un lenguaje cuyo objetivo es resolver o representar un problema determinado y suministrar un método para solucionar dicho problema. Los DSL se clasifican en: **textuales**, basados en una gramática como SQL para bases de datos o R para problemas de estadística; y **visuales**, basados en elementos gráficos como Logo.
Los lenguajes de programación de propósito general como C, C++ o Java, así como los lenguajes de modelado de propósito general como UML, no son considerados DSL ya que, como indica su nombre, estan pensados para usarse en cualquier tipo de problema y no se dedican a resolver uno específico.

- **IDSL**: *Interface Description Specific Language*. Sirve para definir las interfaces por las que se comunicarán los componentes y sus tópicos, tipos básicos de datos, tipos enumerados, excepciones...
- **InnerModelDSL**: este tipo de fichero se usa para describir toda la cinemática del robot y representa su modelo interno. Está basado en ficheros de tipo XML y mediante la herramienta **rcisInnerModel** es convertido en una clase C++ llamada *InnerModel*, que estudiaremos más en profundidad en el siguiente apartado.

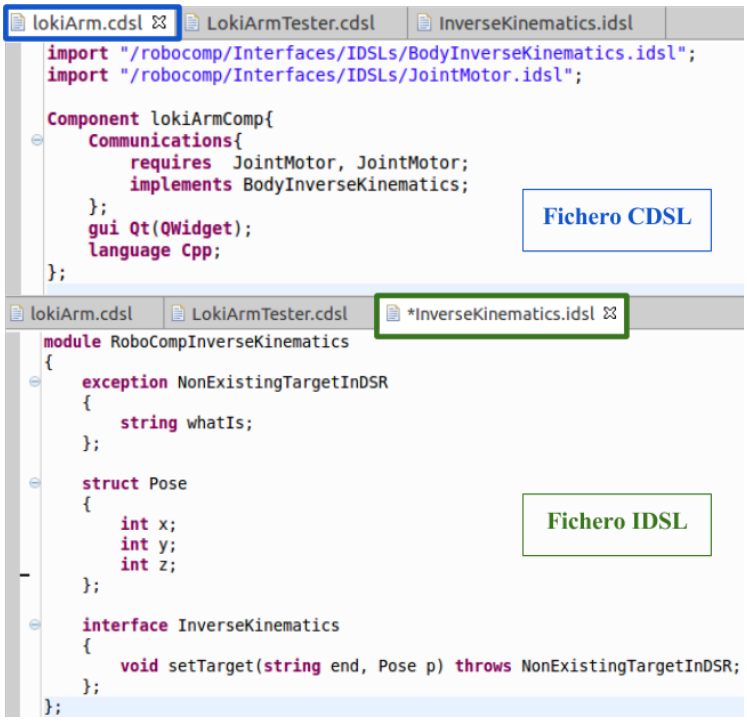


Illustration 13: Ficheros CDSL e IDSL utilizados en el proyecto

La figura 13 presenta los dos ficheros, CDSL e IDSL, utilizados en este proyecto de cinemática inversa. En el CDSL podemos ver, precedidos de la palabra reservada **requires**, los elementos que necesitará nuestro componente para comunicarse, como la interfaz del *JointMotor* que sirve para obtener, actualizar y modificar los valores de las articulaciones del robot y que, además, está repetido, por lo que se generarán dos joint_motor_proxy, pensados para 1) poder comunicarse con el innerModel y 2) con el

robot real. También podemos ver, precedidas de la palabra reservada **implements**, las interfaces que implementará el componente, que en este caso es la interfaz *BodyInverseKinematics*. También podemos ver que hará uso de una interfaz gráfica de usuario o GUI (*Graphical User Interface*) de Qt y que está programado en lenguaje C++. Mientras tanto, en el fichero IDSL podemos ver cómo se declara un tipo de excepción con la palabra reservada **exception**, una estructura declarada por la palabra **struct**, y una interfaz de comunicación. Vemos que para crear una interfaz primero se define el nombre de la interfaz, *InverseKinematics*, precedido de la palabra **interface** y después se especifican los métodos de que dispondrá la interfaz entre llaves {}.

Por otra parte tenemos los ficheros PDSL y DDSL. El fichero **PDSL** (*Parameter Definition Specific Language*) define una plantilla de parámetros de configuración que después será utilizada por el DSLEditor para generar estructuras de código específicas que se incluyen en las clases genéricas del componente. Y por último, el fichero **DDSL** (*Deployment Description Specific Language*) se encarga de describir qué componentes se van a usar, cómo se deben ejecutar y qué configuración deben usar.

Gracias a esto, gran parte del código del componente es auto-generado, simplificando mucho la tarea del programador. De esta forma cualquier componente que haya sido creado con la ayuda de la herramienta DSLEditor tendrá siempre dos partes bien definidas:

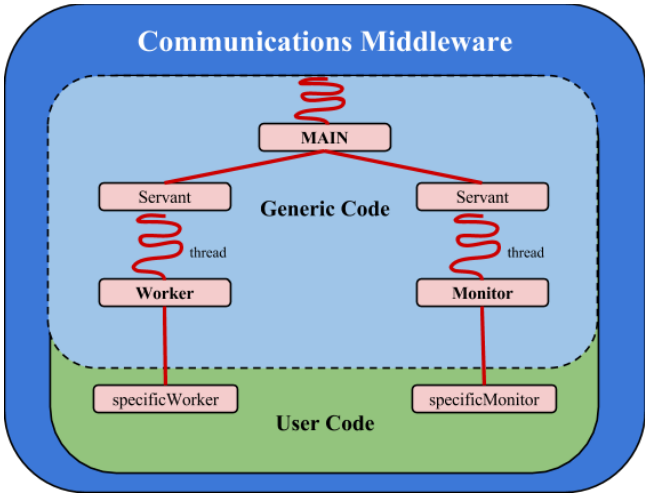


Illustration 14: Estructura general de un componente en Robocomp

Por un lado tenemos la *parte genérica*, que contiene la lógica de la comunicación entre los componentes y la estructura general del componente, como el programa principal y los subprocesos. En otras palabras, esta parte contiene todo el código genérico del componente. Está implementada con clases abstractas que son heredadas y ampliadas por el código específico del usuario. El código genérico de esta parte es regenerado cada vez que el fichero CDSL se modifica.

Y por el otro lado tenemos la *parte específica* donde se guarda el código del usuario. Esta parte es generada una sola vez por la herramienta DSLEditor. Aquí es donde el usuario puede programar su propio código, con la seguridad de que nunca va a ser borrado cuando quiera modificar alguna característica del CDSL de su componente.

El funcionamiento del componente es sencillo. El programa principal o *main* (guardado en el fichero *nombre_del_componente.cpp*) se encarga de llamar al Monitor. El Monitor es un hilo que lee los parámetros de configuración del componente guardados en el fichero de configuración, y lo inicializa, levantando a su vez el hilo del Worker. El Worker es el 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. Por otra parte, el componente dispone del *CommonBehavior*, una interfaz que permite acceder y cambiar los parámetros comunes del componente en tiempo de ejecución.

5.1.2 Librerias de Robocomp: InnerModel

Robocomp cuenta con una serie de clases y librerías destinadas a tratar temas tan variados como la visión por computador, el cálculo de matrices, acceso a hardware, widgets gráficos, lógica difusa...

Entre todas ellas, ***InnerModel*** (Modelo Interno) es, sin lugar a dudas, el núcleo de Robocomp. Esta clase se encarga de representar toda la cinemática del robot, es decir, se ocupa de la representación del cuerpo del robot y del mundo en el que trabaja.

InnerModel supone un gran apoyo a la hora de realizar los cálculos algebraicos relacionados con las transformaciones para pasar de un sistema de referencia a otro, para los cuales utiliza la librería de matemáticas desarrollada para Robocomp por Robolab,

QMat, y sus métodos han sido ampliamente utilizados durante el desarrollo de este proyecto.

Pero antes de explicar en profundidad cómo funciona y en qué se basa esta clase, es necesario introducir al lector en el sistema de referencia y otros conceptos que utiliza Robocomp (y por extensión *InnerModel*) para luego poder comprender correctamente el funcionamiento del componente implementado, *inverseKinematicsComp*.

5.1.2.1 Sistema de referencia de Robocomp

A la hora de utilizar un sistema de referencia existen dos convenios posibles de aplicar:

- La regla de la **mano derecha**: gráficamente, utiliza los dedos pulgar, índice y corazón de la mano derecha, colocados perpendicularmente entre sí, para representar los tres ejes del sistema cartesiano en 3D. Además, el sentido de giro de los ejes es positivo en sentido antihorario y negativo en sentido horario. Este sistema fue el utilizado durante las pruebas del algoritmo de Levenberg-Marquardt sobre MATLAB, en el apartado 4.4.4.
- La regla de la **mano izquierda** o regla de Flemming: al igual que la regla anterior, utiliza los dedos pulgar, índice y corazón perpendiculares entre sí para representar los ejes cartesianos, pero esta vez usa la mano izquierda. Es el convenio utilizado por el sistema de referencia de Robocomp:

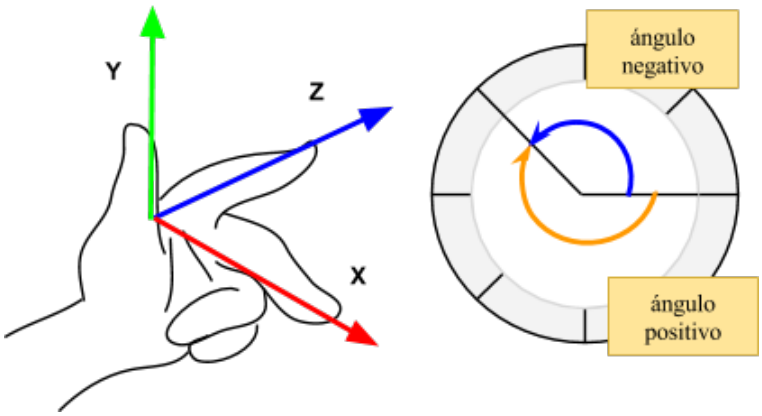


Illustration 15: Regla de la mano izquierda. Sentido de los ángulos

Con esta norma, el sistema de coordenadas utiliza los ejes (X, Y, Z), midiendo las traslaciones en milímetros, y el sentido de los ángulos, medidos en radianes, cambia: es negativo en sentido antihorario y positivo en sentido horario:

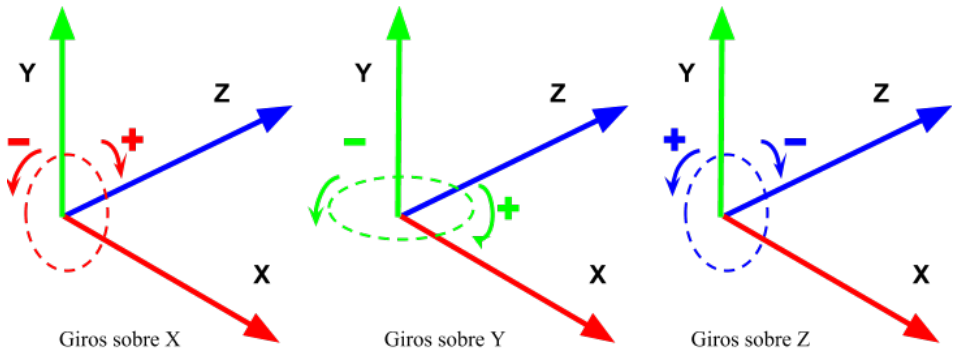


Illustration 16: Sentido de giro de cada eje de rotación

Además, para elementos cuya rotación es compuesta, porque están rotados en más de un eje, se sabe que el orden al aplicar las rotaciones es $R_x \cdot R_y \cdot R_z$, primero se rota en X, después en Y y por último en Z, de tal forma que la matriz de rotación resultante es:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} \cos\beta\cdot\cos\gamma & -\sin\gamma\cdot\cos\beta & \sin\beta \\ \sin\alpha\cdot\sin\beta\cdot\cos\gamma+\sin\gamma\cdot\cos\alpha & -\sin\alpha\cdot\sin\beta\cdot\sin\gamma+\cos\alpha\cdot\cos\gamma & -\sin\alpha\cdot\cos\beta \\ \sin\alpha\cdot\sin\gamma-\sin\beta\cdot\cos\alpha\cdot\cos\gamma & \sin\alpha\cdot\cos\gamma+\sin\beta\cdot\sin\gamma\cdot\cos\alpha & \cos\alpha\cdot\cos\beta \end{pmatrix}$$

Equation 18: Matriz de rotación completa

Esto es muy importante, ya que uno de los problemas de la cinemática inversa es rotar el efector final hasta que su sistema de referencia quede orientado del mismo modo que el sistema de referencia del *target*. Es decir, debemos conseguir que los ejes del sistema de referencia del efector final del robot (Xr, Yr, Zr) se alineen con los ejes del sistema de referencia del *target* (Xt, Yt, Zt). Para este fin se ha desarrollado a lo largo de este proyecto, dentro de **QMat**, los métodos *extractAnglesR* y *extractAnglesR_min*. Estos métodos se encargan de, dada una matriz de rotación, obtener un vector que, en el caso del primer método almacena los ángulos de rotación rx, ry y rz y sus correspondientes

contrarios, y en el caso del segundo método almacena los tres ángulos cuya norma (el camino que forman) es menor:

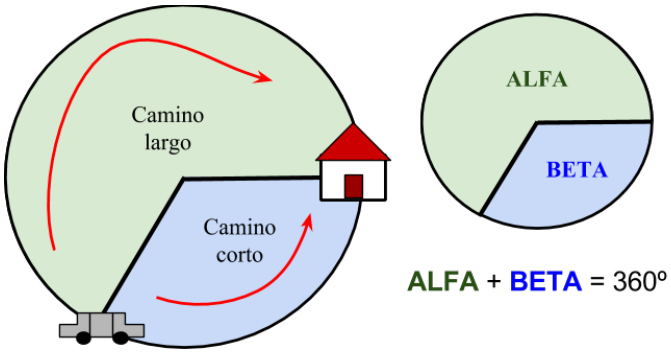


Illustration 17: Un ángulo siempre tiene un opuesto

Teniendo en cuenta la matriz de rotación anterior, para extraer los ángulos de rotación por separado, se ha seguido el método propuesto por Slabaugh en su documento "*Computing Euler angles from a rotation matrix*" en el cual se explica que:

- Para obtener el ángulo de rotación en Y, β, vemos que el elemento R_{13} , en la matriz de rotación anterior, es $\sin(\beta)$, por lo que podemos resolver el ángulo como $\beta_0=\sin^{-1}(R_{13})$. Su opuesto será $\beta_1=\pi-\sin^{-1}(R_{13})$.
- Para obtener el ángulo de rotación en el eje X, α, vemos que $\frac{R_{23}}{R_{33}}=\frac{-\sin(\alpha)\cdot\cos(\beta)}{\cos(\alpha)\cdot\cos(\beta)}=\frac{-\sin(\alpha)}{\cos(\alpha)}=-\tan(\alpha)$, de tal forma que α se calcula como $\alpha=-atan2(R_{23,}R_{33})$ [17]. Sin embargo, esta expresión depende del signo del $\cos(\beta)$: Cuando el coseno es positivo $\cos(\beta)>0\rightarrow\alpha=-atan2(R_{23,}R_{33})$, pero cuando el coseno es negativo $\cos(\beta)<0\rightarrow\alpha=-atan2(-R_{23,}-R_{33})$. Para

[17] "*public static **atan2**(y:Number, x:Number) : Number*" es un método proporcionado por la librería Math de C++. Calcula el ángulo en radianes del punto $\frac{y}{x}$ cuando x es distinto de cero, normalizando el resultado entre -π y π radianes. Este método es similar a la arcotangente, pero además, utiliza los signos de los argumentos que recibe como entrada para determinar a qué cuadrante ([0, π/2], [π/2, π], [0, -π/2], [-π/2, -π]) pertenece el ángulo calculado.

resolver este problema se utilizan las expresiones $\alpha_0 = -atan2(\frac{R_{23}}{\beta_{0,}}\frac{R_{33}}{\beta_0})$ para calcular el primer ángulo, y $\alpha_1 = -atan2(\frac{R_{23}}{\beta_{1,}}\frac{R_{33}}{\beta_1})$ para calcular su contrario.

- Para calcular el ángulo de rotación en el eje Z, γ, vemos que $\frac{R_{12}}{R_{11}} = \frac{-\cos(\gamma)\cdot\sin(\beta)}{\cos(\gamma)\cdot\cos(\beta)} = \frac{-\sin(\gamma)}{\cos(\gamma)} = -\tan(\gamma)$. Con esta expresión tenemos el mismo problema que al calcular el ángulo α. Volvemos a usar el mismo método que para calcular α: $\gamma_0 = -atan2(\frac{R_{12}}{\beta_0}, \frac{R_{11}}{\beta_0})$ y $\gamma_1 = -atan2(\frac{R_{12}}{\beta_1}, \frac{R_{11}}{\beta_1})$ para calcular los dos posibles valores del ángulo γ.

De esta forma obtenemos las dos posibles soluciones para los ángulos α, β y γ, cuando se cumple que $\cos(\beta) \neq 0.$ Sin embargo, cuando el coseno de β es cero las expresiones $-atan2(\frac{R_{23}}{\beta}, \frac{R_{33}}{\beta})$ y $-atan2(\frac{R_{12}}{\beta}, \frac{R_{11}}{\beta})$ quedan reducidas a $-atan2(\frac{0}{\beta}, \frac{0}{\beta})$, y la función atan2 no está definida cuando sus dos argumentos son cero. Cuando sucede esto debemos utilizar otras expresiones para calcular los ángulos de rotación de los ejes:

- En este caso fijamos siempre el ángulo del eje Z a cero, γ=0.
- Si $\sin(\beta) = 1$, entonces fijamos β a π/2. De esta forma α se calculará como $\alpha = atan2(R_{21,} - R_{31})$. Esta fórmula proviene de la siguiente expresión: $\frac{\sin(\alpha)\cdot\sin(\beta)\cdot\cos(\gamma) + \cos(\alpha)\cdot\sin(\gamma)}{-1\cdot(-\cos(\alpha)\cdot\sin(\beta)\cdot\cos(\gamma) + \sin(\alpha)\cdot\sin(\gamma))}$. Si sustituimos β y γ por sus valores, π/2 y 0, vemos que $\sin(\beta) = \sin(\pi/2) = 1$, $\sin(\gamma) = \sin(0) = 0$ y $\cos(\gamma) = \cos(0) = 1$, así que la expresión anterior se simplifica bastante, $\frac{\sin(\alpha)\cdot 1\cdot 1 + \cancel{\cos(\alpha)\cdot 0}}{-1\cdot(-\cos(\alpha)\cdot 1\cdot 1 + \cancel{\sin(\alpha)\cdot 0})}$, quedándonos al final: $\frac{\sin(\alpha)}{\cos(\alpha)} = \tan(\alpha)$.
- Si $\sin(\beta) = -1$, entonces fijamos β a -π/2. De esta forma podemos despejar α como $\alpha = atan2(-R_{21,} R_{31})$. Esta fórmula proviene de la siguiente expresión:

$\frac{-1\cdot(\sin(\alpha)\cdot\sin(\beta)\cdot\cos(\gamma)+\cos(\alpha)\cdot\sin(\gamma))}{-\cos(\alpha)\cdot\sin(\beta)\cdot\cos(\gamma)+\sin(\alpha)\cdot\sin(\gamma)}$. Si sustituimos β y γ por sus valores, -π/2 y 0, vemos que $\sin(\beta)=\sin(-\pi/2)=-1$, $\sin(\gamma)=\sin(0)=0$ y $\cos(\gamma)=\cos(0)=1$, al final nos pasa como antes, la expresión se simplifica,

$\frac{-1\cdot(\sin(\alpha)\cdot\sin(\beta)\cdot\cos(\gamma)+\cancel{\cos(\alpha)\cdot\sin(\gamma)})}{-\cos(\alpha)\cdot\sin(\beta)\cdot\cos(\gamma)+\cancel{\sin(\alpha)\cdot\sin(\gamma)}}$, quedando: $\frac{\sin(\alpha)}{\cos(\alpha)}=\tan(\alpha)$.

Estudiadas las bases matemáticas de los métodos *extractAnglesR* y *extractAnglesR_min*, pongamos un ejemplo gráfico de su funcionamiento.

Supongamos que tenemos la situación que muestra la figura 18, donde podemos ver el sistema de referencia del mundo donde trabaja el robot (que será nuestro sistema de referencia principal para calcular las rotaciones de los demás sistemas, el de la muñeca y el del target), el sistema de referencia de la muñeca del robot (que está rotado con respecto al mundo [0, π/2, -π/2]) y el sistema de referencia del *target,* que para simplificar el problema, hemos supuesto que su orientación es la misma que la del mundo (sus ángulos de rotación con respecto al mundo son [0, 0, 0]):

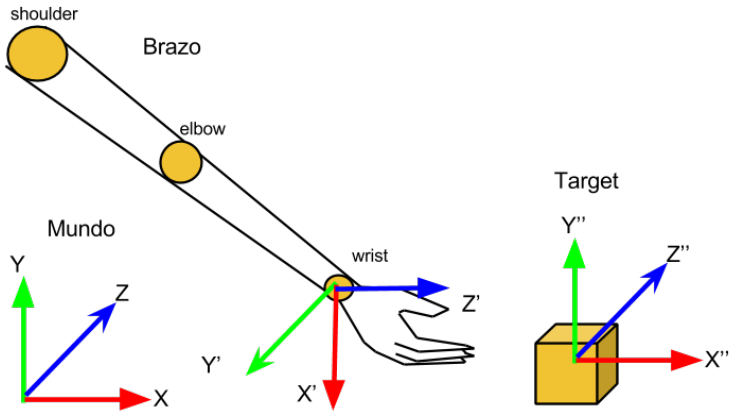


Illustration 18: Sistemas de refrencia de la situación inicial

El objetivo será orientar la muñeca tal y como está el *target,* es decir, encontrar los ángulos que deben girar los ejes de la muñeca para que se alineen con los ejes del *target*. Para ello primero calculamos la matriz de rotación del *target* en la muñeca del robot, ayudándonos de la clase *InnerModel*:

```
QMat maRot = innerModel->getRotationMatrix("muñecaRobot", "target");
```

Una vez que tenemos la matriz de rotación, extraemos el vector con los ángulos de Euler utilizando el método *extractAnglesR* o con *extractAnglesR_min*:

```
QVec anglesRot = matRot.extractAnglesR();
```

En el caso de usar *extractAnglesR* tendríamos que dividir el vector de seis elementos en dos subvectores de tres y quedamos con aquellos ángulos que definan un recorrido más corto (cuya norma sea más pequeña). Con *extractAnglesR_min* este cálculo no es necesario porque ya devuelve los ángulos menores. De esta forma obtendríamos el vector de ángulos [π/2, 0, π/2], que significa que hay que rotar el sistema de referencia de la muñeca π/2 en el eje X, 0 en el eje Y y π/2 en el eje Z, siguiendo la figura 19:

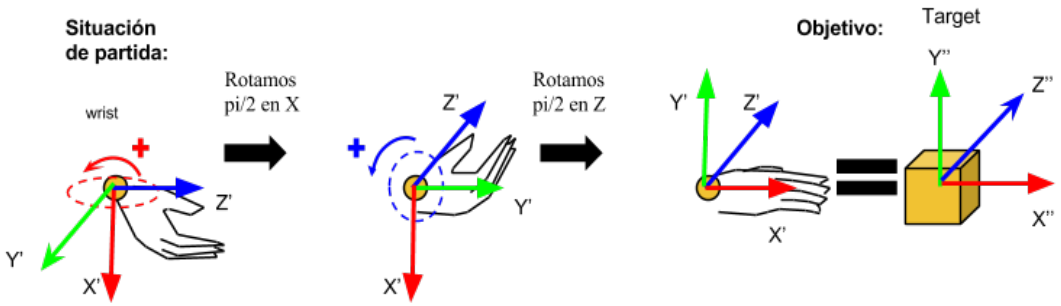


Illustration 19: Transformaciones del sistema de referencia de la muñeca

Podemos comprobar que los resultados de estos métodos son los ángulos que debe girar el sistema de origen (la muñeca del robot) para alcanzar la orientación del sistema final (la orientación del *target*). Estos métodos los utilizaremos más adelante para calcular la función de error de la cinemática inversa.

Aclarado estos puntos, retomemos las explicaciones de la clase *InnerModel* y sus funcionalidades. Como dijimos antes, es una clase en C++ que se basa, por una parte, en descripciones XML de la cinemática de un robot almacenadas en archivos con extensión .xml y, por otra parte, en una clase para actualizarlo y consultarlo. Los ficheros XML definen un **árbol cinemático** donde cada nodo contiene la transformación geométrica respecto a su padre. En este árbol se define el robot y todos los objetos con

los que interactúa. Tanto el robot como los demás elementos cuelgan de un mismo nodo, el mundo.

Pongamos el ejemplo de la figura 20, en la que podemos ver cómo del mundo cuelgan dos elementos, el robot y una mesa. A su vez el robot puede tener una cabeza y sobre la cabeza una cámara u otro sensor, como por ejemplo una kinect, y la mesa puede tener una taza de café encima:

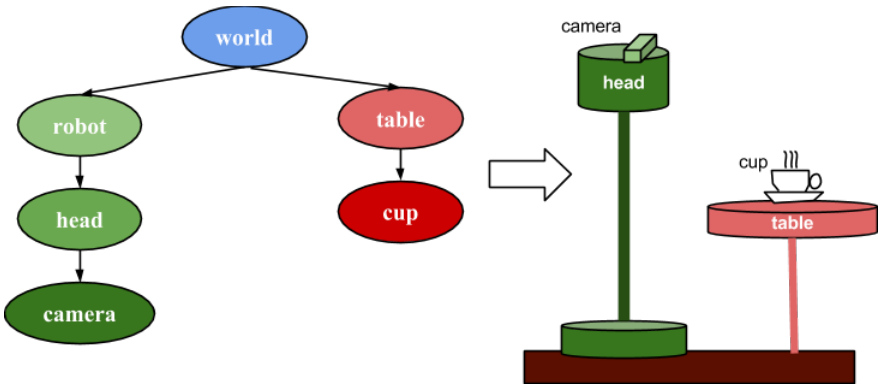


Illustration 20: Resultado visual del árbol cinemático del InnerModel

La parte de la mesa de éste árbol cinemático quedaría recogida en el fichero XML del siguiente modo (bastante simplificado):

```
<innerModel>
  <transform id="world" >
  <transform id="floor">
      // encima del suelo están la mesa y el robot //
      <transform id="table">
          <mesh id="tablemesh" file="/texturas/texturaMesa.jpg" tx="0"
          ty="100" tz="600" rx="1.57" ry="0", rz="0" scale="80" />
          // encima de la mesa hay una taza //
          <transform id="cup" tx="0" ty="700" tz="500">
            <mesh id="Cupmesh" file="/texturas/texturaCup.jpg"
            scale="15" />
          </transform> //fin de la taza
      </transform> // fin de la mesa
       <...>
  </transform> //fin del mundo
</innerModel> //fin del fichero
```

Además podemos distinguir varios tipos de nodos: todos los ficheros de la clase *InnerModel* empiezan y acaban con la etiqueta ***innerModel***; los nodos ***transform*** indican el nombre de un nuevo nodo, donde los atributos tx, ty y tz se corresponden con el

vector de traslación (x, y, z) y los atributos rx, ry y rz se corresponden con los ángulos de rotación (α, β, γ), y los nodos ***mesh*** nos sirven para cargar una malla trasladada y/o rotada con la que representar correctamente el objeto. Del mismo modo, para definir un robot que dispone de una cabeza con una cámara, el fichero XML quedaría de la siguiente manera (para simplificar omitimos las texturas):

```
<innerModel>
  <transform id="world" >
  <transform id="floor">
  // dentro del mundo están la mesa y el robot //
       <...>

    <differentialrobot id="robot" port="10004">
        <transform id="head" ty="1002" tz="-120">
          <transform id="cameraPose" ty="10"/>
          <camera id="camera" width="640" height="480"/>
          </transform>
        </transform>//fin de la cabeza
    </differentialrobot> //fin del robot
    </transform> //fin del suelo
  </transform> //fin del mundo
</innerModel> //fin del fichero
```

Existen más tipos de nodos en *InnerModel*, por ejemplo los nodos ***joint*** indican el nombre de una articulación y vienen acompañadas de los atributos ***axis***, que nos dice el eje sobre el que gira el joint, ***port*** que indica el puerto por donde podemos consultar el estado del joint mediante el proxy jointMotor, y ***min*** y ***max*** que nos señalan los límites mínimo y máximo que tiene el joint. También existen nodos para definir distintos tipos de sensores, como ***camera***, ***imu***, ***laser***...

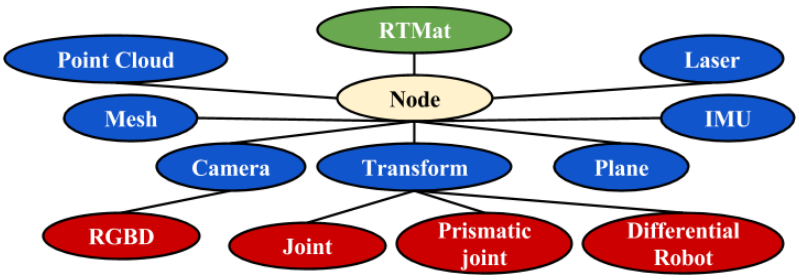


Illustration 21: Jerarquía de nodos de InnerModel

Cada nodo del *InnerModel* (definidos por las etiquetas *transform*, *joint*, *camera*, *imu*, *laser*...) especifica la transformación que tiene respecto al nodo padre del que cuelga,

transformación que depende del tipo de nodo que sea, por ejemplo un nodo de tipo *laser* simula un sensor láser y además puede transformar coordenadas cartesianas a polares y viceversa, de polares a cartesianas. Otro ejemplo, si tenemos un árbol cinemático como el de la figura 22 y queremos calcular las coordenadas de traslación del nodo C vistas desde el nodo E se sigue el siguiente procedimiento:

- FROM: se sube por el árbol desde el nodo C hasta el nodo del que cuelga todo el árbol cinemático, en este caso A, calculando las matrices directas de transformación de cada nodo hijo hacia su correspondiente nodo padre: Md_{CB} (la matriz de transformación directa de C a B) y Md_{BA} (la matriz directa de B a A).
- TO: se baja por el árbol desde el nodo A, hasta el nodo destino E, calculando las matrices inversas de transformación desde el nodo padre hacia el nodo hijo. Así, se calcula la matriz inversa de A a E, Mi_{AE} .

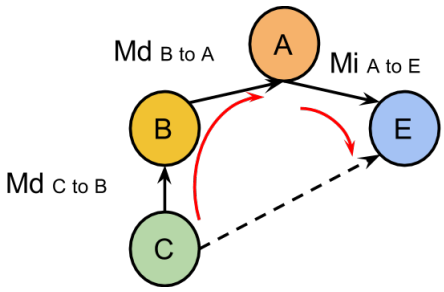


Illustration 22: Transformaciones para pasar de un sistema de referencia de un nodo a otro

Finalmente, la matriz de transformación que se utiliza para pasar del sistema de referencia del nodo C al nodo E es la resultante de multiplicar todas las matrices de transformación que acabamos de ver: $Md_{CE} = Md_{CB} \cdot Md_{BA} \cdot Mi_{AE}$. Este mismo procedimiento es seguido para pasar los ángulos de rotación del sistema C al sistema E, por el cual obtenemos la matriz de transformación $M_{CE} = \begin{pmatrix} \boldsymbol{R} & T \\ 0 & 1 \end{pmatrix}$, donde R es la submatriz de rotación 3x3 de la que se obtiene los ángulos de Euler que debe girar el

sistema C para alinearse con el sistema E, mediante el método *extractAnglesR_min* que acabamos de ver.

Por otra parte, Robocomp dispone de la herramienta **RCIS** (*Robocomp Component Inner Model Simulator*), que es el simulador de este framework. Esta herramienta instancia la clase *InnerModel*. Al ejecutarse, carga el árbol cinemático que recibe como parámetro de entrada de un fichero XML[18], permitiendo:

1. Visualizar el escenario descrito por dicho árbol cinemático. Para ello, RCIS utiliza el motor de gráficos 3D de software libre para videojuegos **osg** (*Open Scene Graph*) como soporte para poder representar gráficamente el contenido de los nodos ***mesh*** (o mallas) del árbol cinemático. Para renderizar el *InnerModel* en osg se utiliza la clase ***InnerModelViewer***, que se encarga de mantener coherente el árbol de *InnerModel* y el grafo de osg. En la figura 23 podemos ver cómo representaría el árbol cinemático del robot Loki:

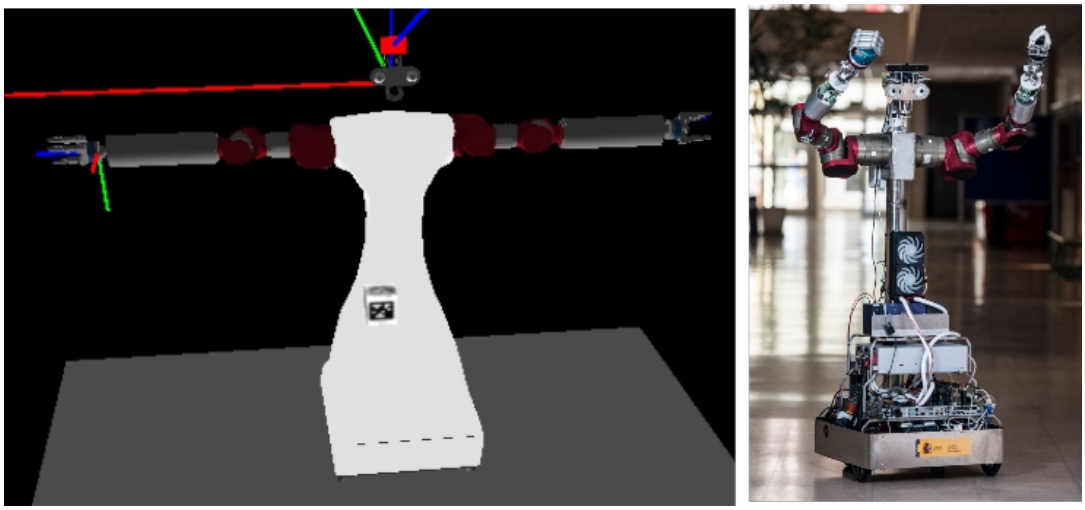


Illustration 23: Representación interna del robot Loki en RCIS

2. Ofrecer para cada nodo especial descrito en el árbol, como pueden ser los joints (motores) o los distintos tipos de sensores (láser, cámaras...), los mismos interfaces de comunicación que Robocomp utiliza para comunicarse con los

[18] Por ejemplo, para cargar en RCIS el árbol cinemático (guardado en el fichero *loki.xml*) que describe tanto la cadena cinemática del robot Loki como el mundo en el que éste trabaja se ejecutaría: *rcis loki.xml*

motores y sensores reales de un robot real. RCIS ofrece la capacidad de simularlos, no como motores o sensores reales, sino imitando el mismo comportamiento del componente físico que representan o envuelven. De esta forma y a todos los efectos, la comunicación con un componente simulado como puede ser un láser, es igual que con el dispositivo real.

Este último punto es posible gracias a que RCIS implementa las interfaces (.Ice) de los componentes reales de un robot, descritas en sus correspondientes ficheros que son cargados por el RCIS en tiempo de ejecución. Entre esas interfaces, a lo largo de este proyecto se han utilizado básicamente tres:

- **InnerModelManager**: proporciona acceso directo a todos los nodos que componen el árbol cinemático de *InnerModel*, así como a los atributos asociados a cada uno de ellos. Esta interfaz permite modificar y actualizar el árbol en tiempo de ejecución, ya sea creando o eliminando nodos del árbol, rotándolos o trasladándolos.
- **JointMotor**: simula el comportamiento de un motor real, con la posibilidad de ajustar la velocidad angular con la que se mueve el motor y poner límites o rangos en sus valores angulares.
- **RGBD**: simula el comportamiento de sensores como una Kinect o un Asus Xtion. Imita el comportamiento de cámaras RGBD a las que les añade un infrarrojo para poder calcular la profundidad de las imágenes que capta. Ésta se utilizará en una fase más avanzada del proyecto para que el robot detecte marcas objetivo a las que llevar su efector final.

Así, desde un componente cualquiera podemos usar toda la potencia del Middleware, conectándonos a través del puerto conveniente a un joint, a una cámara o a una base differential y simular su funcionamiento (sensores sintéticos).

Esto permite al usuario conectarse al RCIS o al robot real de manera fácil y rápida, pudiendo probar su algoritmo sobre el RCIS primero, puliendo errores de forma segura y comprobando que todo funciona correctamente, y después ejecutarlo sobre el robot real, aunque también hay que advertir que el hecho de que funcione correctamente sobre la simulación, no implica que vaya a funcionar sobre la realidad, precisamente por los

errores que ésta presenta, como holguras en los motores reales, límites más restrictivos, una odometría más adversa, errores de posicionamiento...

5.2 Profundizando en el componente *inverseKinematicsComp*

Para comprender las bases de este proyecto haremos una introducción a la estructura y a las clases más importantes del componente *inverseKinematicsComp*, encargado de intentar resolver los problemas de cinemática inversa de un robot, para poder comprender después los métodos y algoritmos utilizados en el proyecto.

5.2.1 Estructura del componente *inverseKinematicsComp*

Uno de los objetivos de este proyecto es desarrollar un componente que sea capaz de resolver ciertos problemas relacionados con la cinemática inversa. Hemos dividido estos problemas en tres tipos:

1. **POSE6D**: se trata de enviar un punto objetivo, con traslaciones y rotaciones, al componente para que una parte del robot lo resuelva. Por ejemplo, tomar la posición de una taza y que el brazo derecho se mueva hasta que la mano coja la taza. Es el problema típico de cinemática inversa en robótica.
2. **ADVANCEAXIS**: su objetivo es hacer que el efector final del robot se mueva a lo largo de un vector. Esto resulta muy útil para mejorar el resultado del problema anterior, por ejemplo, imaginemos que la mano se ha quedado un poco alejada de la taza. Con esta funcionalidad podemos calcular el vector de error entre el efector final, o mano, y la taza y hacer que el efector se mueva a lo largo de ese vector hasta situarlo en una posición óptima.
3. **ALIGNAXIS**: tiene como objetivo que el efector final se quede apuntando al *target*, sin trasladarse hasta él pero rotado como el *target*. Puede resultar útil en ciertos casos en los que interese más quedar el efector orientado con la misma rotación del *target*.

Para crear el esqueleto del nuevo componente encargado de resolver estos problemas, *inverseKinematicsComp*, se ha utilizado la herramienta *DSLEditor*, definiendo los ficheros de tipo IDSL y CDSL.

5.2.1.1 Fichero IDSL del componente *inverseKinematicsComp*

El fichero *BodyInverseKinematics.IDSL* se encarga de declarar los tipos y estructuras propias del proyecto. Así tenemos una excepción especial del componente, la **BIKException**, y varias estructuras propias como la **Pose6D** que define la pose de un *target,* la **WeightVector** que define un vector de pesos para traslaciones y rotaciones de un *target*, **Axis** que representa los tres ejes del sistema (X, Y y Z), o **TargetState**, una estructura que nos indica ciertos parámetros de los *targets*.

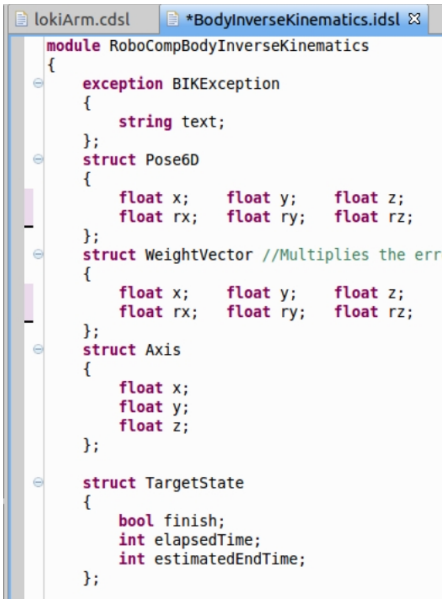


Illustration 24: Estructuras declaradas en el fichero IDSL para el proyecto

También se encarga de definir las cabeceras de los métodos de la interfaz, de las que cabe destacar las siguientes:

- **setTargetPose6D**: este método establece un *target* objetivo de tipo Pose6D para una parte del cuerpo del robot (el brazo derecho, el izquierdo o la cabeza, por ejemplo) a la que pertenece un efector final (como la mano derecha para el brazo derecho, o la izquierda para el brazo izquierdo), cuyos componentes de traslación y rotación pueden atenuarse mediante el vector de pesos de tipo WeightVector. Es la interfaz que usaremos para enviar *targets* al componente para que la cinemática inversa lo resuelva, implicando a toda la extructura.

- **pointAxisTowardsTargets**: este método de la interfaz se encarga de enviar un *target* de tipo ALINGAXIS a una parte del robot para que la cinemática oriente el efector final de esa parte del robot alineándolo con el eje que se le indique.

```
    interface BodyInverseKinematics
    {
        void setTargetPose6D(string bodyPart, Pose6D target, WeightVector weights, float radius) throws BIKException;

        void pointAxisTowardsTarget(string bodyPart, Pose6D target, Axis ax, bool axisConstraint, float axisAngleConstraint) throws BIKException;

        void advanceAlongAxis(string bodyPart, Axis ax, float dist) throws BIKException;
        void setFingers(float d) throws BIKException;
        void goHome(string part) throws BIKException;
        void setRobot(int type) throws BIKException;
        TargetState getState( string part);
    };
};
```

Illustration 25: Declaración de los métodos de la interfaz en el IDSL

- **AdvanceAlongAxis**: este método se encarga de que el efector de la parte del robot que indiquemos avance una distancia *dist* a lo largo del vector que se le de como parámetro. Es un método muy útil que se utilizará para, una vez que se haya ejecutado un target de tipo POSE6D con un error cualquiera de cierre del bucle, "afinar" el resultado de la cinemática inversa, aproximando aún más el efector final al objetivo mediante el avance del efector sobre el vector de error.
- **setFingers**: se encarga de abrir y cerrar las pinzas de las manos del robot.
- **setRobot**: este método se encarga de intercambiar el proxy de comunicación del *JointMotor* simulado en el RCIS (j*ointmotor0_proxy*, situado en el puerto 20000) por el proxy de comunicación del *JointMotor* real (*jointmotor1_proxy*, situado en el puerto 40000), y viceversa. De esta forma podemos ejecutar el componente *inverseKinematicsComp* tanto en el simulador de RCIS como en el robot real.
- **goHome**: este método lleva la parte del robot que se le indique a una posición de inicio o home.
- **Stop**: este método se encarga de abortar la ejecución en el componente *inverseKinematicsCom* 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*.

5.2.1.2 Fichero CDSL del componente *inverseKinematicsComp*

El fichero *lokiArm.CDSL* se encarga de definir, por una parte, los proxies de comunicación del componente, que en este caso son dos proxies de tipo **jointMotor**. Como explicamos en el apartado 5.1.2 cuando hablábamos sobre la herramienta RCIS, *jointMotor* es una interfaz utilizada para leer, actualizar y modificar las posiciones angulares de los joints (simulados o reales) definidos en el árbol cinemático de *InnerModel*. El componente *inverseKinematicsComp* utilizará uno de estos proxies para acceder a los joints simulados por la herramienta *RCInnerModelSimulator* y el otro para acceder a los motores del robot real.

lokiArm.cdsl | LokiArmTester.cdsl | InverseKinematics.idsl

```
import "/robocomp/Interfaces/IDSLs/BodyInverseKinematics.idsl";
import "/robocomp/Interfaces/IDSLs/JointMotor.idsl";

Component lokiArmComp{
    Communications{
        requires  JointMotor, JointMotor;
        implements BodyInverseKinematics;
    };
    gui Qt(QWidget);
    language Cpp;
};
```

Illustration 26: Fichero CDSL del proyecto

Por otra parte, el fichero CDSL indica que este componente será el encargado de implementar la interfaz definida en el fichero *BodyInverseKinematics.IDSL*. Además, el componente *inverseKinematicsComp* será codificado en C++ y utilizará una interfaz gráfica de usuario de Qt.

5.2.1.3 Ficheros y clases de *inverseKinematicsComp*

Al utilizar el *DSLEditor* se nos generan dos tipos de ficheros, unos son los genéricos (*genericworker*, *genericmonitor*, *joint*, *commonbeavior*, etc.) y otros, los específicos (*specificworker* y *specificmonitor*). Éstos últimos se generan una sola vez por la herramienta, cuando se crea por vez primera el componente, y no vuelven a ser regenerados nunca más. De los específicos nos interesan dos, el *specificworker.h* y el *specificworker.cpp*. Estos ficheros son los encargados de contener todas las funcionalidades del componente y en ellos será donde se implementarán las interfaces

definidas por el fichero IDSL y todos los procesos propios encaminados a resolver las tareas con éxito.

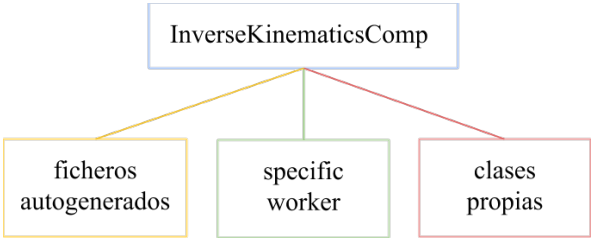


Illustration 27: Partes del componente desarrollado

Pero, aparte de estos ficheros, hemos añadido una serie de clases que nos ayudarán en el desarrollo del proyecto: 1) **Target**, 2) **Body Parts** y 3) **Cinemática inversa**.

En los siguientes apartados iremos explicando lo más relevante de estas clases, sus estructuras y sus principales métodos.

5.2.2 Estructura básica del target

Un aspecto muy importante en este proyecto es tener claro cómo son estructuralmente las posiciones objetivo o deseadas con las que trabaja el componente de cinemática inversa. Para ello hemos definido la clase ***Target***, que se encarga de definir los atributos y métodos más importantes de un punto objetivo o punto de destino.

Para empezar, los *targets* se dividen en los tres tipos que hemos visto en el apartado 5.2.1: **POSE6D**, **ADVANCEAXIS** y **ALIGNAXIS**. Todos estos tipos tienen un vector de seis elementos que define su pose en el espacio, pose6D=[tx, ty, tz, rx, ry, rz], un vector de pesos para sus coordenadas de traslación y rotación, y un *tip* o efector final asignado, de tal manera que el usuario puede decir al componente que lleve la mano derecha del robot a una posición P1, mientras que la mano izquierda se mueve a otra posición P2. Además, los *targets* tienen un estatus que indica si ha sido procesado o no, por la cinemática inversa y cómo se ha procesado:

1. **START**: indica que el *target* acaba de llegar y que aún no ha sido procesado por el algoritmo de Levenberg-Marquardt.

2. **LOW_ERROR**: indica que el *target* ha sido procesado satisfactoriamente por el algoritmo de Levenberg-Marquardt, con un error final bajo.
3. **KMAX**: indica que el *target* ha sido procesado por Levenberg-Marquardt pero que finaliza por haber agotado el máximo número de iteraciones del algoritmo.
4. **LOW_INC**: hemos visto que el algoritmo de Levenberg-Marquardt desprecia los incrementos muy pequeños. Este estado indica que el *target* ha terminado de procesarse porque los incrementos calculados son despreciables.
5. **NAN_INCS**: uno de los problemas del algoritmo es que cuando no encuentra una solución que mejore el error calculado de forma repetida, el factor de amortiguación se hace demasiado grande y, al calcular los incrementos con $\delta_p = (H + \mu \cdot I)^{-1} \cdot g$, éstos se hacen NAN. Este estado indica que el *target* no ha podido resolverse con normalidad.

Antes de que el algoritmo Levenberg-Marquardt los procese, a cada *target* se le añade la posición inicial del *tip* o efector final que tienen asignado así como los valores angulares iniciales de la cadena cinemática a la que pertenece el efector. Cuando el algoritmo termina, les añade su estado de finalización, el error con que finalizó la ejecución del algoritmo, los ángulos finales calculados, la posición final del efector y el tiempo que estuvieron procesándose.

Además, para los *targets* especiales se han añadido campos como el eje y la distancia que debe avanzar el efector (para los ADVANCEAXIS) o algunas restricciones en los ejes y los ángulos de éstos.

Por otro lado, esta clase completa su funcionalidad con métodos para realizar consultas o actualizar los atributos del *target*.

5.2.3 Las partes del cuerpo del robot

Para poder enviar distintos *targets* a distintas partes del cuerpo del robot hemos desarrolado un mapa donde guardamos tres variables pertenecientes a la clase ***BodyPart***, las cuales definen las siguientes tres partes:

1. El brazo derecho: denominado como **RIGHTARM**. Esta parte guarda la lista de joints que forman la cadena cinemática del brazo derecho del robot, así como su efector final (el que verdaderamente debe situarse sobre la pose de destino). También guarda en una cola los *targets* que se le asignan y almacena una variable de la clase *CinematicaInversa* (preparada para usar su lista de joints y su tip) con la que realiza todos los cálculos para resolver los *targets* que tiene almacenados.
2. El brazo izquierdo: llamado **LEFTARM**. También tiene una lista de joints , formada por los motores del brazo izquierdo del robot, y su correspondiente efector final, con los que trabaja su variable de *CinematicaInversa*.
3. La cabeza: etiquetada como **HEAD**, y al igual que las anteriores tiene una lista con los joints y un tip o efector final situado en la cabeza del robot con los que trabaja su variable de *CinematicaInversa*.

Al igual que la clase *Target, BodyPart* complementa su funcionalidad con los métodos de consulta y actualización de sus atributos, tan comunes en la programación orientada a objetos.

5.2.4 La clase encargada de la cinemática directa

Una vez estudiadas las clases *Target* y *BodyPart*, pasamos a explicar la clase ***CinematicaInversa***. Es sin lugar a dudas la más importante de todas las desarrolladas en este proyecto. Esta clase es la que se encarga de recoger el *target* asignado a una parte del cuerpo del robot y resolverlo, llevando el efector a su posición si es de tipo POSE6D, trasladándolo a través de un eje si es de tipo ADVANCEAXIS o alineándolo si es de tipo ALINGAXIS. Para ello, tiene almacenada la cadena cinemática de la parte del robot a la que está asignada, así como el efector final. Cuenta con una copia local del *InnerModel* que va actualizando para optimizar el tiempo de computación en los cálculos matemáticos.

En esta clase quedan recogidos tanto el método Levenberg-Marquardt como todos aquellos métodos auxiliares de los que se vale el algoritmo estudiado para poder resolver los problemas de cinemática inversa planteados. Estos métodos son de suma importancia por lo que los iremos desgranando uno a uno en el siguiente apartado.

5.3 Profundizando en la cinemática inversa

A continuación explicaremos los métodos matemáticos empleados en la resolución de los problemas de cinemática inversa que hemos visto. Todos los métodos están codificados en la clase ***CinematicaInversa***.

5.3.1 Algoritmo de Levenberg-Marquardt ampliado

Durante todas las fases de este proyecto se utilizará la misma estructura del algoritmo de Levenberg-Marquardt vista en el apartado 4.4.2, a la que se le han añadido varias funcionalidades más. Entre ellas destacan dos:

1. se han añadido unos pesos a las traslaciones y rotaciones, y
2. se ha implementado un control para los casos en los que el incremento, calculado para los valores angulares de un joint, supere los límites mínimo y máximo del motor.

5.3.1.1 Matriz de pesos

Comenzaremos explicando en qué consisten los **pesos** añadidos a las traslaciones y a las rotaciones del vector de error, en el algoritmo de Levenberg-Marquardt.

El algoritmo de Levenberg-Marquardt intenta minimizar una función de error para encontrar la solución óptima del problema de la cinemática inversa. La función de error que utilizamos es $\epsilon = P_{target} - F(\theta)$, donde Ptarget es la pose (un vector de tres traslaciones, tx, ty, tz, y tres rotaciones, rx, ry, rz) de la posición deseada o punto objetivo, F es la función de cinemática directa que nos da la posición del efector final y θ es el vector de ángulos de los joints de la cadena cinemática con la que se trabaja, [θ1, θ2..., θn]. En definitiva, el error es un vector de seis elementos: $\epsilon = [t_x, t_y, t_z, r_x, r_y, r_z]$

Hemos visto que al calcular los incrementos, en el algoritmo de Levenberg-Marquardt, utilizamos la expresión $\delta_p = (H + I \cdot \mu)^{-1} \cdot g$, donde g es el descenso de gradiente, $J^t \cdot \epsilon$. En esta expresión podemos comprobar que los errores de traslación y de rotación, ϵ, influyen a la hora de calcular un nuevo incremento. Ante esta situación se nos presentan dos problemas:

- Las traslaciones están medidas en milímetros mientras que las rotaciones están medidas en radianes. ¿Cuál es la relación entre milímetros y radianes? ¿Puede pesar más en el jacobiano las rotaciones que las traslaciones o viceversa?
- ¿Qué pasa si queremos tener en cuenta sólo los errores de traslación? o ¿qué debemos hacer para considerar sólo los errores de rotación en X y Z y no de Y?

Para estos casos se ha seguido el método propuesto por el profesor de la Universidad de Osaka, Tomomichi Sugihara, en su documento "*Solvability-Unconcerned Inverse Kinematics by the Levenberg-Marquardt Method*", en el que introduce una matriz, We, con el fin de absorber las diferencias métricas entre las traslaciones y las rotaciones, por un lado, y de ponderar la importancia de cada restricción de traslación y de rotación, por el otro.

En este proyecto se ha comprobado[19] que la relación metros/radianes es más efectiva y tiene mejores resultados que la relación milímetros/radianes, unidades de medición utilizadas por el sistema de representación de todas las herramientas de Robocomp.

De esta forma, y con el objetivo de simplificar el método de Levenberg-Marquardt, el componente *inverseKinematicsComp* toma los datos que recibe de *InnerModel* en milímetros y los pasa a metros, simplificando la matriz de pesos, We, a una matriz diagonal binaria, donde el 1 significa que se tiene en cuenta las restricciones sobre las que se aplica y el 0 las anula:

$$W_e \cdot \epsilon = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{0} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} t_x \\ t_y \\ t_z \\ r_x \\ r_y \\ r_z \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \\ t_z \\ \mathbf{0} \\ r_y \\ r_z \end{pmatrix}$$

Equation 19: Matriz de pesos

Así, cuando se calcula el gradiente de descenso,

[19] Como se verá en el siguiente apartado donde se explican las pruebas realizadas sobre el componente *inverseKinematicsComp* y los problemas a los que se ha hecho frente durante su desarrollo

$$g = J^t \cdot (W_e \cdot \epsilon)$$

sólo se tienen en cuenta aquellas restricciones marcadas con pesos 1. De esta forma se puede escoger la configuración que más nos convenga: manteniendo traslaciones y rotaciones, sólo traslaciones, algunas rotaciones...

5.3.1.2 Bloqueo de Motores

Al calcular los incrementos para los ángulos de los joints pertenecientes a una cadena cinemática se nos plantea una pregunta: en el mundo real los joints representan los motores de un robot. Los motores pueden tener unos límites máximos y mínimos que no se pueden traspasar, es decir, los motores tienen un rango dentro del cual pueden girar. Por otra parte, el algoritmo de Levenberg-Marquardt calcula unos incrementos que luego son añadidos a los ángulos originales de los joints:

$$1)\quad \delta_p = (H + \mu \cdot I)^{-1} \cdot g\ ; \quad \text{y } 2)\quad p_{new} = p + \delta_p$$

Entonces, ¿qué ocurre cuando el algoritmo de Levenberg-Marquardt calcula unos incrementos que superan los límites de un motor?

Para afrontar estas situaciones se ha seguido el método propuesto por Baerlocher y Boulic en el documento "*An inverse kinematics architecture enforcing an arbitrary number of strict priority levels*". Este método propone añadir en el algoritmo de Levenberg-Marquardt un bucle interno que se encargue de comporbar si algún ángulo calculado supera los límites del joint que le corresponda y, de darse el caso, bloquear el joint y eliminarlo de los cálculos. Para ello se necesita un vector de bloqueos, L_b , de n elementos (con n = número de joints) donde $L_b[i] = 0$ significa que el joint i está libre y $L_b[i] = 1$ significa que el joint i está bloqueado. Cuando el algoritmo se inicie, los elementos de este vector estarán a cero, indicando que están libres. Después de calcular los nuevos ángulos sólo se evaluarán aquellos joints no bloqueados. Si se detecta una violación en los límites del joint libre j, se lo bloquea con $L_b[j] = 1$ y su valor angular calculado se cambia por el valor del límite del joint superado. Después se recalcula el jacobiano, cuya columna j (la columna asociada al joint bloqueado) se pondrá a cero para evitar nuevas violaciones de límites. Los joints bloqueados no se liberarán hasta

que no se encuentre una solución que no bloquee los motores. Esto limita el número de poses alcanzables del método original.

Para no saturar el algoritmo de Levenberg-Marquardt con otro bucle interno que recorra todos los joints y por cada uno calcule las posibles violaciones en sus límites, se ha añadido una llamada a un método, *outLimits*, encargado de ver si los ángulos a aplicar a los motores exceden sus límites. Su estructura es:

Método: outLimits
Entrada/salida: lista de ángulos nuevos a aplicar sobre los motores, *angles*. Cuando un ángulo supera un límite se modifica poniendo el valor del límite superado.
Lista binaria ordenada de n motores, L_b , donde

- $L_b[i]=0$ significa que el motor i no está bloqueado.
- $L_b[i]=1$ significa que el motor i está bloqueado.

Salida: booleano que indica si se han sobrepasado los límites de algún motor (false) o no (true).
Algoritmo:

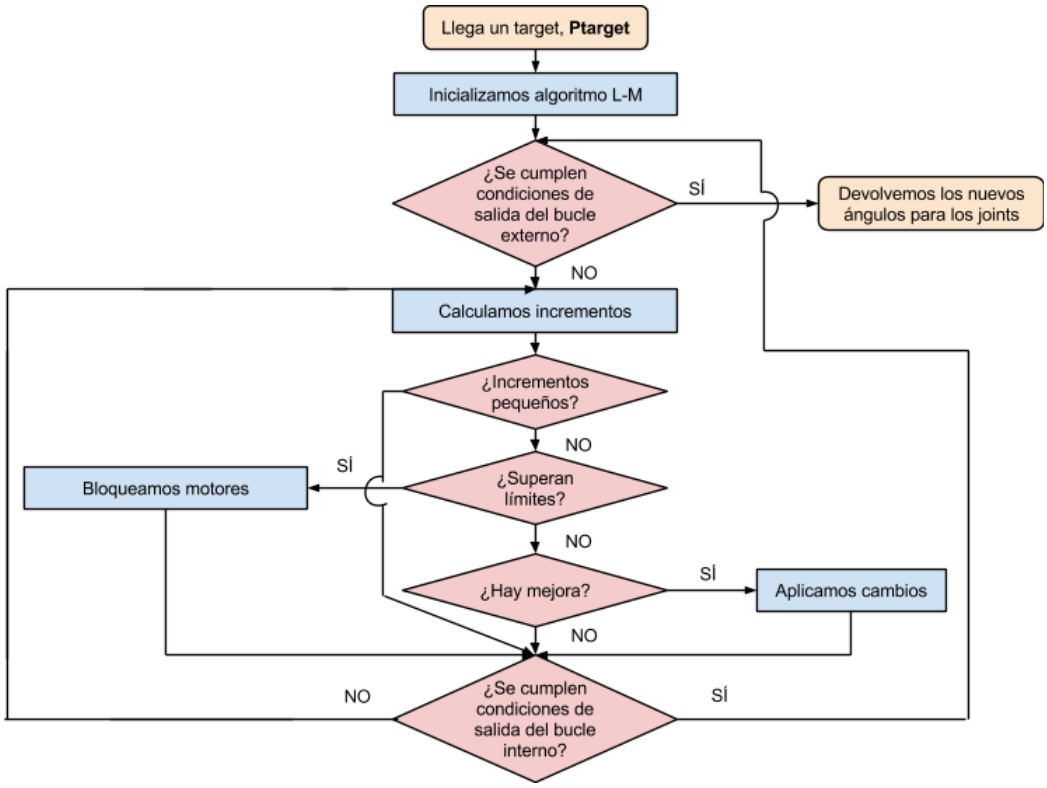
```
/* Inicializamos la bandera y los auxiliares para recoger los
límites de los motores*/
bool noSupera = true;
float limiteMin, limiteMax;
/* Recorremos la lista de joints de la cadena cinemática. Sacamos
los límites mínimo y máximo de cada motor*/
for(int i=0; i<this->listaJoints.size(); i++)
   limiteMin = innerModel->getJoint(listaJoints[i])->min();
   limiteMax = innerModel->getJoint(listaJoints[i])->max();
   /* Comparamos los límites con el nuevo ángulo que se le quiere
   asignar al joint. Si supera los límites, cambia la bandera,
   bloqueamos el motor y cambiamos el ángulo por el límite*/
   if(angles[i]<limiteMin or angles[i]>limiteMax)
     noSupera = true;
      Lb[i]=1;
     if(angles[i] < limiteMin)
       /*Si el ángulo supera el límite mínimo, cambiamos su valor
       por el límite mínimo */
       angles[i] = limiteMin;
     else
       /*Si el ángulo supera el límite máximo, cambiamos su valor
       por el límite máximo */
        angles[i] = limiteMax
     endIf
   endIf
endFor
return noSupera;
```

5.3.1.3 Estructura final del algoritmo de Levenberg-Marquardt

Al añadir estas dos nuevas restricciones al Levenberg-Marquardt, la estructura original del algoritmo vista en el apartado 4.4.2 se amplía, complicando ligeramente su

entendimiento. Con ánimo de simplificarlo, presentaremos esta nueva estructura mediante los siguientes diagramas de flujos.

El primer diagrama de flujo refleja a grandes rasgos la estructura general que tiene nuestro algoritmo completo de Levenberg-Marquardt, con los nuevos controles añadidos:

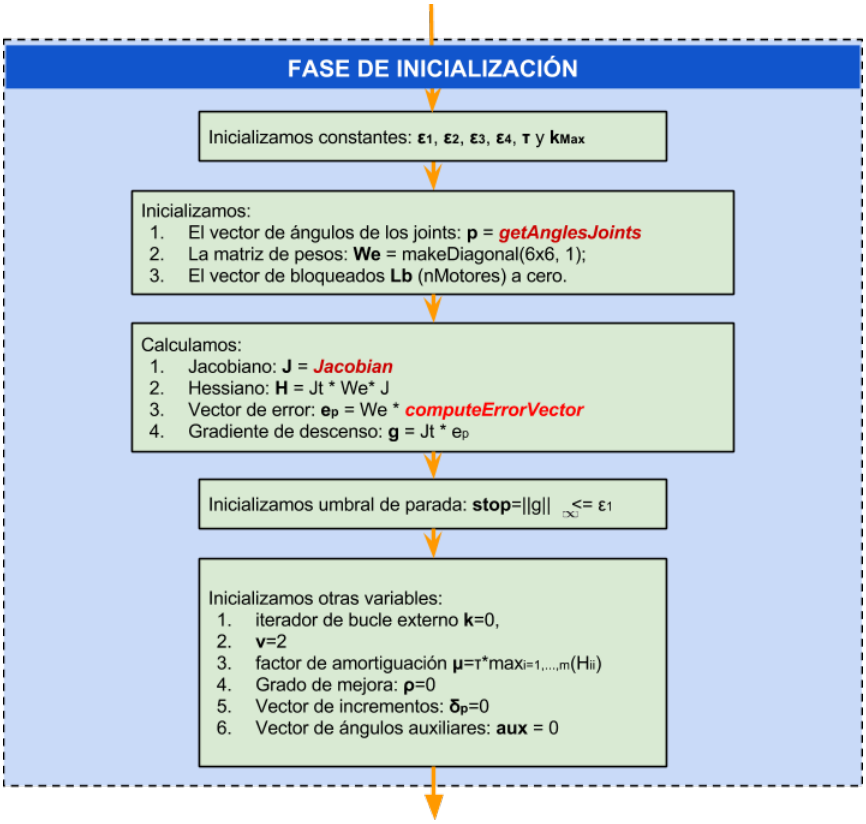


FlowDiagram 1: Estructura general del algoritmo de Levenberg-Marquardt ampliado

En esta estructura de alto nivel distinguimos varias fases:

1) Una **fase de inicialización**: en esta fase inicializamos las constantes del algoritmo, como los distintos ε_i , el vector de valores angulares de los joints, la matriz de pesos que explicamos en el apartado 5.2.1.1, y el vector de motores bloqueados del apartado 5.2.1.2, así como el umbral de parada del algoritmo, el factor de amortiguación y otras variables que se necesitarán en cálculos posteriores, como un vector de ángulos auxiliares para poder deshacer cambios. También se hacen los primeros cálculos del

vector de error entre el *target* y el efector final, el jacobiano, el hessiano y el gradiente de descenso.

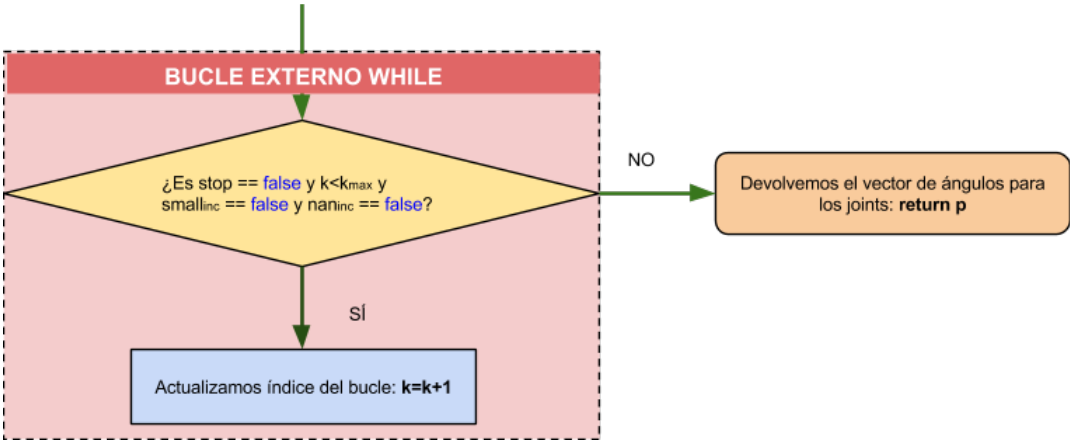


FlowDiagram 2: Fase de inicialización del algoritmo

2) Un **bucle externo**: este bucle detiene la ejecución del algoritmo cuando:

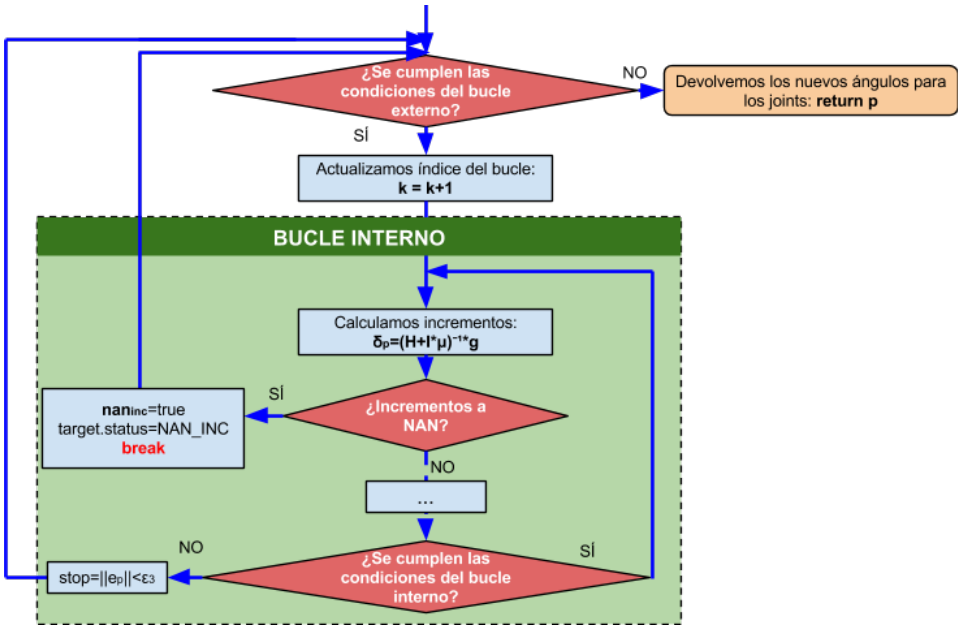
1. Se ha obtenido un error aceptable, en ese caso se sale del bucle y se devuelven los valores angulares calculados para cada joint, indicando que la ejecución del algoritmo ha finalizado correctamente.
2. Se han alcanzado el máximo número de iteraciones. Se devuelve el vector de ángulos calculado, indicando que se ha alcanzado el máximo de iteraciones.
3. Se han obtenido incrementos despreciables: de este modo bajamos la carga de cómputo del algoritmo.
4. Se han obtenido incrementos NAN. Cuando no se encuentran ángulos buenos durante muchas iteraciones seguidas, el parámetro μ crece demasiado y los

incrementos se hacen NAN. Al igual que para los incrementos despreciables, salimos del bucle, ahorrando tiempo de ejecución.



FlowDiagram 3: Condiciones del bucle externo del algoritmo

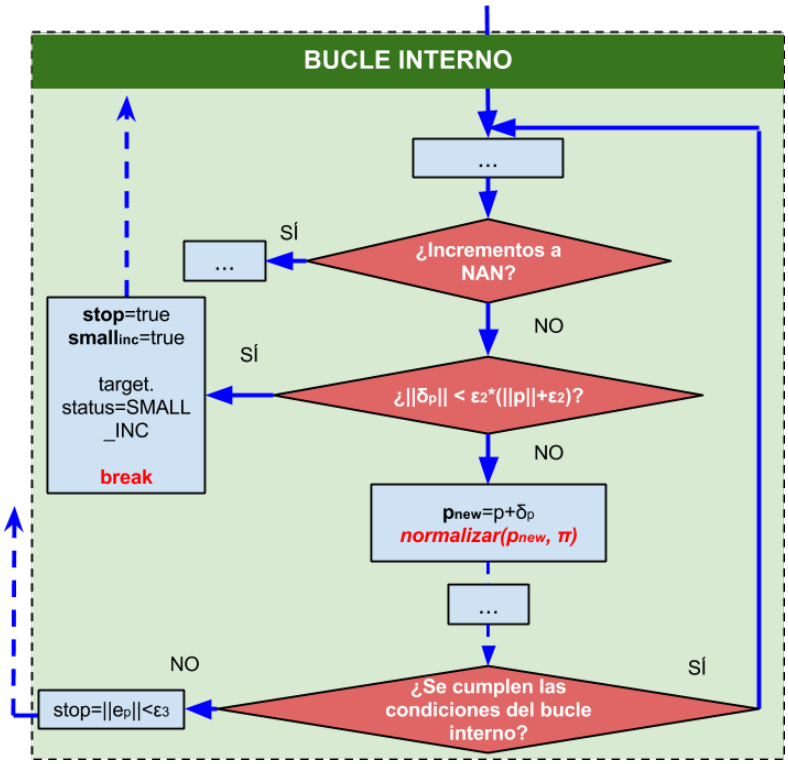
3) Dentro del bucle externo se actualiza el índice de iteraciones, k, y se entra en un **bucle interno do-while**. En este bucle es donde se realizan los cálculos más importantes del algoritmo. Primero se calculan los incrementos δ_p para los valores angulares de los joints y sobre ellos se hacen dos primeras comprobaciones:



FlowDiagram 4: Primera parte del bucle interno

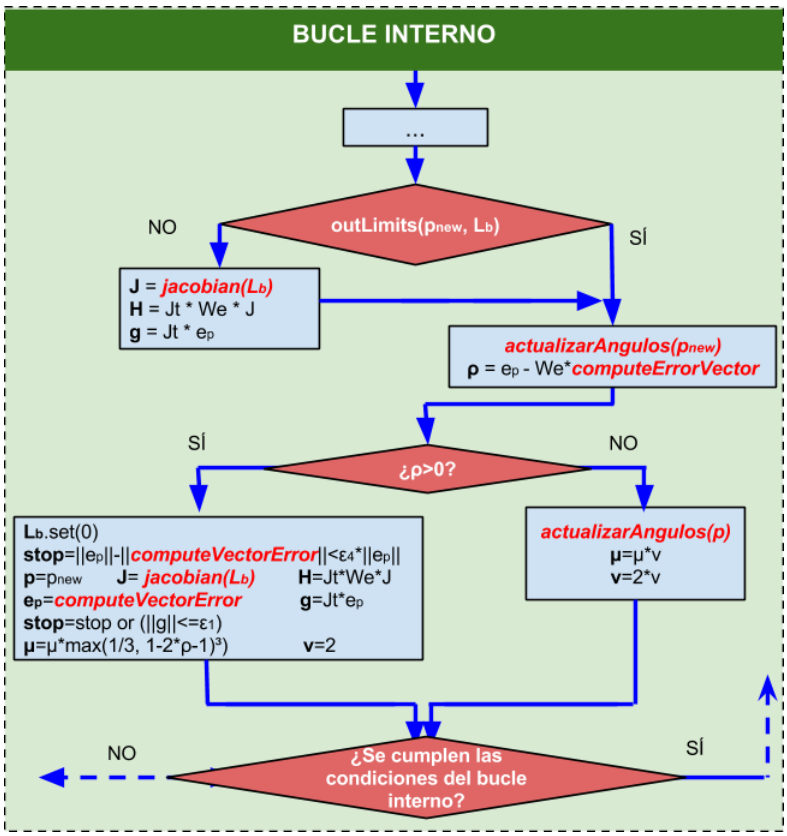
- **Que no sean NAN:** en cuyo caso se levanta la bandera de salida nan_{inc} se pone el *target* con estado NAN_INCS y se sale tanto del bucle interno como del externo.
- **Que no sean demasiado pequeños**: cuando los incrementos se hacen cero o son muy cercanos a cero, no merece la pena aplicarlos al no disponer normalmente de robots con motores precisos y con mucha sensibilidad. Además, cuando se alcanzan incrementos muy pequeños, en las siguientes iteraciones seguirán apareciendo incrementos despreciables. Cuando se llega a esta situación se levantan las banderas *stop* y $small_{inc}$ y se termina la ejecución de los bucles, ahorrando tiempo de cómputo. El *target* guardará el status SMALL_INC.

Una vez aceptados los incrementos, éstos se suman a los ángulos originales de los joints, se normalizan y se almacenan en una variable auxiliar para luego poder deshacer los cambios si el nuevo error obtenido no mejora:



FlowDiagram 5: Segunda parte del bucle interno

Después de normalizarlos, utilizando el método desarrollado en este proyecto para normalizar números reales, *calcularModuloFloat*, se comprueba que los nuevos valores angulares no superen los límites del joint asignado. Para ello aplicamos el método del apartado anterior, 5.3.1.2, *outLimits*, que devuelve **true** si ningún joint ha superado sus límites y **false** en caso contrario. Además se encarga de bloquear los motores mediante la lista de bloqueados L_b y devuelve los nuevos valores angulares cambiando aquellos ángulos que superan las restricciones del joint por el valor del límite superado. Sólo si los límites han sido superados se recalculan las matrices J y H, haciendo cero aquellas columnas del jacobiano cuyos correspondientes joints estén bloqueados, para que, al calcular los siguientes incrementos, el algoritmo no los tenga en cuenta. También se calcula de nuevo el gradiente de descenso g.



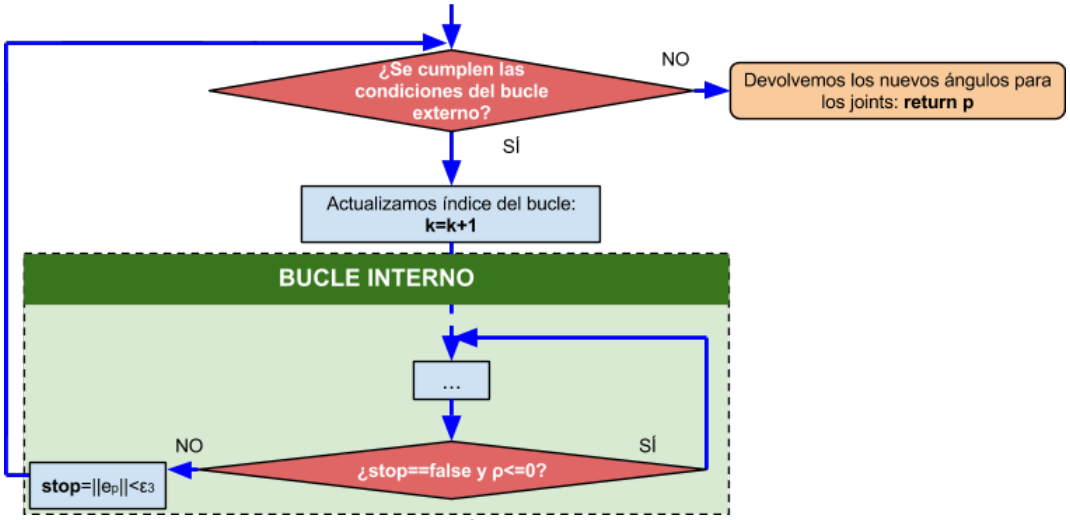
FlowDiagram 6: Tercera parte del bucle interno

Después se aplican los nuevos ángulos a los joints y se calcula el nuevo error alcanzado. Si el nuevo error es menor que el anterior se aceptan los cambios, desbloqueando los

motores (por si la nueva posición permite calcularles unos incrementos razonables), actualizando las matrices J y H, así como el descenso de gradiente g y el factor de amortiguación. También se recalcula el valor de la bandera de salida, *stop*, para que cuando se alcancen unos errores aceptables, se termine el bucle.

Si por el contrario, el error ha empeorado se deshacen los cambios restaurando los ángulos antiguos y se aumenta el parámetro de amortiguación.

Al salir del bucle interno se recalcula la condición de salida *stop* y se vuelve a evaluar la condición del bucle externo:



FlowDiagram 7: Última parte del bucle interno

Como podemos ver en la nueva estructura del método de Levenberg-Marquardt, existen tres métodos fundamentales, encargados de:

- Obtener los valores angulares de los joints o motores, para lo que nos apoyaremos en la herramienta *InnerModel*.
- Crear una función que calcule el vector de error entre el punto *target* y el efector final, el cual deberá tender a nulo (caso ideal).
- Calcular la matriz jacobiana para la configuración de joints que utiliza el método de Levenberg-Marquardt.

Existe un cuarto método que, aunque no es fundamental, es verdaderamente útil. Es el *calcularModuloFloat*, que nos servirá para calcular los ángulos en un determinado rango, evitando por ejemplo ángulos de 9.42 radianes ($3 \cdot \pi$). En los siguientes apartados iremos explicando cómo se han implementado y en qué consisten cada uno de estos cuatro métodos.

5.3.2 Normalización de los ángulos

Normalmente el rango de los ángulos en los que se mueven los motores reales es de $-\pi$ y π radianes o incluso menos. Uno de los problemas a los que nos enfrentamos es que el algoritmo de Levenberg-Marquardt se basa en calcular incrementos que va sumando a los ángulos originales. De este modo es fácil observar que los valores angulares nuevos calculados para los joints normalmente excenden de $-\pi$ y π radianes. Para evitar esto se ha codificado el método *calcularModuloFloat*. Inicialmente pensado para calcular el módulo entre dos números reales ha terminado siendo el método que normaliza el vector de ángulos nuevos entre $-\pi$ y π. Su estructura es la siguiente:

```
Método: calcularModuloFloat
Entrada: vector de número que se quiere normalizar, angles, y número
real con el que se quiere normalizar, mod. Generalmente será π
Entrada/salida: vector de valores angulares a normalizar, angles.
Algoritmo:
    /* Recorremos todo el vector de ángulos para calcular su módulo
    entre mod=π */
    for(int i=0; i<angles.size(); i++)
       int cociente = (int)(angles[i]/mod);
       angles[i] = angles[i]-(cociente*mod);

       /* Normalizamos entre -π y π */
       if(angles[i] > π)
          /* Si es mayor que π, le restamos π*/
          angles[i]=angles[i]-π;
       else
          if(angles[i] < -π)
             /* Si es menor que -π, le sumamos π*/
            angles[i]=angles[i]+π;
          endIf
       endIF
    endFor
```

5.3.3 Cálculo de los ángulos de los Joints

Necesitamos dos métodos relacionados con los ángulos de los joints de la cadena cinemática, uno para obtener los valores de éstos y otro para actualizarlos. Para ello se han codificado dos métodos, *calcularAngulos* y *actualizarAngulos*, respectivamente.

El método *calcularAngulos* toma la lista de joints de la cadena y, con ayuda de *InnerModel*, obtiene sus valores angulares y los guarda en un vector que devuelve al método invocador:

```
Método: calcularAngulos
Salida: vector de valores angulares de los joints de la cadena
cinemática, angulos
Algoritmo:
    QVec angulos; /* Inicializamos vector de ángulos a cero*/
    /* Recorremos la lista de joints. Sacamos el ángulo y lo guardamos*/
    for(int i=0; i<this->listaJoints.size(); i++)
        float angle = innerModel->getJoint(listaJoints[i])->getAngle();
        angulos.push_back(angle);
    endfor
    return angulos;
```

El método *actualizarAngulos* recibe como parámetro de entrada el vector de los nuevos valores angulares para los joints de la cadena cinemática.

```
Método: calcularAngulos
Entrada: vector de valores angulares nuevos para los joints, angles_new
Algoritmo:
    /* Recorremos la lista y actualizamos con ayuda del InnerModel*/
    for(int i=0; i<this->listaJoints.size(); i++)
        innerModel->updateJointValue(listaJoints[i], angles_new[i] );
    endfor
```

5.3.4 Cálculo del error

Hemos visto que para aplicar el algoritmo de Levenberg-Marquardt necesitamos una función de error que debemos minimizar. Esta función se encarga de calcular el vector que une el nodo efector de la cadena cinemática del robot con el *target*:

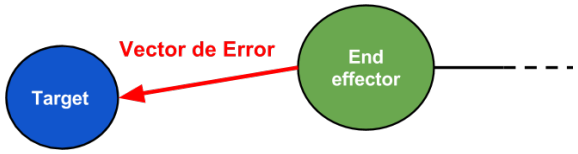


Illustration 28: Vector de error entre el efector y el target

Esto se expresa como el vector pose del *target* menos el vector pose del efector final, calculado mediante la función de cinemática directa F(θ), donde θ es el vector de ángulos [θ1, θ2..., θn] que forman los n joints de la cadena cinemática del robot:

$$\epsilon_p = P_{target} - F(\theta) \quad .$$

Equation 20: Función de error

Para este fin se ha implementado el método *computeErrorVector*, que se encarga de calcular el error mediante la diferencia entre las coordenadas de traslación y rotación del punto objetivo, y las coordenadas de traslación y rotación del efector, vistos ambos desde el **último joint** de la cadena cinemática. ¿Por qué calculamos los errores desde el sistema de referencia del último joint? La razón es sencilla: el algoritmo de Levenberg-Marquardt sólo tiene en cuenta la cadena formada por los **joints** para hacer sus cálculos. Al comportarse de esta forma, el algoritmo parte de la base de que el último joint siempre coincide con el efector final. Pero ¿qué ocurre si el efector final no coincide con el último joint? Normalmente el efector final no es una articulación y además puede estar trasladado y/o rotado con respecto al último joint de la cadena. Esto supone un problema en el cálculo del error:

- Si calculamos el error entre el *target* y el efector, sin contar con el último joint, el algoritmo moverá la cadena hasta ajustar el joint con el error calculado:

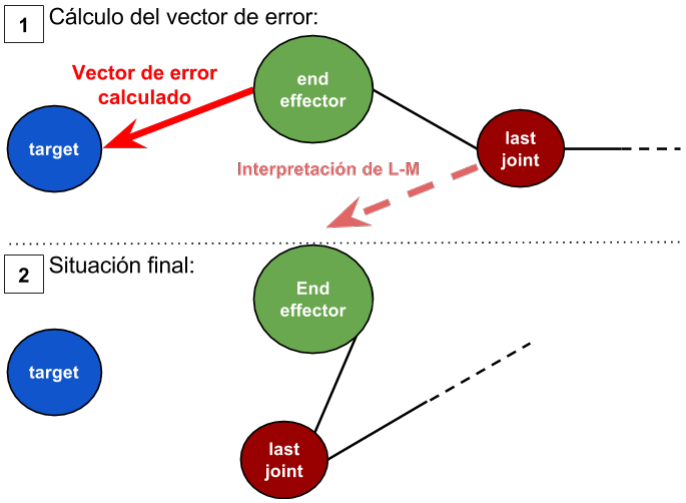


Illustration 29: Problemas con el cálculo del vector de error (I)

- Por el contrario, si calculamos el error entre el último joint y el *target,* sin tener en cuenta el efector final, conseguimos colocar el último joint en la posición deseada, pero el efector final, nuestro objetivo, se queda descolocado:

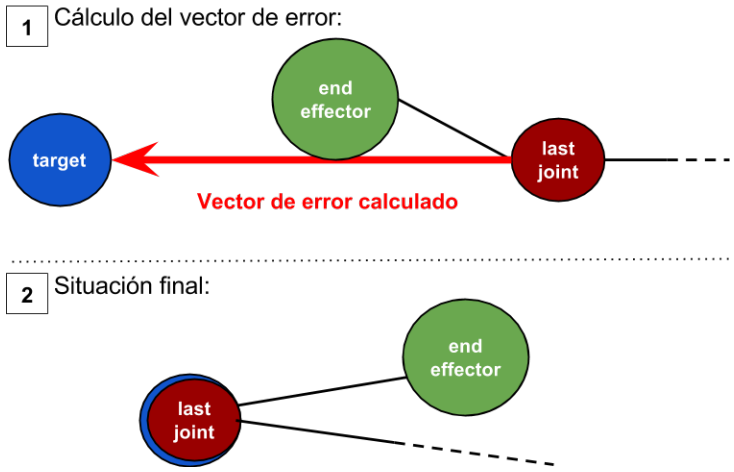


Illustration 30: Problemas con el cálculo del vector de error (II)

Por estas razones necesitamos un método que nos calcule el vector error entre el *target* y el efector final trasladado al último joint. Con el objeto de hacerlo más entendible al lector, explicaremos en qué se basa este método mediante un ejemplo.

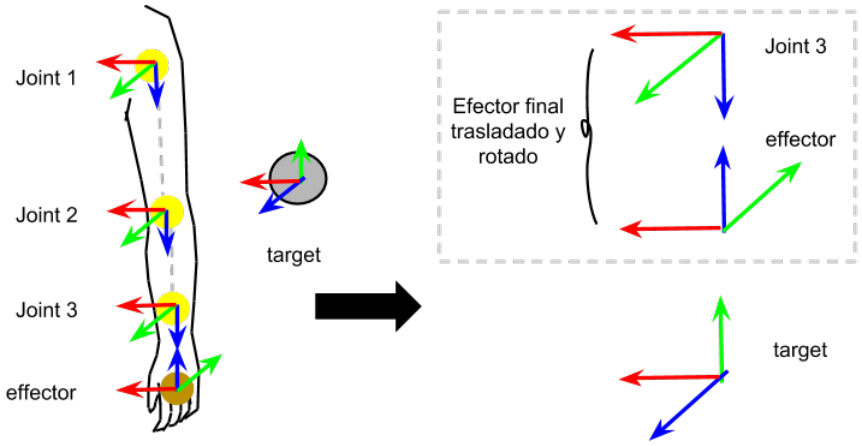


Illustration 31: Ejemplo de efector y joint distintos

Supongamos que tenemos la situación de la figura 31. En ella podemos ver que el nodo efector de la cadena cinemática está trasladado unas unidades en el eje Z y rotado π radianes en el eje X, con respecto al último joint. A partir de aquí se nos plantean dos

preguntas que resolveremos por separado: 1) ¿cómo trasladamos el error de traslaciones al último joint? y 2) ¿cómo trasladamos el error de rotación al último joint?

5.2.5.1 Error de traslación

Calcular el error de traslación entre el efector final y el *target* resulta muy sencillo gracias a la herramienta *InnerModel*. Para ello sólo hay que trasladar las coordenadas de posición del *target* y del efector final, al sistema de referencia del último joint y restarlas:

$$Target_{inJoint} = innerModel.transform(listaJoints.last(), target, world);$$

$$Eff_{inJoint} = innerModel.transform(listaJoints.last(), [0,0,0], endEffector);$$

$$\epsilon_{translations} = Target_{inJoint} - Eff_{inJoint};$$

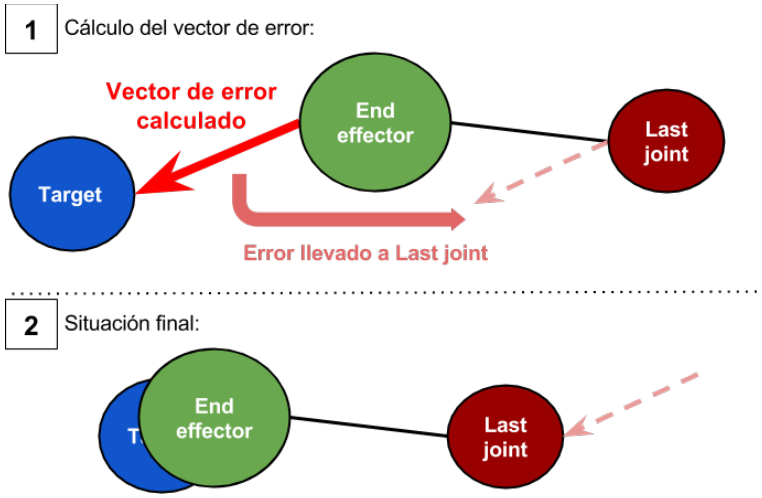


Illustration 32: Error de traslación visto en el joint

5.2.5.2 Error de rotación

En comparación con el cálculo del error de traslaciones, obtener los ángulos de rotación necesarios para que el efector se oriente como el *target* es una tarea bastante más difícil. Si usamos el método implementado durante el proyecto y añadido a *innerModel*, *extractAnglesR_min*, para el sistema mostrado en la anterior figura 31 obtendríamos que:

1. Para que el sistema de referencia del efector final alcance la orientación del *target* debe girar π/2 radianes en X.

2. Para que el sistema de referencia del último joint se oriente como el sistema de referencia del *target* debe girar - π/2 radianes en X.
3. Para que el sistema de referencia del último joint alcance la orientación del efector final debe girar π radianes en X.

Sin embargo, ninguno de estos ángulos nos sirve. Lo que estamos buscando son aquellos ángulos que debe rotar el último joint para que el efector final se oriente como el *target*. Representado de forma gráfica resulta fácil calcularlos: el último joint debe rotar π/2 radianes en el eje X para que el efector final quede orientado como el *target*.

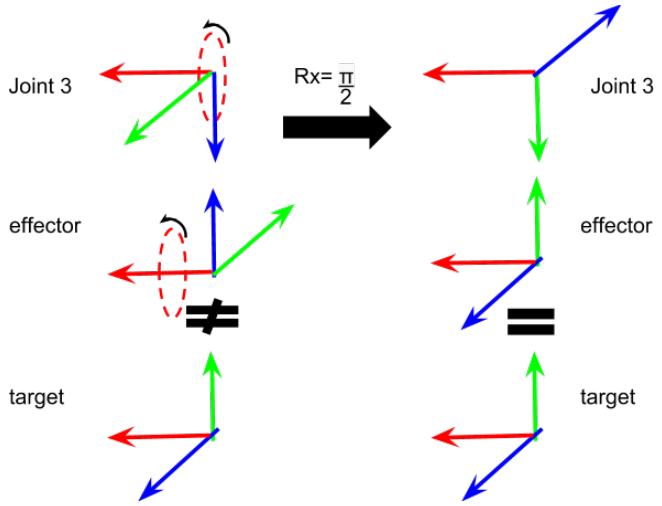


Illustration 33: Rotaciones del joint para orientar el efector

Para obtener ese π/2 en X debemos hacer una serie de cálculos ayudados por el método *extractAnglesR_min*:

1. Primero calculamos la matriz de rotación del *target* al efector final, M_{TE} . De ella extraemos los ángulos que debe girar el efector final para alcanzar la orientación del *target,* y los guardaremos en un vector, $angles_{ET}$. Ese es el error de rotación del efector y es lo que debemos trasladar al sistema de referencia del último joint.
2. Después, calculamos la matriz de rotación del efector al último joint, M_{EJ} . Ésta matriz es la que devuelve el error de rotación entre el último joint y el

efector final. Con ella iremos obteniendo los ángulos que debe girar el último joint para que el efector se oriente como el *target*.

- Obtenemos la primera matriz de rotación, $first_{rot}$, como el producto del error de rotación en X entre el efector final y el *target,* $angles_{ET}[0]$ por la matriz M_{EJ}: $first_{rot} = Rot3D(M_{EJ} \cdot (angles_{ET}[0],0,0))$. Es el cálculo de la matriz de rotación en X, trasladando el error de rotación en X del efector final al joint.
- Obtenemos la segunda matriz de rotación, $second_{rot}$, como el producto del error de rotación en Y entre el efector y el *target,* $angles_{ET}[1]$ por la matriz M_{EJ}: $second_{rot} = Rot3D(M_{EJ} \cdot (0, angles_{ET}[1], 0))$. Estamos calculando la matriz de rotación en Y, trasladando el error de rotación en Y del efector final al joint.
- Obtenemos la tercera matriz de rotación, $third_{rot}$, como el producto del error de rotación en Z entre el efector y el *target,* $angles_{ET}[2]$ por la matriz M_{EJ}: $third_{rot} = Rot3D(M_{EJ} \cdot (0,0, angles_{ET}[2]))$. Es el cálculo de la matriz de rotación en Z, trasladando el error de rotación en Z del efector final al joint.

3. Una vez transformadas las rotaciones del sistema de referencia del efector final al último joint, creamos la matriz de rotación completa para éste último, calculada como el producto de las matrices de rotación en X, Y y Z anteriores: $M_{final} = (first_{rot} \cdot second_{rot}) \cdot third_{rot}$.
4. Al fin, con esta última matriz obtenemos los ángulos que debe girar el último joint para que el efector final quede orientado como el *target*: $angles_{final} = M_{final}.extractAnglesRmin()$

Uniendo los cálculos del error de traslación y del error de rotación obtenemos la estructura final del método *computeErrorVector*:

Método: computeError
Entrada: El *target* a resolver por la cinemática inversa, P_{target}
Salida: vector de error entre el *target* y el efector final, ϵ_p

ALgoritmo:

```
/* ERROR TRASLACIONES */
/*Calculamos las coordenadas del punto objetivo en el sistema de referencia del último joint. El transform nos permite "transformar" las coordenadas de un sistema de referencia (en este caso el mundo "world" a otro (el del último joint)*/
QVec Target_Joint=innerModel.transform(listaJoints.last,[0,0,0],P_target.name);
/* Calculamos las coordenadas del efector final en el sistema de referencia del último join*/
QVec Effector_Joint=innerModel.transform(listaJoints.last,[0,0,0],endEffector);
/* Restamos las coordenadas obtenidas */
QVec ϵ_translations=Target_inJoint−Effector_inJoint;

/* ERROR ROTACIONES*/
/* Obtenemos la matriz de rotación del target al efector final y el vector de error entre el efector final y el target*/
QMat M_TE=innerModel.getRotationMatrixTo(effector,P_target);
QVec angles_ET=M_TE.extracTAnglesRmin();

/* Obtenemos la matriz de rotación del efector final al último joint y sacamos las matrices de rotación trasladadas al joint. Para ello necesitamos pasar primero los ángulos de rotación de un sistema a otro:*/
QMat M_EJ=innerModel.getRotationMatrixTo(listaJoints.last,effector);
QVec aux_first=M_EJ·(angles_ET[0],0,0);
Rot3D first_rot(aux_first[0],aux_first[1],aux_first[2])
QVec aux_secod=M_EJ·(0,angles_ET[1],0);
Rot3D second_rot(aux_second[0],aux_second[1],aux_second[2]);
QVec aux_third=M_EJ·(0,0,angles_ET[2]);
Rot3D third_rot=M_EJ(aux_third[0],aux_third[1],aux_third[2]);
/* Calculamos la matriz de rotación final del joint y extraemos los ángulos que debe girar para orientar el efector como el target*/
QMat M_final=(first_rot·second_rot)·third_rot
QVec angles_final=M_final.extractAnglesRmin()

/* ERROR TOTAL*/
/*Componemos el error final uniendo el error de traslaciones y el error de rotaciones 8Siempre va primero el error de traslaciones y después el de rotaciones para que coincida con el orden de los elementos de una pose)*/
QMVec ϵ_p=[ϵ_traslations,angulos_final]
return ϵ_p
```

Sin embargo, este método es válido solamente para calcular el error cuando el *target* que se procesa es de tipo POSE6D o ADVANCEAXIS. Para los *targets* de tipo ALINGAXIS necesitamos lamentablemente otro tipo de cálculos.

5.2.5.3 Cálculo del error con targets del tipo ALINGAXIS

Esta funcionalidad está todavía en pruebas y, por ahora, para calcular el error entre el efector final y el *target* de tipo ALINGAXIS no contemplamos la posibilidad de que el efector final y el último joint sean diferentes.

Entonces, partiendo de la premisa de que el efector final y el último joint son el mismo nodo, el objetivo es orientar el efector final con la misma orientación que el *target*. Para ello se utiliza la **fórmula de Rodrigues** aplicada a rotaciones, que ofrece un método computacionalmente eficiente para el cálculo de la matriz de rotación R en el SO(3)[20].

$$v_{rot} = v \cdot \cos(\alpha) + (k \times v) \cdot \sin(\alpha) + k \cdot (k \cdot v) \cdot (1 - \cos(\alpha))$$

La fórmula para rotaciones de Rodrigues está pensada para girar un vector, **v**, **α** radianes alrededor del eje indicado por el vector unitario **k**, descomponiendo el vector en sus proyecciones paralelas y perpendiculares al eje de rotación. Para rotar el vector **v** toma la proyección perpendicular al eje de rotación y la gira, dejando fija la proyección paralela al eje de rotación:

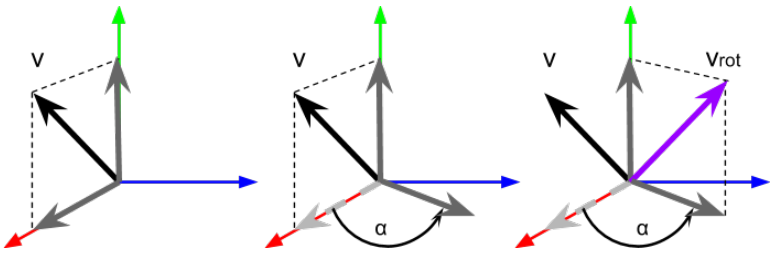


Illustration 34: Método de Rodrigues

Si expresamos la fórmula anterior en notación matricial vemos que:

1. El producto $v \cdot \cos(\alpha)$ para poder representarlo como un producto matriz por vector se transforma en $I \cdot \cos(\alpha) \cdot v$.
2. El producto vectorial $k \times v$ se transforma en el producto de la matriz inversa formada por los elementos del vector unitario k por el vector v, de tal forma que lo que antes era $(k \times v) \cdot \sin(\alpha)$ pasa a ser $[k]_x \cdot \sin(\alpha) \cdot v$.

[20] SO(3) o grupo de rotaciones 3D se refiere a todas las rotaciones sobre el espacio euclídeo tridimensional (X, Y, Z) bajo la función de composición $x \rightarrow f(x)$

3. El producto $k\cdot(k\cdot v)\cdot(1-\cos(\alpha))$ se transforma en $(1-\cos(\alpha))\cdot(k\cdot k^t)\cdot v$, donde k^t es el vector unitario k transpuesto.

De esta forma nos quedaría la siguiente expresión:

$$v_{rot}=I\cdot\cos(\alpha)\cdot\boldsymbol{v}+[k]_x\cdot\sin(\alpha)\cdot\boldsymbol{v}+(1-\cos(\alpha))\cdot([k]_x\cdot[k]_x)\cdot\boldsymbol{v}$$

De esta expresión podemos factorizar el vector v, de tal forma que obtenemos:

$$v_{rot}=(I\cdot\cos(\alpha)+[k]_x\cdot\sin(\alpha)+(1-\cos(\alpha))\cdot([k]_x\cdot[k]_x))\cdot\boldsymbol{v}$$

Si cambiamos $k\cdot k^t$ por el equivalente $[k]_x^2+I$ reducimos la expresión a:

$$v_{rot}=(I\cdot\cos(\alpha)+[k]_x\cdot\sin(\alpha)+(1-\cos(\alpha))\cdot([k]_x^2+I))\cdot\boldsymbol{v}$$

$$v_{rot}=(I\cdot\cos(\alpha)+[k]_x\cdot\sin(\alpha)+[k]_x^2+I-[k]_x^2\cdot\cos(\alpha)-I\cdot\cos(\alpha))\cdot\boldsymbol{v}$$

$$v_{rot}=(I+\cancel{I\cdot\cos(\alpha)}-\cancel{I\cdot\cos(\alpha)}+[k]_x\cdot\sin(\alpha)+[k]_x^2-[k]_x^2\cdot\cos(\alpha))\cdot\boldsymbol{v}$$

$$\boldsymbol{v_{rot}=(I+[k]_x\cdot\sin(\alpha)+[k]_x^2\cdot(1-\cos(\alpha)))\cdot v}$$

Si observamos bien, esta última forma de expresar la fórmula de Rodrigues es lo mismo que $v_{rot}=\boldsymbol{R}\cdot\boldsymbol{v}$, siendo R la matriz de rotación 3x3 para rotar el vector v y dejarlo como vrot. Por lo tanto, de la fórmula de Rodrigues podemos calcular la matriz de rotación R mediante:

$$R=I+[k]_x\cdot\sin(\alpha)+[k]_x^2\cdot(1-\cos(\alpha))\quad .$$

Equation 21: Matriz de rotación con Rodrigues

Donde I es la matriz identidad 3x3, α es el ángulo de rotación en radianes y $[k]_x$ es la matriz antisimétrica obtenida al representar el producto vectorial $k\times v$ como un producto matriz por vector columna $[k_x]\cdot v$, cuyos elementos son:

$$\text{Mano derecha}\rightarrow[\mathrm{k}]_x=\begin{pmatrix}0&-\mathrm{k}_3&\mathrm{k}_2\\ \mathrm{k}_3&0&-\mathrm{k}_1\\ -\mathrm{k}_2&\mathrm{k}_1&0\end{pmatrix}$$

Equation 22: Matriz antisimétrica (derecha)

$$\text{Mano izquierda}\rightarrow[\mathrm{k}]_x=\begin{pmatrix}0&\mathrm{k}_3&-\mathrm{k}_2\\ -\mathrm{k}_3&0&\mathrm{k}_1\\ \mathrm{k}_2&-\mathrm{k}_1&0\end{pmatrix}$$

Equation 23: Matriz antisimétrica (izquierda)

Como resultado se obtiene una matriz de rotación 3x3, $R=R_x\cdot R_y\cdot R_z$, donde Rx, Ry y Rz son las matrices de rotación en los respectivos ejes:

$$R_x=\begin{pmatrix}1 & 0 & 0\\ 0 & \cos\alpha & -\sin\alpha\\ 0 & \sin\alpha & \cos\alpha\end{pmatrix} \quad R_y=\begin{pmatrix}\cos\alpha & 0 & \sin\alpha\\ 0 & 1 & 0\\ -\sin\alpha & 0 & \cos\alpha\end{pmatrix} \quad R_z=\begin{pmatrix}\cos\alpha & -\sin\alpha & 0\\ \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 1\end{pmatrix}$$

Esto significa que primero se aplican las rotaciones en X, después las rotaciones en Y y por último las rotaciones en Z. De la matriz R obtenemos el error de rotación entre el efector final y el *target*.

Al añadir este último procedimiento para el cálculo del error de rotación entre el efector final y el *target* de tipo ALINGAXIS a nuestro método *computeErrorVector*, éste se estructura definitivamente como:

Método: computeError
Entrada: El *target* a resolver por la cinemática inversa, P_{target}
Salida: vector de error entre el *target* y el efector final, ϵ_p
ALgoritmo:

```
/* Dividimos el problema dependiendo de si el target es de tipo
POSE6D-ADVANCEAXIS o ALINGAXIS*/
if( P_target.type ==POSE6D or P_target.type ==ADVANCEAXIS)
  /* Aplicamos método anteriormente explicado */
else
  /*Obtenemos las coordenadas del target en el efector final
  normalizadas, así como los ejes del target*/
  QVec Tar_Eff=innerModel.transform(endEffector,[0,0,0],P_target.name);
  QVec Tar_axis=P_target.getAxis();
  /*Calculamos los ejes en los que queremos rotar*/
  QVec axis_rot=Tar_axis·Tar_Eff;
  /*Calculamos el seno y el coseno del ángulo que se quiere rotar*/
  float cos=(float)(Tar_axis·Tar_Eff);
  float sin=||axis_rot||_2;
  /*Calculamos el ángulo de rotación como la tan⁻¹(sin/cos)*/
  float angle=atan2(sin,cos);
  /*Obtenemos la matriz antisimétrica*/
  QMat k=axis_rot.crossProductMatrix();
  /*Aplicamos la fórmula de Rodrigues*/
  QMat R=QMat::identity(3)+(k·sin(angle))+(k·k)·(1-cos(angle));
  /*Extraemos los errores de rotación y componemos el error total de
  traslaciones (a cero) y rotaciones*/
  QVec error_rot=R.extractAnglesRmin();
  QVec ε_p=[[0,0,0],error_rot];
endif
return ε_p
```

5.3.5 Cálculo del Jacobiano

Para calcular la matriz jacobiana, que utilizará el algoritmo de Levenberg-Marquardt, se ha seguido uno de los métodos explicados por los profesores del departamento de ingeniería eléctrica de la Universidad estatal de Ohio, David E. Orin y William W. Schrader, en el documento "*Efficient Computation of the Jacobian for Robot Manipulators*". Este método se encarga de crear un jacobiano para cualquier cadena cinemática con n-grados de libertad.

La forma final de la matriz jacobiana se caracteriza por dos elementos importantes:

1. Los componentes de la matriz, las derivadas parciales, deben estar expresados en cualquier sistema de referencia perteneciente a la cadena cinemática. En nuestro caso siempre los expresaremos en el sistema de referencia del último joint de la cadena, coincidiendo con la forma de calcular el error.
2. El punto de referencia en el efector final (ya sea real como la punta de un dedo, o ficticia como un tip un poco más alejado de la mano) puede ser elegido arbitrariamente.

En la matriz jacobiana, las filas representan las coordenadas de traslación y de rotación y las columnas representan los joints de la cadena:

$$J=\begin{pmatrix} t_1^x & t_2^x & t_3^x & \dots & t_n^x \\ t_1^y & t_2^y & t_3^y & \dots & t_n^y \\ t_1^z & t_2^z & t_3^z & \dots & t_n^z \\ r_1^x & r_2^x & r_3^x & \dots & r_n^x \\ r_1^y & r_2^y & r_3^y & \dots & r_n^y \\ r_1^z & r_2^z & r_3^z & \dots & r_n^z \end{pmatrix}$$

Por cada joint se calculan los elementos de traslación y de rotación del siguiente modo:

1. Para joints no prismáticos, aquellos cuyos movimientos implican una rotación o una rotación más traslación, 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} . Por otra parte, los elementos de rotación del joint i, se calcula como el producto vectorial $r_i = -(z_{i-1} \times t_{i-1})$.

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

Por otra parte, este algoritmo contempla el caso en que los joints han superado sus límites de acción, es decir, sus límites físicos. Para estos casos, la columna de la matriz jacobiana asociada al joint bloqueado se hace cero.

De esta forma el algoritmo que calcula el jacobiano tendrá la siguiente estructura:

Entrada: lista binaria de joints donde un 0 recoge que el joint no está bloqueado y el 1 indica que el joint está bloqueado, L_b .

Salida: la matriz jacobiana J, de 6 filas (traslaciones y rotaciones) por tantas columnas como articulaciones tenga el robot.

Algoritmo:

```
QMat J(6, listaJoints.size()); /*matriz 6xn*/
int i=0; /*índice que recorre la lista de joints */
/*Sacamos el nombre de cada joint dentro de la lista de joints*/
foreach(linkName, listaJoints)
  /* Si el motor no está bloqueado, hacemos los cálculos*/
  if( Lb[i] ==0)
    /*calculamos vector de ejes unitarios: ej. [1,0,0]*/
    Qvec unitaryAxis=innerModel.getJoint(linkName).unitaryAxis();
    /*Lo pasamos al sistema de referencia del último joint*/
    unitaryAxis=innerModel.transform(listaJoints.last, unitaryAxis,
    linkName);
    /*Coordenadas del joint actual vistas desde el último joint*/
    Qvec coorJoint=innerModel.transform(listaJoints.last,
    [0,0,0],linkName);
    Qvec axis= coorJoint - unitaryAxis;
    Qvec toEffector = coorJoint-innerModel(listaJoints.last, [0,0,0],
    endEffector);
    Qvec res= toEffector.crossProduct(axis); /* MANO IZQUIERDA */

    /* Distintos casos dependiendo de si es prismático o no */
    if(listaJoints[i].isPrismaticJoint()==false)
      /*Montamos los elemntos de traslación */
      J[0,i]= res.x;   J[1,i]= res.y;   J[2,i]= res.z;
      /* Montamos los elementos de rotación*/
      J[3,i]= axis.x;  J[4,i]= axis.y;  J[5,i]= axis.z;
```

```
      else
         J[0,i]= 0;        J[1,i]= 0;        J[2,i]= 0;
         /* Montamos los elementos de rotación*/
         J[3,i]= res.x;    J[4,i]= res.y;    J[5,i]= res.z;
      endif
    endif
  endForeach
  return J
```

5.4 Pruebas realizadas con *inverseKinematicsComp*

El componente *inverseKinematicsComp* ha pasado por varias y tortuosas fases, antes de llegar ser lo que hoy en día es. Este apartado está dedicado a esa evolución del componente, de la que haremos un breve repaso a través de las pruebas que se fueron realizando en su momento.

5.4.1 Primera fase: trabajando en 2D sin rotaciones

En esta primera versión del *inverseKinematicsComp* no hay interfaces de comunicación. Ningún componente le envía los *targets* al *inverseKinematicsComp* si no que es el propio componente el que se los crea internamente mediante una clase auxiliar, el *Generador*. Esta clase trabaja en un sistema 2D y crea los *targets* de forma muy simple: como vectores de dos componentes, la coordenada de traslación en X y la coordenada de traslación en Y (P_{target}=[Px, Py]).

De esta forma, la primera versión del *inverseKinematicsComp* sólo trabaja con puntos 2D sin rotaciones, por lo que tanto el cálculo del jacobiano como el método del error se simplifican al considerar sólo la parte de traslaciones.

Para probar el componente se utilizó el fichero *pruebaIK.xml*, donde se describe un brazo robótico muy parecido al de la figura 4. Este brazo está compuesto por:

- Dos segmentos:
  - **L1**: mide 400mm de longitud e inicialmente forma un ángulo α=0 radianes con el eje X (en rojo).
  - **L2**: mide 300mm de longitud y forma, en un primer momento, un ángulo β=0 radianes con el primer segmento.

- Dos articulaciones, sin límites en su movimiento:
  - **M1**: es el primer joint del brazo robótico. Está anclado al punto (0, 0) del sistema de referencia y se une al siguiente nodo por L1.
  - **M2**: es el segundo joint del sistema. Su posición depende del ángulo α que forme el link L1 con el eje X. Inicialmente se coloca en el punto (400, 0) del sistema.
- **endEffector**: es el extremo de la cadena cinemática, no es un joint. Representa la mano que queremos mover de un punto inicial a un punto destino. Su posición final dependerá de los ángulos α y β que forman los links L1 y L2. Inicialmente se coloca en el punto (700, 0) del sistema.
- **Ptarget**: Además de definir la cadena cinemática que compone el brazo, el fichero *pruebaIK.xml* define este nodo como el punto destino al que queremos mover el endEffector. En principio se coloca en el punto (650, 100) del sistema, pero su posición dependerá de la pose que el *Generador* calcule.

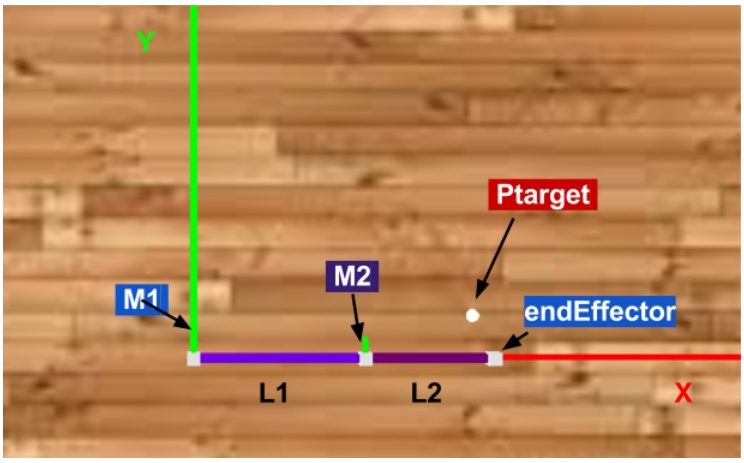


Illustration 35: Brazo robótico de pruebaIK.xml

Así, el *SpecificWorker* crea, en su constructor, una lista con los nombres de los joints ordenados desde el origen hasta el final de la cadena (M1 y M2) y guarda el nombre del tip o efector final (endEffector). Con estos datos inicializa la variable de la clase *CinematicaInversa* y después llama a la clase *Generador*, que le devuelve una cola de

target*s* 2D, que es procesada en orden en el método principal del componenete, enviando los *targets* a la variable de *CinematicaInversa*, para que ésta los resuelva.

Con este sistema tan sencillo, simulado con anterioridad en MATLAB, se probó el método de Levenberg-Marquardt con la estructura original que presentaban Lourakis y Argyros, sin tener en cuenta el bloqueo de motores ni los distintos controles sobre los incrementos calculados. Las pruebas realizadas durante esta fase se pueden clasificar en tres tipos:

1. *Colocar puntos objetivos de forma aleatoria dentro del rango de alcance del brazo*. En este tipo de prueba se colocó el *target* en cien puntos aleatorios dentro de la franja de alcance del efector final. Esta franja está delimitada por dos circunferencias concéntricas, con radios R=700mm. la exterior (la longitud total del brazo estirado 400+300=700) y r=100mm. la interior (400-300=100).
2. *Colocar los puntos objetivos alrededor del efector final, formando una circunferencia de radio 100mm y centro en el efector*. En este tipo de prueba se colocó el efector final en diez posiciones distintas, calculando unos ángulos α y β aleatorios. Para cada una de esas posiciones se situaron diez *targets* alrededor del efector, formando una circunferencia de 100mm de radio.
3. *Colocar puntos objetivos aleatorios cerca del efector final del brazo*. Este tipo de prueba mezcla las dos anteriores. Crea, para cada posición del efector final, un *target* aleatorio situado siempre a 100mm del efector.

Uno de los problemas que presentaron estas pruebas fue que, cuando el brazo quedaba completamente estirado, la matriz $(H+\mu\cdot I)$ se hacía singular. Además, las primeras pruebas consistieron en colocar el *target* de forma completamente aleatoria, pudiendo situarse fuera del alcance del brazo y provocando situaciones singulares. Por otra parte, se interpretó erróneamente la estructura original del algoritmo de Levenberg-Marquardt, deduciéndose que si el algoritmo agotaba el número de iteraciones era porque no había alcanzado una solución correcta y, por lo tanto, ésta se debía desechar. Como resultado, más de 60% de las veces el algoritmo se veía incapaz de solucionar el problema[21].

[21] De estas primeras pruebas se adjunta, como parte del proyecto, dos vídeos con la ejecución del algoritmo en el sistema de *pruebaIK.xml*. El vídeo *Prueba de la esfera.mp4* muestra los

Otro problema al que se hizo frente fue el movimiento del brazo. Éste se movía desde su posición original hasta la de destino muy lentamente, de tal forma que a mitad del recorrido llegaba un nuevo *target*, interrumpiendo la trayectoria. Esto se hacía más grave cuando el algoritmo de Levenberg-Marquardt calculaba unos ángulos que superaban el rango de -π y π, retorciendo el brazo. Estos dos problemas se solventaron durmiendo la ejecución del programa de 5 a 8s, para que el brazo tuviera tiempo de llegar a la nueva posición, y creando el método *calcularModuloFloat*.

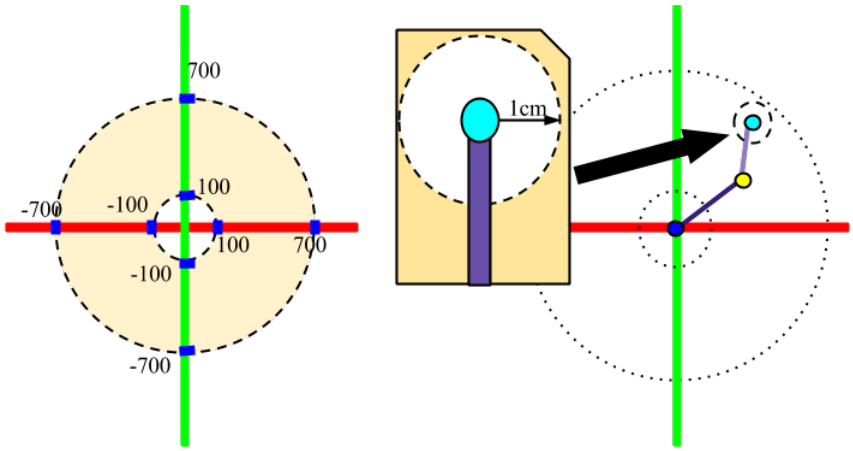


Illustration 36: Tipos de pruebas 1 y 2 realizadas sobre el sistema

Al arreglar el algoritmo, controlando las matrices singulares y eliminando el control que desechaba las soluciones que superaban el número de iteraciones, se consiguió un 100% de éxito en las tres pruebas, con errores del orden de 10^{-3}mm.

5.4.2 Segunda fase: trabajando en 3D con rotaciones

Con unos resultados tan aceptables como los obtenidos en la primera fase se aumentó la complejidad del componente *inverseKinematicsComp*. Así, el *Generador* se adaptó para crear *targets* que representaran puntos en el espacio tridimensional, con traslaciones y rotaciones (P_{target}=[Px, Py, Pz, Rx, Ry, Rz]).

resultados obtenidos al aplicar la prueba nº2 sobre el sistema, con un 100% de éxito. Por el contrario, el archivo *problemas con la inversa.mp4* muestra el caso del *target* fuera del alcance del brazo, que debe quedar estirado y que forma una matriz no invertible.

Sin embargo la adaptación de la clase *CinematicaInversa* para trabajar con las tres traslaciones y las tres rotaciones no resultó tarea fácil, a pesar de seguir usando la misma estructura simplificada del Levenberg-Marquardt, sin bloqueos ni controles adicionales. Por eso, esta segunda fase la dividiremos en varias subfases, explicando en cada una de ellas los problemas que se encontraron y las soluciones adoptadas.

5.4.2.1 Rotaciones con Ursus

Después de probar el correcto funcionamiento del algoritmo de Levenberg-Marquardt con las traslaciones en el sistema 2D del apartado anterior, el componente *inverseKinematicsComp* se ha adaptado para resolver exclusivamente los problemas de orientación y rotación del efector final en el espacio tridimensional.

Para ello se han necesitado aplicar varios cambios en la estructura de algunos métodos pertenecientes a las clases *SpecificWorker*, *CinematicaInversa* y *Generador*. El cambio más inmediato ha sido reestructurar los *targets,* que pasan de ser vectores con los dos componentes de traslación en X y en Y, P_{target}=[Px, Py], a vectores con los seis componentes completos, las tres traslaciones y las tres rotaciones en X, Y y Z, P_{target}=[Px, Py, Pz, Rx, Ry, Rz].

Como consecuencia del cambio de los *targets,* en la clase *Generador* se han tenido que desechar los métodos que creaban *targets* 2D y se ha desarrollado un nuevo método, *generarPuntosEsfera*:

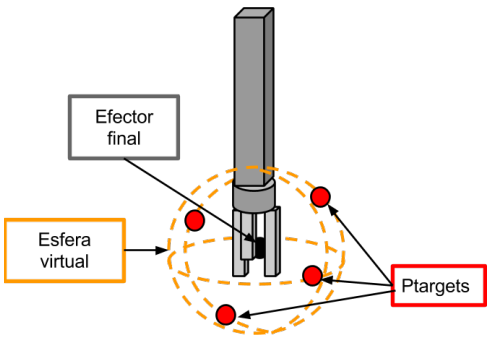


Illustration 37: Targets situados en una esfera virtual con centro en el efector

Este método crea *targets* 3D con traslaciones y rotaciones formando una esfera virtual de 100mm de radio con centro en el efector final del robot. Los *targets* generados se han

colocado relativamente cerca del efector final en traslación con la idea de probar, en el futuro, el algoritmo de Levenberg-Marquardt aunando rotaciones y traslaciones, mientras que las rotaciones se han calculado sumando a la rotación del efector final unas pequeñas perturbaciones.

Por otra parte, en la clase *CinematicaInversa* se han aplicado dos cambios: 1) el método encargado de calcular la matriz jacobiana, *Jacobian*, se ha ampliado para introducir las rotaciones, aunque todavía no hace distinciones entre joints prismáticos y no prismáticos y no hace cero aquellas columnas cuyos motores están bloqueados (porque no se está trabajando con motores restringidos), y 2) también se ha modificado la función de error del *computeErrorVector*, que calcula únicamente los errores de rotación.

Para probar los cambios realizados se ha utilizado el árbol cinemático descrito por el fichero *ursus2.xml*:

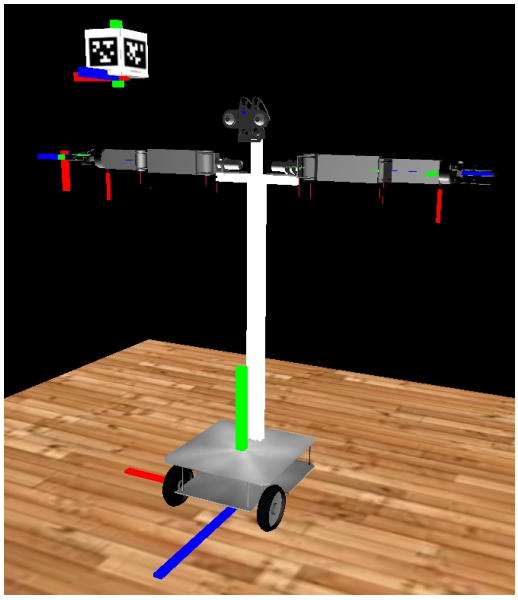


Illustration 38: Robot Ursus descrito en ursus2.xml

En este fichero (que utiliza como unidad de medida el milímetro) se describe la arquitectura del robot Ursus, a la cual dedicaremos próximamente un apartado completo para estudiarla en profundidad, y de la que, por ahora, nos interesa sólo su brazo derecho.

Por convención, los "huesos" de los brazos de cualquier robot crecen siempre en el eje Z, por lo que cada link de la cadena cinemática que forma el brazo derecho de Ursus sigue el eje Z del joint anterior. El brazo se compone de siete joints, sin límites ni restricciones en sus movimientos: *shoulder_right_1*, es el primer joint de la cadena y gira en Z; le sigue el *shoulder_right_2*, que gira en X; el *shoulder_right_3*, que gira en Z; el *elbow_right*, que gira en X; el *wrist_right_1*, que gira en Z; el *wrist_right_3* que gira en X; y el *wrist_right_4* que gira en Y. La cadena cinemática termina en el nodo transform *grabPositionHandR*.

La disposición de los joints y los links que los unen puede verse en la figura 39:

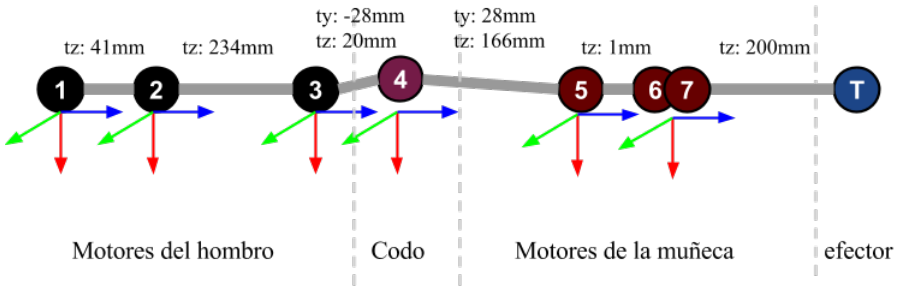


Illustration 39: Estructura del brazo derecho (visto de espaldas)

Al igual que antes, en el constructor del *SpecificWorker* se crean la lista con los nombres de los nuevos joints y la variable donde se guarda el nombre del efector final, y con ellos se inicializa la variable de *CinematicaInversa*. También se guarda la cola de *targets* que el *Generador* prepara, y con los que se modifican la traslación y orientación del cubo de marcas de AprilTags que vemos en la figura 38. Por último, en el bucle *compute* se toman los *targets* ordenados de la cola y se los pasa a la variable de *CinematicaInversa* para que los resuelva.

Como primer resultado, el algoritmo de Levenberg-Marquardt no fue capaz de encontrar unos valores angulares que rotasen el efector final como el *target*. Esto se debe a dos causas:

1. Una de las dos condiciones originales para entrar en el bucle externo del algoritmo de Levenberg-Marquardt, *while stop==false and k<kmax*, no se cumple. El umbral *stop*, calculado como $stop = \|g\|_\infty \leqslant \varepsilon_1$ es verdadero,
2. Si se consigue superar el umbral del *stop*, los incrementos calculados se consideran demasiado pequeños para aplicarlos y la condición $if\,(\|\delta_p\| \leqslant \varepsilon_2 \cdot (\|p\| + \varepsilon_2))$ es verdadera.

Ambos problemas se relacionan con los valores de las distintas constantes ε_i del algoritmo. Esto supone una dificultad importante ya que Lourakis y Argyros no explican de forma clara las magnitudes de los elementos con los que se comparan estos umbrales. Al final los umbrales se fijaron mediante prueba y error en $\varepsilon_1 = 0.001$, $\varepsilon_2 = 0.00000001$, $\varepsilon_3 = 0.0004$, $\varepsilon_4 = 0.0$ y $\tau = 10^{-3}$.

Con estos datos se ha realizado varios tipos de pruebas:

- Con el brazo extendido y utilizando exclusivamente los joints *elbow_right*, *wrist_right_1*, *wrist_right_3* y *wrist_right_4* se colocó un *target* con rotaciones a cero (orientado como el mundo). Al ejecutar el algoritmo de Levenberg-Marquardt, se giraron los joints quedando el efector final orientado como el *target*.

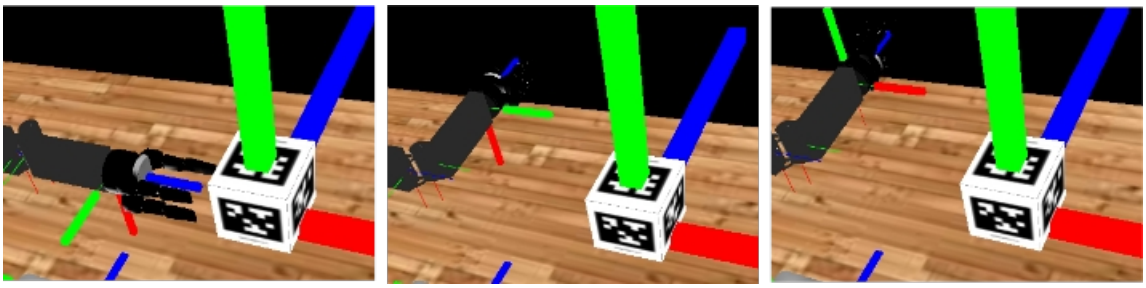


Illustration 40: Rotaciones de la muñeca y el codo del robot Ursus. Podemos ver cómo los ejes del sistema de referencia del target y el efector final (los ejes más gruesos) se orientan de la misma forma

- Con el brazo completamente extendido y utilizando los joints que forman la muñeca derecha del robot, *wrist_right_1*, *wrist_right_3* y *wrist_right_4*, se colocó un *target* con las rotaciones a cero (orientado igual que el mundo). Al

ejecutar el algoritmo de Levenberg-Marquardt el efector final quedó orientado como el *target*.

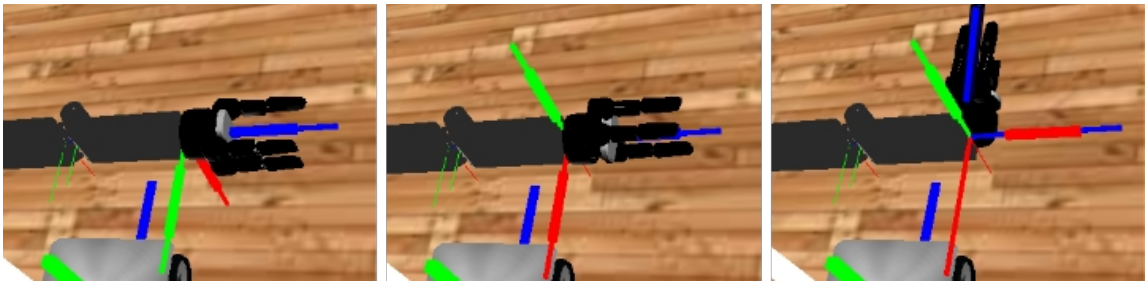


Illustration 41: Rotaciones de la muñeca de Ursus.

También se colocaron cien *targets* formando una esfera virtual de 10cm de radio con centro en el efector final alcanzándose en todas las pruebas el 100% de éxito, con un error del orden de los 10^{-2}mm. Estos resultados cambiarán de forma drástica cuando se introduzcan los límites de los motores.

5.4.2.2 Traslaciones y rotaciones con Ursus y Bender

Una vez comprobado que las rotaciones funcionan correctamente en el algoritmo de Levenberg-Marquardt, se unieron las traslaciones y las rotaciones, para llevar el efector final de la posición inicial con su orientación inicial hasta la posición deseada con la orientación deseada. Para ello, en la clase *CinematicaInversa*, se completó el método *computeErrorVector*, con los errores de traslación y de rotación, mientras que los umbrales del algoritmo de Levenberg-Marquardt se fijaron en $\varepsilon_1=0.0001$, $\varepsilon_2=0.00000001$, $\varepsilon_3=0.0004$, $\varepsilon_4=0.0$ y $\tau=10^{-3}$.

Se realizaron las mismas pruebas que en el apartado anterior sin obtener resultados positivos: al terminar la ejecución del algoritmo de Levenberg-Marquardt, el efector final ni se orientaba ni se trasladaba correctamente. Sin embargo, separadas las traslaciones y las rotaciones, el algoritmo si funcionaba, trasladando o rotando correctamente el efector.

Para encontrar la solución al problema, se decidió utilizar un escenario más sencillo, cambiando el robot Ursus (de siete grados de libertad) por el robot Bender (de seis grados de libertad), descrito en el fichero *BetaWorldArm.xm*:

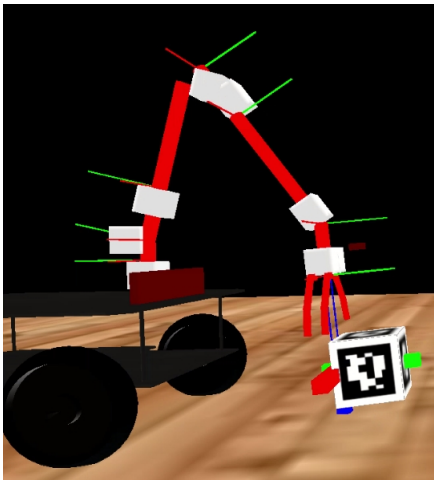


Illustration 42: Robot Bender descrito en el fichero BetaWordlArm.xml

Este robot está formado por una plataforma de la que sobresale un brazo de seis joints a los que se les eliminó los límites: *shoulder_right_1*, es el primer joint de la cadena cinemática que forma el brazo y gira en el eje Z; *shoulder_right_2*, gira en el eje X; *shoulder_right_3*, gira en el eje Z; *elbow_right*, que gira en el eje X; *wrist_right_1*, que gira en el eje Z; y *wrist_right_2*, que gira en el eje X:

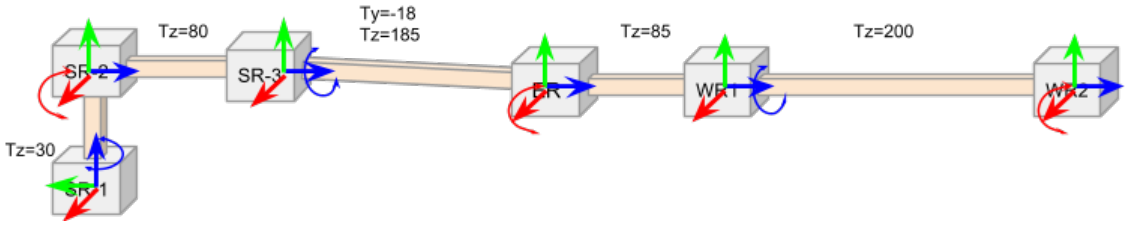


Illustration 43: Disposición de los joints del robot Bender

Para simplificarlo aún más, de todos estos joints sólo se utilizaron los que giran en el eje X: *shoulder_right_2*, *elbow_right* y *wrist_right_2*. En total sólo se usan tres grados de libertad. De esta forma los *targets* siempre se sitúan alineados sobre el eje Z del mundo y sólo con rotaciones en el eje X. Pero, a pesar de la simplificación, las rotaciones y las traslaciones seguían sin funcionar correctamente al juntarlas en el método de Levenberg-Marquardt.

El "problema" residía en:

```
Qvec res= axis.crossProduct(toEffector);
```

Este producto vectorial (situado en el método *Jacobian*) da como resultado un vector perpendicular, *res*, a los vectores *axis* y *toEffector*, y cuyo sentido depende del ángulo que formen estos dos últimos. El problema surge cuando distinguimos entre el producto vectorial de la mano derecha y el de la mano izquierda.

En el apartado 5.2.5.3 explicamos que el producto vectorial entre dos vectores $k \times v$ es lo mismo que multiplicar la matriz antisimétrica formada por los elementos del vector k por el vector v, $[k]_x \cdot v$. Sin embargo, los signos de los elementos de la matriz $[k]_x$ cambian, dependiendo de si se guían por la regla de la mano derecha o por la regla de la mano izquierda. Para explicar este fenómeno usaremos como ejemplo el producto vectorial entre los vectores *axis*=[0, 1, 0] y *toEffector*=[1, 0, 0] usando ambas reglas.

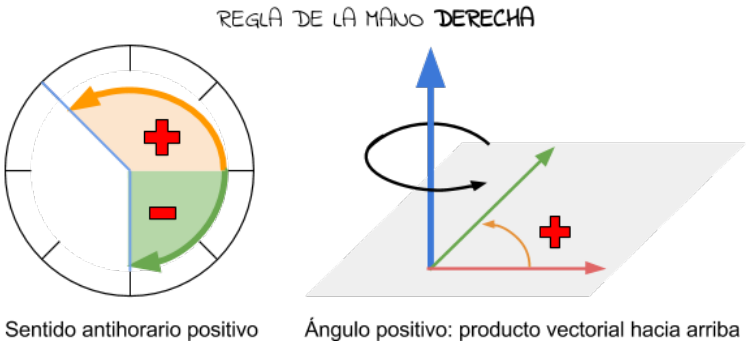


Illustration 44: Sentido de los ángulos y producto vectorial de la mano derecha

Para la **regla de la mano derecha**, el producto de la matriz antisimétrica $[k]_x$ por el vector v es:

$$[k]_x \cdot v = \begin{pmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & -k_1 \\ -k_2 & k_1 & 0 \end{pmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad .$$

Equation 24: Producto matriz por vector con la mano derecha

En el producto vectorial $axis \times toEffector$, donde *axis*=[0, 1, 0] (es el eje Y) y *toEffector*=[1, 0, 0] (es el eje X), la matriz antisimétrica se construiría con los elementos de *axis*, siguiendo los signos de la mano derecha, de tal forma que:

$$[axis]_x \cdot toEffector = \begin{pmatrix} 0 & -axis_z & axis_y \\ axis_z & 0 & -axis_x \\ -axis_y & axis_x & 0 \end{pmatrix} \cdot \begin{bmatrix} toEffector_x \\ toEffector_y \\ toEffector_z \end{bmatrix}$$

$$[axis]_x \cdot toEffector = \begin{pmatrix} 0 & -0 & 1 \\ 0 & 0 & -0 \\ -1 & 0 & 0 \end{pmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mathbf{-1} \end{bmatrix}$$

El vector resultado, *res*, tiene la dirección del eje Z con sentido negativo. Sin embargo, si calculamos el producto vectorial $toEffector \times axis$, la matriz antisimétrica se construye con los elementos del vector *toEffector*, $[toEffector]_x \cdot axis$, y se obtiene como resultado un vector con la dirección del eje Z y sentido positivo:

$$[toEffector]_x \cdot axis = \begin{pmatrix} 0 & -toEffector_z & toEffector_y \\ toEffector_z & 0 & -toEffector_x \\ -toEffector_y & toEffector_x & 0 \end{pmatrix} \cdot \begin{bmatrix} axis_x \\ axis_y \\ axis_z \end{bmatrix}$$

$$[toEffector]_x \cdot axis = \begin{pmatrix} 0 & -0 & 0 \\ 0 & 0 & -1 \\ -0 & 1 & 0 \end{pmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mathbf{1} \end{bmatrix}$$

Si representamos los resultados obtenidos de forma gráfica observamos que, para ángulos positivos el producto vectorial tiene signo positivo y "asciende", mientras que para ángulos negativos el producto vectorial tiene signo negativo y "desciende":

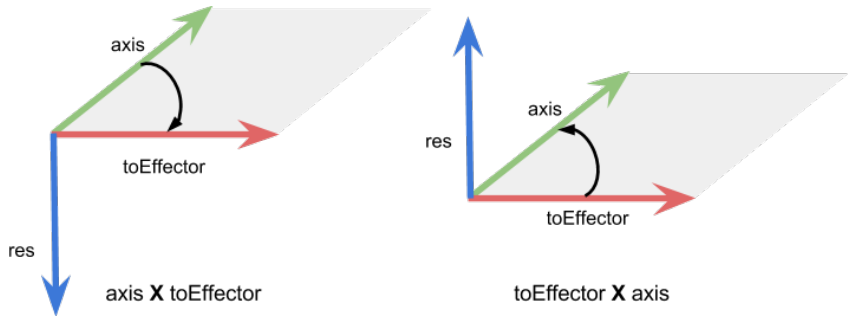


Illustration 45: Ejemplo de productos vectoriales con mano derecha

Esto cambiará en la mano izquierda, donde el sentido de los ángulos es justo el contrario.

Pongamos el mismo ejemplo pero utilizando esta vez la mano izquierda y el sistema de ejes utilizado por el framework Robocomp:

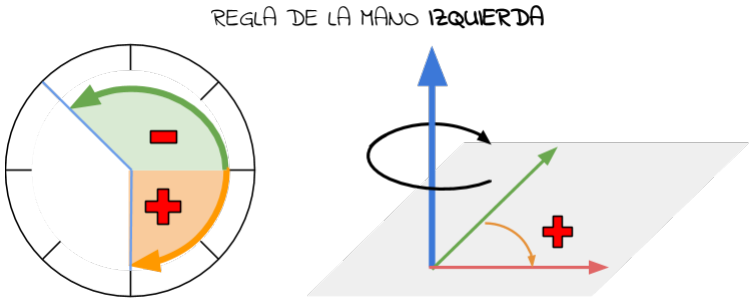


Illustration 46: Sentido de los ángulos y del producto vectorial con mano izquierda

Para la **regla de la mano izquierda**, el producto de la matriz antisimétrica $[k]_x$ por el vector v es:

$$[k]_x \cdot v = \begin{pmatrix} 0 & k_3 & -k_2 \\ -k_3 & 0 & k_1 \\ k_2 & -k_1 & 0 \end{pmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad .$$

Equation 25: Producto matriz por vector en la mano izquierda

En el producto vectorial $axis \times toEffector$, la matriz antisimétrica se construye con *axis*, siguiendo los signos de la mano izquierda, de tal forma que:

$$[axis]_x \cdot toEffector = \begin{pmatrix} 0 & axis_z & -axis_y \\ -axis_z & 0 & axis_x \\ axis_y & -axis_x & 0 \end{pmatrix} \cdot \begin{bmatrix} toEffector_x \\ toEffector_y \\ toEffector_z \end{bmatrix} = \begin{pmatrix} 0 & 0 & -1 \\ -0 & 0 & 0 \\ 1 & -0 & 0 \end{pmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mathbf{1} \end{bmatrix}$$

El vector resultado, *res*, tiene la dirección del eje Z con sentido positivo. Si calculamos el producto vectorial $toEffector \times axis$, la matriz antisimétrica se construye con los elementos del vector *toEffector,* y se obtiene *res* con la dirección del eje Z y sentido negativo:

$$[toEffector]_x \cdot axis = \begin{pmatrix} 0 & toEffector_z & -toEffector_y \\ -toEffector_z & 0 & toEffector_x \\ toEffector_y & -toEffector_x & 0 \end{pmatrix} \cdot \begin{bmatrix} axis_x \\ axis_y \\ axis_z \end{bmatrix}$$

$$[toEffector]_x \cdot axis = \begin{pmatrix} 0 & 0 & -0 \\ -0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mathbf{-1} \end{bmatrix}$$

Si, al igual que antes, representamos gráficamente los resultados obtenidos, vemos que para ángulos positivos, el producto vectorial mantiene el signo positivo (como en la mano derecha), mientras que para ángulos negativos, el producto vectorial tiene signo negativo:

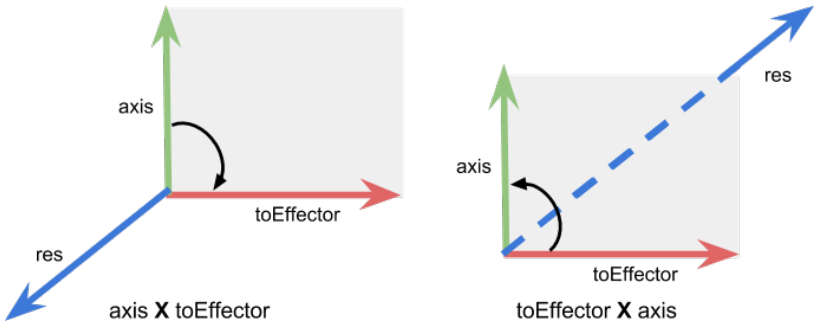


Illustration 47: Ejemplo de productos vectoriales con la mano izquierda

Comprobamos entonces que el producto $axis \times toEffector$, en la mano derecha, es igual que $toEffector \times axis$ en la mano izquierda, y que $toEffector \times axis$, en la mano derecha es igual a $axis \times toEffector$ en la mano izquierda. Entonces el problema se reduce a cambiar el sistema de referencia de la mano derecha por el sistema de referencia de la mano izquierda. Esto se puede hacer de varias formas:

1. Multiplicando la matriz antisimétrica utilizada por el método que calcula el producto vectorial, *crossProduct*, por -1.
2. Multiplicando el producto vectorial `axis.crossProduct(toEffector)` por -1.
3. Cambiar el orden de los multiplicandos, `axis.crossProduct(toEffector)`, por el `toEffector.crossProduct(axis)`, que devuelve el vector resultado de signo contrario.

Para solucionarlo, se optó por la tercera opción de tal forma que el problema al unir las rotaciones y las traslaciones desapareció, completando el algoritmo para poses enteras, con sus traslaciones y sus orientaciones.

Los cambios se ejecutaron de nuevo sobre el robot Bender, mediante dos tipos de pruebas[22]:

1. Se generó una trayectoria recta sobre el eje Z del mundo, mediante la colocación de puntos objetivos situados con una separación de 100mm y rotados:
    - +0.05 radianes en el eje X con respecto al *target* anterior
    - -0.05 radianes en el eje X con respecto al *target* anterior.
2. Se generó una trayectoria recta sobre el eje Y del mundo, mediante la colocación de *targets* situados a 100mm unos de otros y rotados:
    - +0.05 radianes en el eje X con respecto al *target* anterior.
    - -0.05 radianes en el eje X con respecto al *target* anterior.

En ambas pruebas se obtuvieron resultados muy satisfactorios: el efector era capaz de llegar con un error despreciable a aquellas posiciones del *target* dentro del rango de alcance y con unas rotaciones adecuadas.

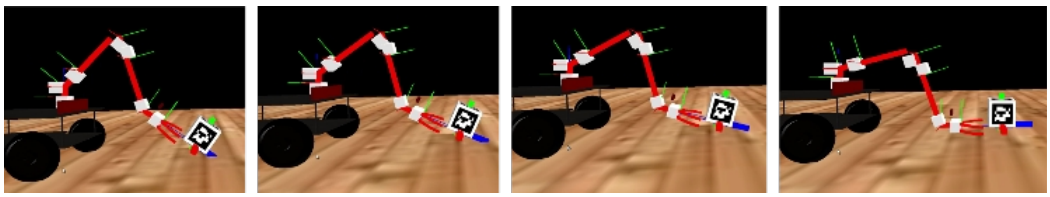


Illustration 48: Recta en Z con rotación en X de 0.05 radianes

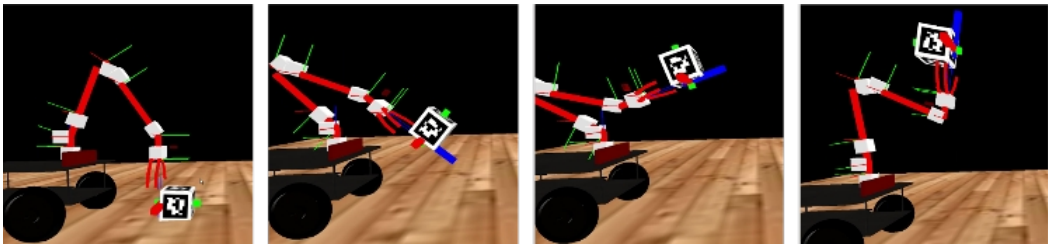


Illustration 49: Recta en Y con rotación en X de 0.05 radianes

[22] Estas pruebas quedaron recogidas en cuatro videos que se adjuntan al proyecto: *RectaZ1.mp4*, *RectaZ2.mp4*, *RectaY1.mp4* y *RectaY2.mp4*. En estos vídeos se aprecia que pesan más las rotaciones que las traslaciones, un error que se solventará en el próximo apartado.

5.4.3 Tercera fase: trabajando con pesos en las restricciones de traslación y rotación.

Con las traslaciones y las rotaciones funcionando juntas podríamos pensar que el proyecto está terminado y el problema, resuelto. Sin embargo no es así, existe un nuevo problema que es necesario resolver para el correcto funcionamiento del componente *InverseKinematicsComp*: al aunar traslaciones en milímetros con rotaciones en radianes se ha comprobado que, en el jacobiano, los componentes de traslación adquieren mucho peso, ya que suelen ser del orden de 10^3, frente a los componentes de rotación, que normalmente son del orden de 10^0 o 10^{-1}. A consecuencia de esto, se obtenían errores finales, cuanto menos, extraños, con una componente de traslación bastante buena, del orden de 10^{-4}, pero con la componente de rotación realmente mala (del orden de 10^2). Para subsanar ese desequilibrio se probó a pasar a metros los componentes de traslación directamente en el método *Jacobian*, obteniéndose como resultado un mayor peso en las rotaciones que en las traslaciones.

Para resolver este problema se añadió la matriz de pesos comentada en el apartado 5.3.1.1. Sin embargo no resultó fácil encontrar unos pesos que relacionasen correctamente los milímetros de las traslaciones con los radianes de las rotaciones. Se probaron los pesos (10^{-2}, 1) (donde el primer peso va asociado con las traslaciones y el segundo con las rotaciones), (10^{-3}, 1), (10^{-4}, 1) y (10^{-5}, 1), observándose que, a medida que disminuía el peso de las traslaciones el error aumentaba, sin que se apreciara una mejoría en las rotaciones.

Como resultado se decidió cambiar la unidad de medida de distancias que utiliza *InnerModel*, los milímetros, por la unidad de medida del sistema internacional, los metros. Esto obligó a modificar todos los ficheros xml que utiliza la clase *InnerModel* para representar el mundo interno del robot, transformando los valores de los atributos de traslación, tx, ty y tz, de milímetros a metros. Los pesos se simplificaron a **1**, para indicar que se tiene en cuenta la restricción sobre la que se aplica, y **0**, para indicar que no se tiene en cuenta la restricción sobre la que se aplica.

De esta forma, se pudo demostrar que existe una mejor y más equilibrada relación entre metros y radianes que entre milímetros y radianes. Sin embargo, modificar todo Robocomp, sus clases, librerías y herramientas, para que usen metros en vez de milímetros resulta una tarea gigantesca que se ha preferido no abordar, adaptando únicamente el componente *inverseKinematicsComp* para que, interiormente, haga las transformaciones de milímetros a metros sin que el resto de componentes, clases y librerías de Robocomp se vean afectadas. Sin enbargo, en la nueva versión de Robocomp se plantea trabajar en las unidades del sistema internacional.

5.4.4 Cuarta fase: trabajando con límites en los joints

El objetivo del componente *inverseKinematicsComp* no se queda en aplicar la cinemática inversa a un robot simulado, en un entorno exacto y perfecto, al contrario, el verdadero fin de este componente es aplicar la cinemática inversa en un robot real, en un entorno real.

Sin embargo, hasta este momento, estamos trabajando partiendo de la premisa de que los joints no están limitados en sus movimientos. Esto ayuda a la hora de probar que el algoritmo de Levenberg-Marquardt resuelve todos los *targets* que recibe dada una cadena cinemática perfecta. Sin embargo, en el mundo real no existen motores sin límites en sus movimientos, ya sea para protegerse del choque con el resto de la estructura del robot o porque tiene unos topes de fábrica, ni motores exactos con movimientos precisos, al menos, del orden que requiere el algoritmo.

Para solventar el problema de los límites se añadió al algoritmo de Levenberg-Marquardt el vector de joints bloqueados, junto al método *outLimits*, haciendo cero la columna correspondiente al motor bloqueado en la matriz jacobiana. Así, nos aseguramos que ningún joint supera sus límites mínimo o máximo, para que, al probarse sobre el robot real, no exista el peligro de giros imposibles o choques entre los links.

En esta fase también se crea el diccionario o mapa de partes del cuerpo del robot (en el que se asigna una etiqueta a una cadena de motores) para probar el componente en los brazos y la cabeza del robot. Al aplicar el algoritmo reformado sobre el robot simulado Ursus, colocando límites a los joints de ambos brazos, se obtuvieron resultados más

realistas: ahora no se pueden alcanzar posiciones con orientaciones demasiado forzadas, ni el robot puede doblar articulaciones como el codo o la muñeca hacia atrás, obligándolo a adoptar posturas más "humanas", pero reduciendo a la vez el número de poses alcanzables.

5.4.5 Quinta fase: externalización de los targets

Llegados a este punto, con casi todas las funcionalidades del componente trabajando correctamente, se han separado las funciones relacionadas con la cinemática inversa de las funciones relacionadas con la construcción del *target*. El objetivo principal es que el componente *inverseKinematicsComp* resuelva los *targets* que reciba como parámetros de entrada, generados por otros componentes. Por ejemplo, si un componente encargado de la visión del robot capta a través de una cámara un *target* y lo envía al *inverseKinematicsComp* para que lo resuelva y mueva la cadena cinemática asociada a él para situar el efector final en la pose deseada.

Para ello se han implementado los métodos de la interfaz de *inverseKinematicsComp*:

1. Los que mueven el robot a una posición determinada. Por ejemplo *goHome* envía la parte del cuerpo a la posición de home mediante cinemática directa. Es el único método del proyecto que utiliza posiciones angulares fijas para los motores sin usar la cinemática inversa.
2. Un método especial es *setFingers*, que aplica un pequeño cálculo de cinemática inversa para abrir y cerrar las pinzas de la mano del robot.
3. Los que envían *targets* para ser resueltos por la cinemática inversa: estos métodos encolan el *target* que reciben como parámetro de entrada en la cola de *targets* pendientes de la parte del cuerpo del robot al que va dirigido. De esta forma se han podido incluir los distintos tipos de *targets* que puede resolver la cinemática inversa, adaptando los métodos necesarios para poder contemplarlos.
4. También se ha implemnetado un nuevo método para la interfaz, *stop*, que detenga la ejecución del algoritmo de Levenberg-Marquardt y limpie las colas de *targets* pendientes de las partes del cuerpo del robot. Útil para abortar tareas y como botón de seguridad para el robot real.

Para probar el nuevo funcionamiento del componente *inverseKinematicsComp*, se ha tenido que desarrollar un nuevo componente, *inverseKinematicsTesterComp*, encargado de "crear" los distintos tipos de *targets* y enviárselos al componente de cinemática inversa, al que se le ha añadido la opción de ejecutar el resultado devuelto por el algoritmo de Levenberg-Marquardt en el simulador de *InnerModel* o en el robot real.

inverseKinematicsTesterComp es esencialmente una interfaz de usuario que dispone de:

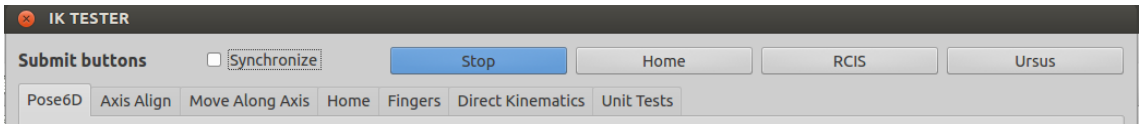


Illustration 50: Botones y pestañas de la interfaz de inverseKinematicsTesterComp

- En la parte superior, aparte de la opción *synchronize* (que sirve para sincronizar el robot con el simulador de *InnerModel*), tiene una serie de botones de control:
  - **Stop**: llama al método *stop* de la interfaz para parar la ejecución de un *target* y limpiar las colas de *targets* pendientes.
  - **Home**: envía todas las partes del cuerpo del robot a la posición de home a través del método *goHome*.
  - **RCIS**: indica que el *target* o la lista de *targets* se ejecutarán en el simulador de *InnerModel*.
  - **Ursus**: indica que el *target* o la lista de *targets* se ejecutarán en el robot real.
- Debajo de los botones dispone de siete pestañas:
  - **Pose6D**: (figura 51) en esta pestaña se crean *targets* de tipo POSE6D. El usuario puede crear un único *target* ,con sus traslaciones, orientaciones y su matriz de pesos, y enviárselo a una o varias partes del cuerpo del robot, o seleccionar una trayectoria completa

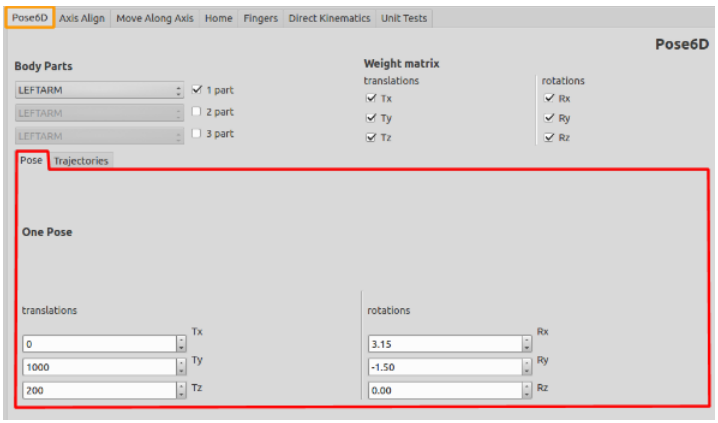


Illustration 51: Pose6D. Creación de un único target

- **Axis Aling**: (figura 52) en esta pestaña se pueden crear *targets* de tipo AXISALING y enviárselos a una o varias partes del cuerpo del robot. El usuario puede elegir en qué eje está situado el *target* y con qué traslación y orientación.

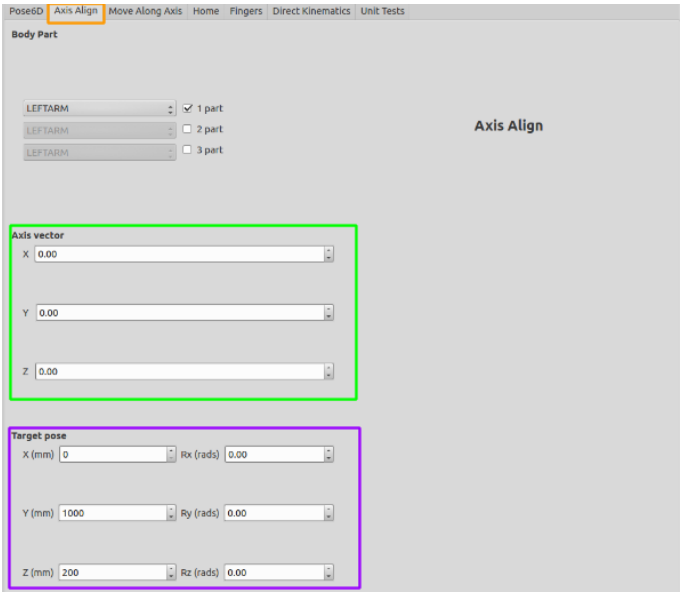


Illustration 52: Pestaña Axis Aling

- **Move Along Axis**: en esta pestaña se pueden crear *targets* de tipo ADVANCEAXIS, y enviarlos a una o varias partes del cuerpo del robot. El usuario puede elegir en qué eje (o composición de ejes) quiere avanzar y cuánto quiere avanzar sobre él.

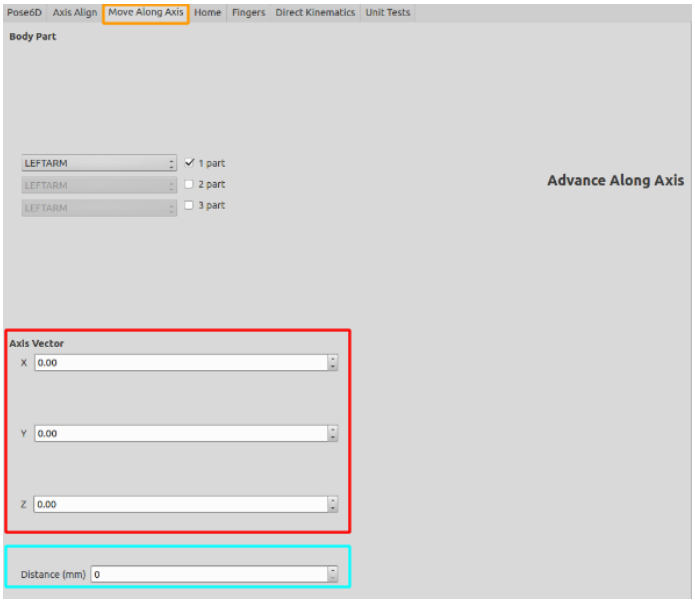


Illustration 53: Pestaña Move Along Axis

- **Home**: es esta pestaña se envía a la posición de home una, varias, o todas las partes del cuerpo del robot.
- **Fingers**: en esta pestaña se puede indicar cuánto quiere el usuario abrir o cerrar las pinzas de la mano del robot.
- **Direct Kinematics**: esta pestaña muestra los valores angulares actuales de cada joint, así como sus límites mínimo y máximo. También muestra una copia del simulador de *InnerModel*.
- **Unit Test**: esta pestaña recoge los pasos que debe seguir el robot, tanto real como simulado, para tomar un objeto en una posición de destino, colocarlo sobre una bandeja y ofrecerlo, como si de un camarero se tratase. Se ha utilizado principalmente para comprobar que la cinemática inversa funciona correctamente sobre el robot real.

5.4.6 Sexta fase: troceando el target

Al explicar el algoritmo de Levenberg-Marquardt, vimos que éste dependía demasiado de la situación inicial en la que comenzaba a ejecutarse, los x^0. A consecuencia de esto, cuando se ejecuta varias veces el algoritmo sobre el mismo *target*, pero partiendo de

posiciones distintas, se obtienen resultados muy diferentes: en algunos casos se llega muy bien a la posición de destino, con un error despreciable, pero en otros simplemente no se alcanza.

Para evitar este problema se ha añadido la técnica del **troceado del *target***. Esta técnica[23] divide la trayectoria que separa al efector final del *target* asignado en pequeños *subtargets*, relativamente fáciles de resolver por Levenberg-Marquardt, siguiendo la fórmula:

$$R=(1-\lambda)\cdot P+\lambda\cdot Q \quad .$$

Equation 26: Fórmula para trocear rectas

Donde λ es un número que va desde el 0 hasta el 1, P es la pose del efector final y Q es la pose del *target*. Como resultado se obtendrá R, un punto entre P y Q, que estará más cerca de P o de Q dependiendo de λ:

1. Cuando λ=1: $R=(1-1)\cdot P+1\cdot Q=0\cdot P+1\cdot Q=Q$
2. Cuando λ=0: $R=(1-0)\cdot P+0\cdot Q=1\cdot P+0\cdot Q=P$

Esto se entiende mucho mejor con un ejemplo, así que supongamos que tenemos un efector colocado en el punto x=0 y un *target* situado en el punto x=10, y que por lo tanto distan uno del otro 10cm, como la situación que nos marca la figura 54:

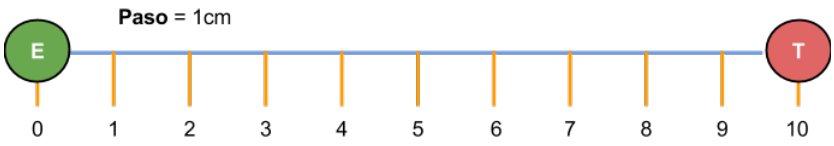


Illustration 54: Distancia entre el efector final y el target en cm

Y supongamos que queremos que la máxima distancia entre el efector final y el *target* sea de 1cm, superado el cual la trayectoria entre el efector y el *target* debe trocearse. En este ejemplo, como el *target* está 10cm de distancia y 10cm>1cm, debemos trocear la trayectoria entre el efector y el *target*.

[23] Se trata de la estrategia "divide et impera", tantas veces utilizada en informática. Ésta consiste en dividir el problema en subproblemas más pequeños, resolviéndolos por separado y uniendo las "sub-soluciones" para encontrar una solución global al problema original.

Para aplicar la ecuación $R=(1-\lambda)\cdot P+\lambda\cdot Q$ asignaremos la posición del efector a P, P=0, y la del *target* a Q, Q=10. Ahora debemos asignar un valor a λ entre 0 y 1. Para ello calculamos el número de puntos intermedios entre el efector y el *target* dividiendo la trayectoria entre el paso o distancia máxima: puntos=distancia/paso=10/1=10 puntos. De esta forma λ será la inversa del número de puntos: λ=1/puntos=1/10=0.1. Ya podemos aplicar la fórmula:

$$R=(1-0.1)0+0.110=1$$

El *subtarget* calculado estará en la posición x=1cm, como muestra la siguiente figura:

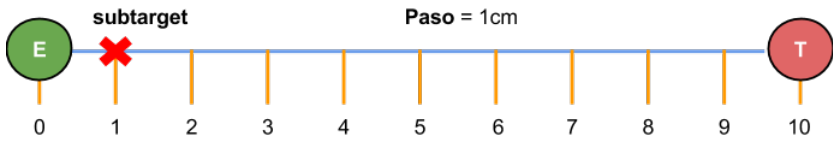


Illustration 55: Subtarget calculado entre el efector y el target

Podemos comprobar que este método divide la trayectoria en "trocitos" que no superan el tamaño que indique el paso que hemos marcado.

Tras una intensa batería de pruebas se encontraron una serie de problemas, uno de ellos es que el efector esté demasiado lejos del *target*, por ejemplo porque el *target* esté fuera del rango de alcance del brazo. En estos casos, el método de troceado entra en un bucle infinito, enviando al Levenberg-Marquardt el último *subtarget* al que el efector es incapaz de llegar.

También se ha detectado que, al trabajar con el robot real, el método de troceo del *target* puede repetir una serie de *subtargets* creando patrones repetidos en el tiempo. Para evitar estos dos problemas se han codificado dos métodos en la clase *CinematicaInversa*: *chopPath* y *comprobarBucleChop*.

El método *chopPath* se encarga de calcular el *subtarget* entre el efector final y el *target*. Responde a la siguiente estructura:

```
Entrada: Recibe como parámetro de entrada una variable de tipo Target,
target, que representa el punto objetivo a resolver por la IK
Algoritmo:
```

```
/*Lista de vectores auxiliar donde guardamos los subvectores que
calcula el método. Sirve para detectar patrones repetidos cuando el
chop entra en un bucle.*/
 Static QList<QVec> listaSubtargets;
/*Toma el target en el efector final y hace la norma de sus
coordenadas de traslación y de rotación. Si es mayor a un umbral,
divide el target en un subtarget*/
QVec  targT_inTip=innerModel.transform(effector,[0,0,0],target.name);
QMat  m=innerModel.getRotationMatrix(effector,target.name);
QVec  targR_inTip=m.extractAnglesRmin();
QVec  targ_total=[targT_inTip,targR_inTip];
const float step=0.1;
float dist=  ||target.getWeights·targ_total||
if(dist>step)
  int nPuntos=dist/step;
  T λ=1/nPuntos;
  QVec   P=[innerModel.tranform("world",[0,0,0],efector),   innerModel.
  getRotationMatrix("world", efector).extractAnglesR_min()];
  /*Calculamos el subtarget entre el target original y el efector*/
  QVec R=P*(1-λ)+target.getPose*λ;
  /*Existe el problema de que el IK no pueda llegar al target final
  debido a los límites de los joints, a que el target sea un posición
  muy retorcida...etc. En estos casos el método chopPath calcula el
  error entre el efector y el target y vuelve a trocear, creando un
  bucle: troceo y mando subtarget -no puedo llegar al subtarget -
  vuelvo a trocear y envío el mismo subtarget - sigo sin poder ir.
  Para estos casos se ha añadido el siguiente control*/
  //Target total multiplicado por λ:
  Qvec  R_2=targ_Total*λ;
  //Matriz de rotación del efector en el mundo:
  QMat  matEff_inWorld=innerModel.getRotationMatrix("world",effector);
  /*Matriz de rotación con valores angulares de R2 (errores de
  rotación intermedios entre el efector final y el target*/
  Rot3D  matError_inTip(R_2[3],R_2[4],R_2[5]);
  /*Matriz para pasar los errores de rotación vistos desde el mundo*/
  QMat  mat_result=matEff_inWorld·matError_inTip;
  /*Sacamos error de traslación y de rotación en el mundo*/
  Qvec  error_T=innerModel.tranform("world",R_2.subvector(0,2),effector);
  Qvec  error_R=mat_result.extractAnglesRmin();
  /*Componemos el nuevo vector R:*/
   R=[error_T,error_R];
  /*Comprobamos que no sea el mismo subvector que se le envió antes o
  que no está formando un patrón repetido*/
  if(comprobarBucleChop(listaSubtargets, R)==true)
       /*Cuando el subtarget actual es muy parecido o igual al
       subtarget anterior, eliminamos el chop y limpiamos*/
```

```
        target.setChopped(false);
        listaSubtargets.clear();
    else
        /*Si es un subtarget distinto, lo guardamos en la lista, lo
        mandamos al Levenberg-Marquardt, actualizamos el target y
        levantamos bandera de chop*/
        listaSubtargets.add(R);
        InnerModel.updateTransformValues(target.name, R.x, R.y, R.z,
                                         R.rx, R.ry, R.rz);
        target.setChopped(true);
        target.setChoppedPose(R);
        target.setExecuted(false);
    endif
  else
    /*Si el error entre el target y el efector final no supera el
    umbral, no troceamos y limpiamos*/
    target.setChopped(false);
    listaSubtargets.clear();
  endif
```

El método *comprobarBucleChop* se encarga de buscar *subtargets* repetidos en la lista de *targets* o muy parecidos. Cuando detecta que el método *chopPath* ha entrado en un bucle envía TRUE para detener el troceado. Su estructura es la siguiente:

Entrada: Recibe como parámetro de entrada la lista de subtargets calculada por el método ChopPath, listaSubtargets, y el vector con la pose del nuevo subtarget calculado por el chopPath, subtarget.

Salida: devuelve FALSE si no detecta subtargets repetidos y TRUE si detecta bucles.

Algoritmo:

```
  /*Booleano para detectar bucles*/
  bool subtargetRepetido = false;
  /*Umbrales con los que comparar targets dentro d ella lista*/
  const float minTraslaciones = 0.0001;
  const float minRotacioness = 0.001;
  /*Si la lista de subtargets no está vacía hace las comprobaciones*/
  if(listaSubtargets.isEmpty()==false)
    /*Si el nuevo subtarget calculado por el chopPath ya está dentro de
    la lista levanta la bandera*/
    if(listaSubtargets.contains(subtarget))
      subtargetRepetido=true;
    else
      /*Recorremos la lista y comparando los elementos almacenados. Si
      encuentra alguno parecido al subtarget levanta bandera y sale*/
      for(int i=0; i<listaSubtargets.size(); i++)
        QVec anterior = listaSubtargets.at(i);
```

```
            if(||subTarget[0]-anterior[0])<minTraslaciones|| and
                ||subTarget[1]-anterior[1])<minTraslaciones|| and
                ||subTarget[2]-anterior[2])<minTraslaciones|| and
                ||subTarget[3]-anterior[3])<minRotaciones|| and
                ||subTarget[4]-anterior[4])<minRotaciones|| and
                ||subTarget[5]-anterior[5])<minRotaciones)
                    subtargetRepetido=true;
                    break;
            endIf
        endFor
    endIf
endIf
```

Como resultado, el algoritmo de Levenberg-Marquardt es capaz de llevar el efector del robot a una misma posición objetivo desde distintos puntos de partida uniformemente y se evitan bucles y *subtargets* repetidos.

6. Ursus, el robot social de Robolab

El objetivo del componente *inverseKinematics* es ser utilizado como para resolver los problemas de la cinemática inversa de diferentes robots reales, como los robots desarrollados por el laboratorio Robolab, Ursus y Loki. El objetivo es dotar al robot en cuestión de autonomía, para que sea capaz de alcanzar objetivos no prefijados y manipular objetos cotidianos de nuestro entorno, como puede ser una taza, el periódico...

Para ello, el componente *inverseKinematicsComp* ha sido probado en el robot social Ursus, obteniéndose buenos resultados.

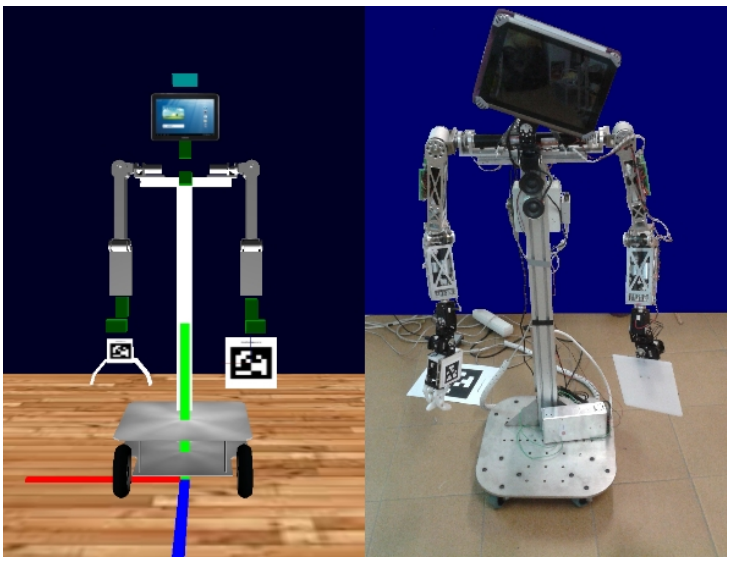


Illustration 56: Robot Ursus, en el simulador y en la realidad

Este robot está pensado para que interactúe de forma autónoma con los seres humanos, desplegando las normas de comportamiento social en sus entornos diarios. La primera versión de Ursus[24], del 2011, consistió en desarrollar un robot asistencial de bajo coste, dedicado a interaccionar con niños de 5 a 10 años con parálisis cerebral o con discapacidad en las extremidades superiores. De apariencia amable, forrado con la piel de un oso de peluche y con dos brazos móviles, su objetivo es plantear juegos que

[24] Desarrollado en colaboración por el Laboratorio de Robótica y Visión Artifical de la Universidad de Extremadura (Robolab) y el Hospital Universitario Vírgen del Rocío de Sevilla.

mejoren la capacidad motriz y de recuperación de los niños, corrigiendo sus movimientos mediante una kinect y monitoreando la evolución clínica del paciente.

Illustration 57: Versiones 1.0 y 2.0 del robot Ursus

La (por ahora) última versión del robot Ursus, 3.0, está pensada para, además de ser utilizada como un robot terapéutico, emplearla como un robot social genérico, capaz de moverse, comprender e interactuar con el entorno humano.

En los siguientes apartados explicaremos de forma simplificada la estructura del robot definida en el fichero XML de *InnerModel*, *ursusMM.xml*, así como la estructura del proyecto donde se enmarca los componentes desarrollados, *inverseKinematicsComp* e *inverseKinematicsTesterComp*.

6.1 Estructura del robot Ursus

En la descripción que hace el fichero XML de *InnerModel*, el robot Ursus consta de una base diferencial con tres ruedas (dos direccionales y una de apoyo) que gira sobre el eje Y del mundo. Ha sido construida en el laboratorio de Robolab y en ella, además, se colocarán los elementos electrónicos encargados del control del movimiento de la base.

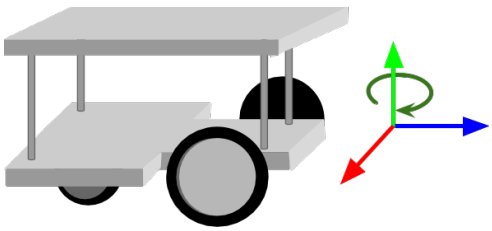


Illustration 58: Base del robot Ursus

Sobre la base se sitúa (girado -π/2 radianes en el eje X para colocar el eje Z hacia arriba, eje en el que por convenio, crecen los "huesos" del robot) el joint "*body*", que gira sobre el eje Z entre -1 y 1 rad. Del joint nace el eje que sostendrá la cabeza y los brazos del robot. El eje o columna vertebral mide cerca de 930mm de largo y a 15mm sobre él se coloca el eje transversal en cuyos extremos se sitúan los hombros.

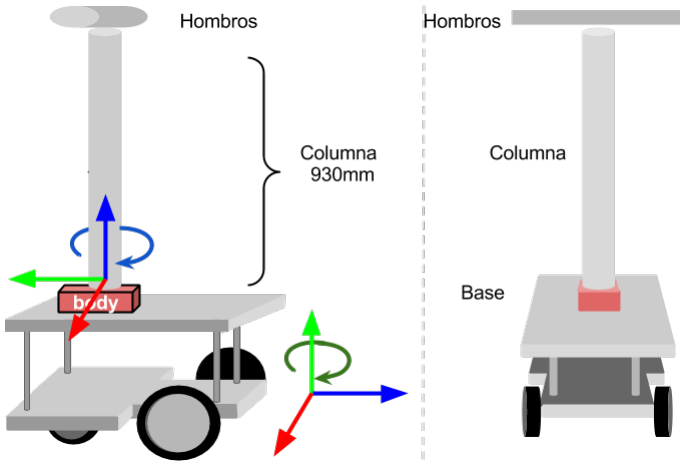


Illustration 59: Columna vertebral de Ursus

La base de la cabeza de Ursus, "*base_head*", se coloca sobre la columna vertebral a 24mm y está rotado π/2 radianes en el eje X. La cabeza cuenta con tres joints:

- "*head1*" que gira en el eje X entre -0.05 y 0.6 radianes.
- "*head2*", trasladado 73mm en el eje Y con respecto al joint anterior y rotado -π/2 en el eje X. Gira entre -0.3 y 0.3 radianes en el eje Y.
- "*head3*", trasladado 100mm con respecto al segundo joint, gira en el eje Z y por ahora no se le han añadido límites en su movimiento.

Después de "*head3*" se introduce una series de transforms que, primero rotan el sistema de referencia – π/2 radianes en ele eje X, y después lo trasladan 125mm en Y y 27mm en Z. Sobre ese sistema de referencia se coloca la cámara RGBD, que puede tener una rotación variable y que actualmente está a 1.05 radianes en el eje X con el objetivo de que pueda ver el entorno de trabajo de sus brazos. Esta cámara dispone de una distancia focal de 525 píxeles en el dispositivo real y 585 píxeles en el IK. Apoyándose en ella y en el componente *InverseKinematicsComp*, se pretende que el robot Ursus reconozca

objetos, marcas y posiciones y sea capaz de llevar la mano hacia la pose de destino. Por debajo de la cámara se coloca una tablet cuyo objetivo es hacer de cara para el robot. Para ello se está desarrollando de forma paralela otro componente para Robocomp que sea capaz de detectar emociones y modificar la cara del robot con las expresiones faciales asignadas a cada emoción.

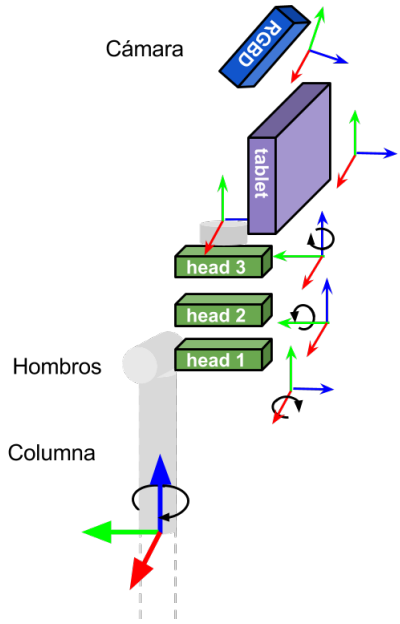


Illustration 60: Cabeza de Ursus

Pasamos a explicar el brazo derecho del robot Ursus. A pesar de haber hecho una breve introducción sobre este brazo en el apartado 5.4.2.1, no especificamos los límites en los que trabajan los distintos joints:

1. El primer joint del brazo derecho es el "*rightShoulder1*". Se sitúa a 990mm de la base, y está rotado π/2 radianes tanto en el eje Y como en el eje Z, con el objetivo de dejar el eje Z como la dirección de crecimiento del hueso. Gira en el eje Z entre -π y 0.54 radianes.
2. El segundo, "*rightShoulder2*" está trasladado 41mm en Z y rotado π/2 radianes en el eje X. Este joint gira entre -π y 0.34 radianes en el eje X.
3. El tercer joint, "*rightShoulder3*", se encuentra a 234mm del segundo joint y está rotado -π/2 radianes en el eje Z. Esta articulación gira entre -1.80 y 1.8 radianes en el eje Z.

4. El cuarto joint es el codo, "*rightElbow*". Se sitúa a -28mm en el eje Y y 20mm en el eje Z con respecto al anterior joint. Rota entre 0.01 y 2.6 radianes en el eje X.
5. El quinto joint es el primer giro de la muñeca derecha, "*rightForeArm*". Está colocado a 28mm en Y y 166mm en Z del codo. Rota entre -1.8 y 1.8 radianes en el eje Z.
6. El sexto joint es el "*rightWrist1*". Está trasladado con respecto al "*rightForeArm*" 60mm en el eje Z. Este joint gira entre $-\pi/2$ y $\pi/2$ radianes en el eje X.
7. El séptimo y último joint del brazo derecho se corresponde con el "*rightWrist2*". Está trasladado 60mm en el eje Z del "*rightWrist1*" y gira entre -0.1 y $\pi/2$ radianes en el eje Y.

Este brazo acaba en tres dedos, dos paralelos (movidos por el joint "*rightFinger1*") y uno opositor (movidos por el joint "*rightFinger2*") para poder hacer pinza. Ambos joints giran en el eje Y, entre $-\pi/2$ y 0 radianes el primero, y entre 0 y $\pi/2$ radianes el segundo. Por último, el efector final tenido en cuenta para los cálculos de cinemática inversa, "*grabPositionHandR*", se ha colocado a 50mm en Z del último motor de la muñeca, "*rightWrist2*".

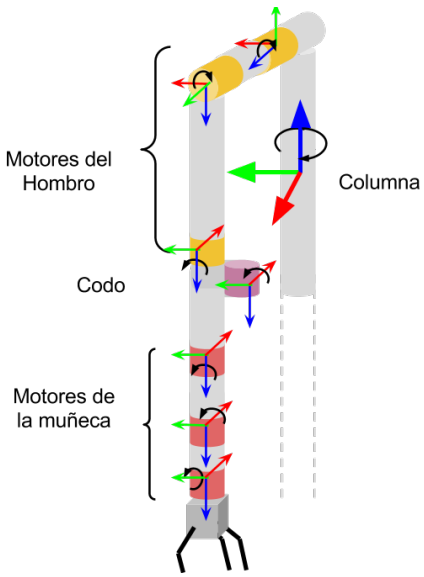


Illustration 61: Brazo derecho de Ursus

Para finalizar este apartado vamos a explicar el brazo izquierdo del robot. Es muy parecido al brazo derecho en cuanto a número y disposición de los joints:

1. El primer joint del brazo izquierdo es el "*leftShoulder1*". Está situado a 990mm de la base, y está rotado con respecto a ella -π/2 radianes en el eje Y para dejar el eje Z como la dirección de crecimiento del hueso. Gira en el eje Z entre -0.54 y π radianes.
2. El segundo joint es "*leftShoulder2*". Está trasladado 41mm en el eje Z con respecto al primero y está rotado -π/2 radianes en el eje X. Este joint gira entre -0.34 y π radianes en el eje X.
3. El tercer joint es el último motor del hombro del robot, "*leftShoulder3*". Se encuentra a 234mm en el eje Z del segundo joint y está rotado π/2 radianes en el eje Z. Esta articulación gira entre -1.80 y 1.8 radianes en el eje Z.
4. El cuarto joint es el codo izquierdo, "*leftElbow*". Se sitúa a 28mm en el eje Y y 20mm en el eje Z con respecto al anterior motor. Rota entre -2.6 y 0 radianes en el eje X.
5. El quinto joint es el primer giro de la muñeca izquierda, "*leftForeArm*". Está colocado a -28mm en Y y 166mm en Z del codo. Rota entre -2.3 y 2.3 radianes en el eje Z.
6. El sexto joint es el "*leftWrist1*". Está trasladado 63mm en el eje Z con respecto al "*leftForeArm*", y gira entre -π/2 y π/2 radianes en el eje X.
7. El séptimo y último joint del brazo izquierdo se corresponde con el último motor de la muñeca, el "*leftWrist2*". Está trasladado 60mm en el eje Z y rota entre -0.1 y π/2 radianes en el eje Y.

A diferencia del brazo derecho, el brazo izquierdo no acaba en unos dedos que hacen pinza, sino en una tabla rectangular. Esto es así para emular, durante las pruebas del *inverseKinematicsComp*, los movimientos de un camarero que toma objetos con su mano derecha y los deposita en una bandeja sujetada por la mano izquierda.

Por otra parte, el efector final utilizado en este brazo para realizar los cálculos de cinemática inversa, "*grabPositionHandL*", se ha rotado para que su sistema de

referencia no coincida con el del último joint y, así, poder probar el método del cálculo del vector de error explicado en el apartado 5.3.4.

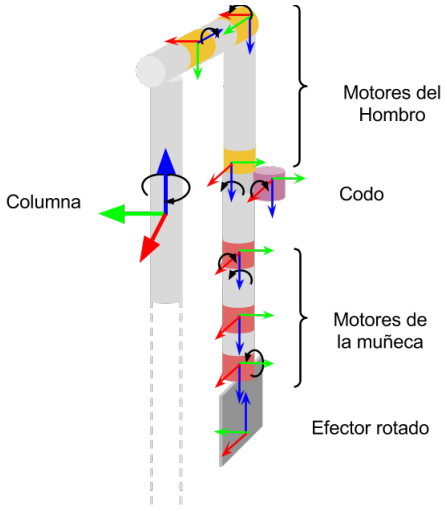


Illustration 62: Brazo izquierdo del robot Ursus

6.2 Grafo de componentes

A continuación explicaremos la estructura final del proyecto en el que se enmarca el componente de cinemática inversa, *inverseKinematicsComp*.

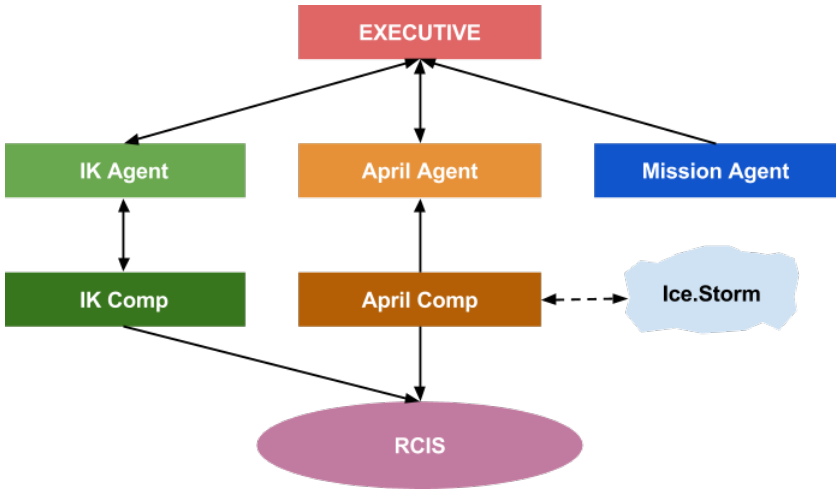
En este proyecto podemos distinguir dos partes bien diferenciadas:

1. La parte sobre simulación: esta parte del proyecto se ejecuta enteramente sobre el simulador de Robocomp, RCIS. Su propósito es depurar errores y comparar los resultados obtenidos en el robot real (con sus holguras, sus errores de calibración y los problemas de odometría) con los que se obtienen en el robot simulado.
2. La parte sobre el robot real: esta parte del proyecto se ejecuta sobre el robot real, Ursus.

Analizaremos las estructuras de estas dos partes del proyecto por separado mediante los grafos de componentes desarrollados por Robocomp.

6.2.1 La parte sobre la simulación

La estructura de esta parte del proyecto queda recogida en el siguiente grafo. En el que podemos ver claramente siete componentes distintos:



Graph 1: Grafo de componentes de la parte sobre simulación

Estos componentes se pueden clasificar en varios tipos:

1. Los que se encargan de la representación simbólica del mundo del robot:
    1. **Mission Agent**: este componente envía la misión (*set mission*) o propone al ejecutivo el estado final del grafo simbólico (AGM o *Active Grammar-Based Modeling*[25]).
    2. **Executive**: este componente deduce como tiene que llegar al estado final deseado y planifica las acciones a llevar a cabo.
    3. **April Agent**: este agente propone cambios al grafo simbólico mediante la información del componente *aprilComp*.
    4. **IK Agent**: propone cambios en el grafo simbólico cuando la acción de cinemática inversa es llevada a cabo. También ejecuta acciones de cinemática inversa cuando el Executive así lo requiere.

[25] Para profundizar en los grafos simbólicos se recomienda la lectura de la tesis doctoral de Manso, L. (2013) "*Perception as Stochastic Grammar-based Sampling on Dynamic Graph Spaces*"

2. Los componentes básicos:
    1. **April Comp**: Reconoce marcas de la familia *aprilTags* y devuelve su identificación y su posición (componentes de traslación y de rotación) en el sistema de referencia de la cámara que las detecta. Se comunica, en este caso, con el RCIS a través de la interfaz RGBD.ice.
    2. **IK Comp**: es el componente *inverseKinematicsComp* y se encarga de resolver los *targets* que le envían. También es posible utilizar el *inverseKinematicsTesterComp* para enviar *targets* específicos y realizar determinadas comprobaciones.
3. El simulador de Robocomp:
    1. RCIS: el *RoboComp InnerModel Simulator*, encargado de definir y simular el mundo interno del robot.

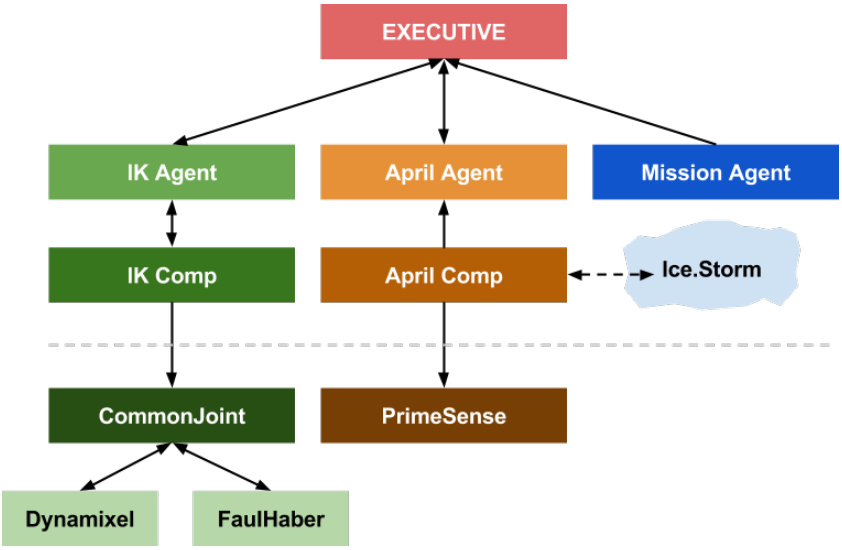
Por otra parte tenemos el **Ice.Storm** que implementa el tipo de comunicación publicación-subscripción. Hace las veces de buzón al que envían los datos aquellos componentes que publican, como es el caso de *AprilComp*. Por otra parte, los componentes que necesitan recibir esos datos se suscriben a él o a un tópico.

6.2.2 La parte sobre la realidad

La estructura de esta parte del proyecto, que se ejecuta sobre el robot Ursus, queda recogida en el segundo grafo. En él podemos comprobar que la parte de la representación simbólica del mundo y del robot se mantiene constante, así como los componentes básicos encargados de la detección de marcas y de la resolución de la cinemática inversa (por encima de la línea discontínua). Sin embargo el RCIS desaparece y es reemplazado por cuatro componentes nuevos:

1. **PrimeSense**: devuelve la imagen de profundidad y de color de un dispositivo RGBD. El componente *AprilComp* se comunica con él a través de la interfaz RGBD.ice.
2. **CommonJoint**: construye un joint virtual a partir de todos los joints del robot descritos en los componentes *Dynamixel* y *FaulHaber*, facilitando el acceso a todos los motores del robot.

3. **Dynamixel**: este componente se encarga de mover la cabeza y los dedos de las pinzas del robot. En total se encarga de controlar nueve motores o grados de libertad del robot.
4. **FaulHaber**: Este componente se encarga de mover los brazos del robot, en total diez motores o grados de libertad.



Graph 2: Grafo de componentes sobre el robot real

Con esta última descripción damos por zanjada toda la estructura del proyecto, en constante cambio para adaptarla a más usos y hacerla cada vez más robusta, así como su desarrollo y pruebas actuales, depurando errores y mejorando algunos conceptos aplicados a sus componentes y métodos.

7. Conclusiones finales

Resulta francamente complicado poner el broche final a un proyecto de esta envergadura y con estas características, debido principalmente a su complejidad y extensión.

Se han tocado todos (o me gustaría pensar que casi todos) los palos relacionados con la cinemática inversa. Desde el algoritmo original, se ha ido evolucionando, añadiendo nuevos casos de uso, hasta construir una estructura más completa, capaz de controlar muchos de los casos problemáticos a los que debe hacer frente la cinemática inversa.

Sin embargo, no todo está dicho, ni se puede dar completamente por finalizado el desarrollo de este proyecto. Aún quedan problemas que resolver y errores que depurar.

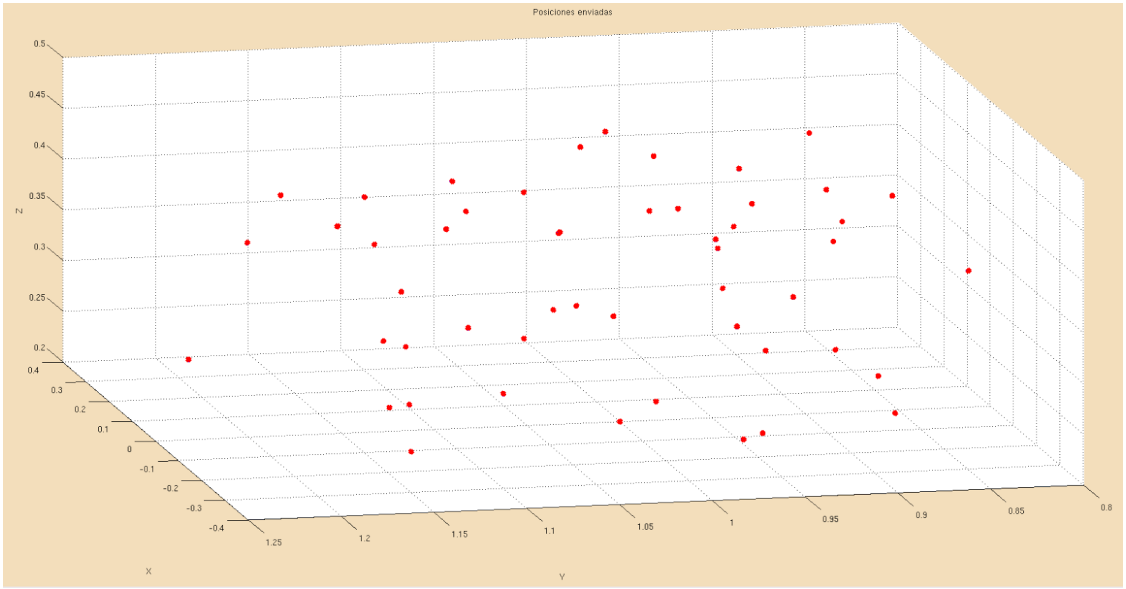
Es el caso de la ejecución de la cinemática inversa sobre el robot real. El algoritmo de Levenberg-Marquardt está pensado para trabajar sobre escenarios matemáticamente perfectos, sin límites ni restricciones, no está preparado para hacer frente a los errores típicos de trabajar en un mundo inexacto como es el real. Prueba de ello es el hecho de que, al ejecutar un mismo target sobre el robot simulado y el robot real, no se obtienen dos resultados iguales, muchas veces ni siquiera parecidos. Es necesario añadir al proyecto un enfoque de autocalibración, que sea capaz de corregir las holguras propias del robot real mediante la corrección de lo que el robot es capaz de percibir a través de sus sensores y lo que el algoritmo de Levenberg-Marquardt supone o calcula. Para ello se necesita retocar los métodos que calculan el jacobiano y la función de error para tener en cuenta ese error de cierre, no desde la muñeca, sino desde la cámara, además de añadir nuevos métodos que mejoren los resultados del *inverseKinematicsComp*. También el propio algoritmo de Levenberg-Marquardt puede ser mejorado, cambiando sus estructuras de control y haciéndolo más eficiente, computacionalmente hablando.

Por otra parte, la funcionalidad añadida para trabajar con motores prismáticos reales aún no ha podido ser probada, al carecer el laboratorio de este tipo de joints. Tampoco se ha podido probar con joints prismáticos simulados, para los que no hay un soporte aún programado.

También se está puliendo el movimiento de los motores del robot para que sea más uniforme y elegante, calculando trayectorias de aceleración y frenado que minimizen el impacto de los tiempos de espera tanto para la actualización del árbol cinemático como para el hardware del robot real.

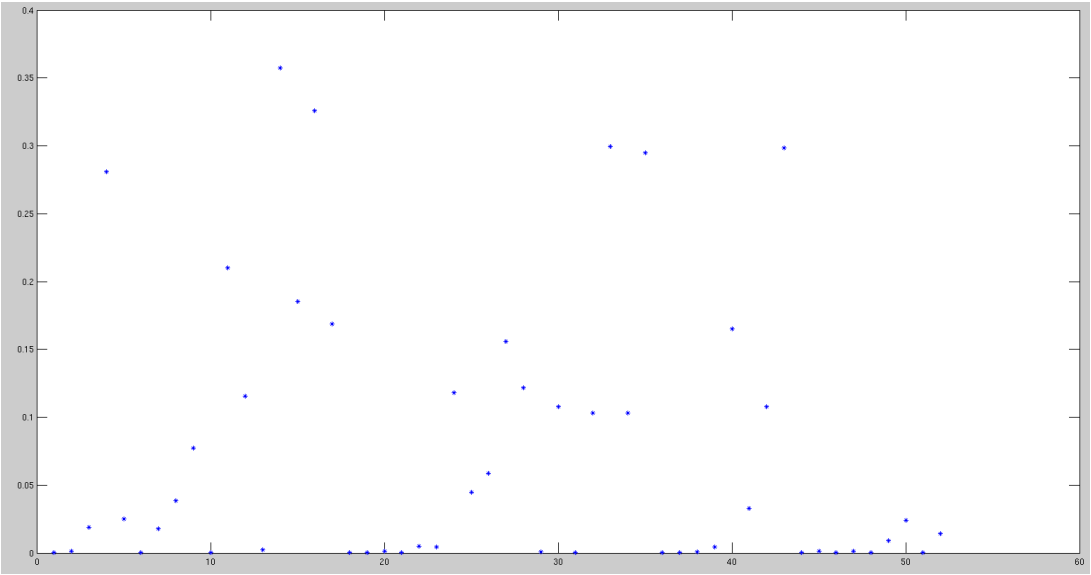
Por último, y como todo software, no tiene fecha de fin y las actualizaciones y mejoras son constantes, aunque parece ir por la senda correcta. Prueba de ello es el último ensayo realizado, esta vez sobre el robot real, Ursus 3.0, a cuyo brazo se le han asignado cincuenta poses destino, calculados de forma aleatoria (y sin repeticiones) sobre un cubo virtual situado en la zona de trabajo del robot, alcanzando el algoritmo de Levenberg-Marquardt el 100% de éxito con un error medio de 0.0751.

Esto podemos comprobarlo en las dos siguientes gráficas. En esta primera gráfica vemos los cincuenta targets a los que se envió el brazo derecho del robot Ursus:



Graphic 7: Posiciones destino o targets

En el segundo gráfico vemos marcados en azul los errores alcanzados en esas posiciones de destino. En ningún caso superan el umbral de 0.36 (esta es la norma de los errores d etraslación y de rotación por lo que la magnitud aún no está del todo clara):



Graphic 8: Errores alcanzados en cada target

Basándose en esta última prueba se desarrollará en el mes de Julio un juego interactivo con niños de entre siete y diez años, en el que el niño colocará una pelota ante el robot Ursus y éste tendrá que tocarla.

Con estos últimos datos damos por concluido este documento sobre la cinemática inversa en robot sociales, un documento bastante amplio y denso que esperamos resulte de utilidad.

Bibliografía

Robots sociales y de servicios:

1. Engelberger, J. F. (1989). *Robotics in Services*. Great Britain: Biddles Ltd, Guildford. ISB 0-262-05042-0
2. Ollero Baturone, A. Robótica. (2001). *Manipuladores y robots móviles*. Barcelona: Marcombo, S.A. ISBN 84-267-1313-0
3. Diccionario de Merriam-Webster: http://www.merriam-webster.com/
4. San Juan Terrones, T. Violante González, N. (2011). *Robots manipuladores y su importancia en la industria*. Ensayo. Universidad del Valle de México. Disponible en la dirección http://www.tlalpan.uvmnet.edu/oiid/download/Robots%20manipuladores_04_ING_IMECA_PII_E%20E.pdf
5. García, J. A. Vásquez, A. (¿?). *Los Robots en el Sector Agrícola*. Artículo para el Máster en Automática y Robótica de Universidad Politécnica de Madrid. Disponible en la siguiente dirección URL: http://www.disam.upm.es/~barrientos/Curso_Robots_Servicio/R_servicio/Agricultura_files/Robot%20en%20la%20Agricultura.pdf
6. Sass, L (2009). *Introducción a la Robótica: Robots Manipuladores*". Apuntes de la asignatura IME-440: Introducción a la robótica, impartida por Sass en el curso 2008-2009, disponibles en http://profesores.usfq.edu.ec/laurents/IME440.html
7. Moriello, S. (2008). *Robots Sociales, la nueva generación*. Artículo publicado el sábado 13 de Diciembre del 2008 en Tendencias21, disponible en la dirección: http://www.tendencias21.net/Robots-sociales-la-nueva-generacion_a2833.html

Problemas de cinemática directa e inversa:

1. Craig, J. J. (2006). *Robótica*. Pearson Education, Prentice Hall. 3º edición, ISBN 9702607728

2. Barinka, L. & Berka, R. (2002). *Inverse Kinematics – Basic Methods*. Dept. of Computer Science & Engineering. pp. 10. Disponible en la dirección URL: http://www.cescg.org/CESCG-2002/LBarinka/paper.pdf

3. Nilsson, R. (2009). *Inverse Kinematics*. Master's thesis. Lulea University of Technology. Disponible en la dirección URL: http://epubl.ltu.se/1402-1617/2009/142/LTU-EX-09142-SE.pdf

4. Lander, L. (1998). *Oh my God, I inverted kine*. Article of Game Developer (September of 1998). Disponible en la dirección URL: http://www.darwin3d.com/gamedev/articles/col0998.pdf

5. Manual de *prácticas de Robótica utilizando Matlab* del Grupo de Investigación NBIO de la Universidad Miguel Hernández (2012). Disponible en la dirección URL http://nbio.umh.es/files/2012/04/practica2.pdf

6. Manuales del Grupo AUROVA (Automática, Robótica y Visión Artificial) sobre la cinemática directa, forward kinematics (disponible en la dirección URL http://www.aurova.ua.es/robolab/EJS4/PRR_Suficiencia_Intro_2.html) y para cinemática inversa, inverse kinematics (disponible en la dirección URL: http://www.disclab.ua.es/robolab/EJS2/RRR_Intro_3.html).

7. Coro Conde, H. (2012). *Diseño e Implementación de un Robot SCARA de dos Grados de Libertad con Fines Didácticos*. Tesis de grado, Universidad Mayor de San Andrés. Disponible en la dirección: http://tesisdegradohectorc2.es.tl/Perfil-de-Tesis.htm

Teoremas y métodos utilizados:

1. Martínez de la Rosa, F. & Garrido Atienza, M. J. (1998). *Matemáticas II: Resúmenes teóricos y ejercicios*. Cádiz: Servicio de Publicaciones de la Universidad de Cádiz. ISBN 84-7786-517-5 (teorema de Taylor en pág. 31-33)

2. Hecker, C. (1997). *Physics, Part 4: The Third Dimension*. Article of Game Developer (June 1997). Disponible en la dirección URL: http://chrishecker.com/images/b/bb/Gdmphys4.pdf

3. Weisstein, E. W. (2003). *CRC Concise Encyclopedia of Mathematics*. Chapman & Hall/CRC, ISBN 1-58488-347-2. (Fórmula de Rodrigues pág. 2584)

4. Bauchau, O. A. (2011). *Flexible Multibody Dynamics*. Springer Dordrecht Heidelberg London New York. ISBN 978-94-007-0334-6 (Fórmula de Rodrígues para rotaciones en pág. 191)

5. Scales, L. E. (1985). *Introducction to Non-Linear Optimization*. London: Macmillan Publishers Ltd. ISBN 0-333-32552-4

6. Slabaugh, G. G. (1999) "*Computing Euler angles from a rotation matrix*". Journal Article of TRTA implementation. Disponible en la dirección URL: http://www.soi.city.ac.uk/~sbbh653/publications/euler.pdf

7. Gonźalez Álvarez, L. *Estudio e implementación amigable del método Gradiente Conjugado con el uso de precondicionadores*. Trabajo de Diploma, Universidad de la Habana. Trabajo publicado en www.ilustrados.com

8. Morales, J. L. (2009). *Análisis aplicado, dirección de descenso*. Apuntes. ITAM (Instituto Tecnológico Autónomo de México). Departamento de Matemáticas. 2009. Disponible en la dirección URL: http://users.eecs.northwestern.edu/~morales/PSfiles/AA_grad.pdf

9. Orin, David E. & Schrader, William W. (1984) *Efficient Computation of the Jacobian for Robot Manipulators*. Article of The International Journal of Robotics Research, volume 3, issue 4, pages 66-75. Disponible en la dirección URL: http://ijr.sagepub.com/content/3/4/66.short

10. Meredith, M. and Maddock, S. (2004). *Real-Time Inverse Kinematics: The Return of the Jacobian*. University of Sheffield. Pages 1-15. Disponible en la dirección http://www.dcs.shef.ac.uk/intranet/research/public/resmes/CS0406.pdf

11. Goldenberg, Andrew A., Benhabib, B. & Fenton, Robert G. (1985) *A Complete Generalized Solution to the Inverse Kinematics of Robots*. Article of IEEE Journal of Robotics an Automation, volume 1, issue 1, pages 14-20. Disponible en: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1086995&tag=1

12. Lourakis, M. I., Argyros, A. (2009). SBA: *A Software Package for Generic Sparse Bundle Adjustment*. Article of ACM Transactions on Mathematical Software, volume 36, issue 1, pages 1-30. Disponible en la dirección URL: http://doi.acm.org/10.1145/1486527

13. Ranganathan, A. (2004). *The Levenberg-Marquardt Algorithm*. Disponible en la dirección URL: http://www.cc.gatech.edu/~ananth/docs/lmtut.pdf

14. Tomomichi Sugihara. (2011). *Solvability-Unconcerned Inverse Kinematics by the Levenberg-Marquardt Method.* Article of IEEE Transactions on Robotics, volume 27, issue 5, pages 984-991. Disponible en la dirección URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5784347&tag=1

Componentes y software utilizados en el proyecto:

1. Software utilizado: http://robocomp.sourceforge.net/wordpress/

2. Manso, L., Bachiller, P., Bustos, P., Núñez, P., Cintas, R. and Calderita, L. (2010). *Robocomp: a Tool-based Robotics Framework*. Proceedings, SIMPAR Second International Conference on Simulation, Modeling and Programming for Autonomous Robots. ISBN 978-3-540-89075-1, pp 251-262. Disponible en pdf en la dirección URL: http://robolab.unex.es/index.php?option=com_remository&Itemid=53&func=startdown&id=82

3. Gutiérrez, M. A., Romero-Garcés, A., Bustos, P. and Martínez, J. (2012). *Progress in Robocomp*. Workshop en Agentes Físicos (WAF2012), Santiago de Compostela, Spain, 3-4 September 2012

4. Open Scene Graph: http://www.openscenegraph.org/

Otros conceptos:

1. Suárez Mejías, C., Echevarría C., Núñez, P. Bustos, P., Manso, J. L., Calderita, L. V., Leal, S. & Parra, C. (2012) *Ursus: a robotic assintant for training of children with motor impairments*. Book, Converging Clinical and Engineering Research on Neuro-rehabilitation, Springer series on BioSystems and BioRobotics, Editors, J.L Pons, D. Torricelli and Marta Pajaro. Springer, ISBN 978-3-642-

34545-6, pages 249-254. January 2012. Disponible en la dirección: http://robolab.unex.es/index.php?option=com_remository&Itemid=53&func=startdown&id=110

2. Mirantes, D. L. (2011). *Robots que dan vida*. Artículo publicado el 21/01/2011 en el Diario de León. Disponible en la dirección URL: http://www.diariodeleon.es/noticias/leon/robots-dan-vida_579190.html

3. Sosa Sosa, V. J. (2014). *MIDDLEWARE: Arquitectura para aplicaciones distribuidas*. Apuntes de Cinvestav-Tamaulipas. Disponible en la dirección URL http://www.tamps.cinvestav.mx/~vjsosa/clases/sd/Middleware_Recorrido.pdf

4. AprilTags. Documentación disponible en la dirección URL: http://april.eecs.umich.edu/wiki/index.php/AprilTags