Software de Control Desarrollado en ROS para Teleoperar de Manera Eficiente el Robot KUKA Lightweight mediante el Háptico Phantom Omni

Proyecto de Fin de Carrera
Facultad de Informática

12 de Julio de 2013

Jose Luis Outón Méndez

Supervisor Tecnalia:
Dr. Urko Esnaola

Supervisora UPV/EHU:
Dra. Elena Lazkano

Universidad del País Vasco
Euskal Herriko Unibertsitatea

tecnalia Inspiring Business

Agradecimientos

Me gustaría agradecer a la entidad Tecnalia la oportunidad que me ha brindado para desarrollar mi proyecto en un ambiente trabajador, cooperativo y con ansias de investigación.

Particularmente me gustaría agradecer a Urko Esnaola su confianza y dedicación que ha depositado sobre mí. A mi compañero Axel Harth, que a pesar de trabajar en proyectos diferentes hemos colaborado mucho juntos. A Damien Sallé y Mildred Puerto que con su perseverancia han sacado lo mejor de mí. Y por último, a Miguel Prada por su ayuda incondicional.

A mi directora de proyecto Elena Lazkano por su fantástica acogida y su dedicación durante el transcurso del proyecto.

No me gustaría terminar sin agradecer a mis familiares y en especial a mi madre por todo el apoyo que me han mostrado.

Resumen

La teleoperación o telerobótica es un campo de la robótica que se basa en el control remoto de robots esclavo por parte de un usuario encargado de gobernar, mediante un dispositivo maestro, la fuerza y movimiento del robot. Sobre dicho usuario recaen también las tareas de percepción del entorno, planificación y manipulación compleja.
Concretamente se pretende desarrollar el control software necesario para teleoperar un manipulador esclavo, *Kuka Lightweigh* mediante un dispositivo háptico *Phamton Omni,* que se comporta como maestro, sin que afecten las diferencias dinámicas y estructurales existentes entre ambos dispositivos, aportando información adicional al operador para facilitar la operación.
La principal motivación de la evolución de esta tecnología se debe a la necesidad de realizar trabajos en entornos hostiles, de difícil acceso, o perjudiciales para la salud del usuario.

Abstract

Teleoperation or telerobotics is an area of robotics concerned with the remote control of slave robots by a user who is responsible for governing, by a master device, the force and movement of the robot. Such user also performs tasks as environment perception, planning and complex manipulation.
Specifically, the aim is to develop the needed software control to teleoperate a slave manipulator, *Kuka Lightweigh* by *Phantom Omni* haptic device, which behaves as a master, without being affected by the dynamic and structural differences between the two devices and providing additional information to the operator to facilitate the operation.
This technology owes its evolution to the need of working at hostile, hard access environments, which can sometimes be harmful for user's health.

Palabras clave: **Teleoperación, cinemáticas, retroalimentación de fuerzas, Kuka LWR, Háptico Omni.**

Índice de contenido

IV DESARROLLO TÉCNICO

V VISIÓN GLOBAL

Índice de ilustraciones

Índice de tablas

Índice de Códigos

Parte I

INTRODUCCIÓN

Introducción

Desde tiempos remotos el ser humano siempre se ha ayudado de herramientas para aumentar el alcance de su capacidad de manipulación. Desde largos palos para alcanzar los frutos de los árboles, pasando por herramientas forjadas en hierro, hasta las grúas elevadoras del presente.
Estas herramientas se desarrollaron fruto del ingenio de personas que buscaban facilitar tareas que podían ser peligrosas, repetitivas o de difícil acceso.
Estos sistemas han evolucionado hasta lo que se conoce hoy en día como sistemas teleoperados maestro-esclavo. Donde los robots, frecuentemente manipuladores industriales y/o robots móviles, reproducen los movimientos guiados por un usuario, también conocido como usuario final o teleoperador.

Esta tecnología no se desarrolló con el fin de sustituir a las personas de sus actividades, si no para ayudarlas y asistirlas en trabajos con riesgo de accidentes o en condiciones perjudiciales para la salud, o incluso para manipular objetos de grandes dimensiones.
También es cierto que el hombre siempre ha sentido admiración por la forma del cuerpo humano y por sus capacidades, es por ello que con cada vez más frecuencia se desarrollan para estas tareas manipuladores antropomórficos y robots humanoides.
Los sistemas más evolucionados de teleoperación dotan al operador de realimentación sensorial del entorno, es decir, distancias, fuerzas, imágenes.
Actualmente hay una rama de la robótica que estudia la capacidad de programar robots/autómatas de manera sencilla. Esta necesidad ha surgido debido a la complejidad de programar tareas mediante código fuente. Se han acuñado diferentes términos anglosajones para este estudio, como pueden ser *Programing by demonstration,* y/o *Easy programing.*

Mediante teleoperación se puede enseñar tareas y trayectorias al robot para que tras ello éste sea capaz de repetirlas autónomamente.
El dispositivo o dispositivos que el usuario final utiliza en el proceso de teleoperación puede ser de diferente naturaleza. El hecho de utilizar un robot concreto no implica utilizar un dispositivo maestro determinado. Existen multitud de formas de controlar remotamente un robot: mediante sistemas hápticos, visión por cámaras, mandos a distancia genéricos (wiimote, PSMove, ...) o dedicados para el robot, robots a escala.
No es menester que haya una relación entre dispositivo maestro y esclavo en cuanto a forma o cinemática.
Esto se extiende incluso a los medios y protocolos de comunicación ya sean infrarrojos, bluetooth, cableado (ethernet, ethercat, firewire...), etc.

Parte II

OBJETIVOS

Y

PLANIFICACIÓN

1 Objetivos del proyecto

El proyecto se ha desarrollado en el centro de investigación Tecnalia, concretamente en la división de industria y transporte que se encarga de dar respuesta a las necesidades de movilidad sostenible y de fabricación eficiente en un entorno globalizado.

El objetivo que engloba todo el proyecto se basa en la investigación y creación de una respuesta de control software para realizar una correcta teleoperación del manipulador semi-industrial, *KUKA Lightweight* [Cap 3.7], en adelante denominado *Lwr*, mediante el dispositivo háptico *Phantom Omni* [Cap 3.6] de la compañía *SensAble*.

El objetivo global se divide en diversos sub-objetivos de menor tamaño conceptual y carga de trabajo separados entre sí cronológicamente.
Estos se presentan a continuación:

1. Realizar un estudio previo detallado de todas las tecnologías y componentes hardware y software utilizados en el proyecto.

2. Crear un modelo 3D del robot *Lwr* y de sus componentes con el fin de utilizarlo en un simulador. [Cap 4.3]

3. Instalar, manejar y visualizar el háptico *Omni* en el computador. [Cap 4.2]

4. Diseñar el código fuente necesario para realizar una primera aproximación de teleoperación sencilla entre *Lwr* y *Omni*. [Cap 4.4]

5. Diseñar dos modos diferentes de teleoperación que sean capaces de cubrir todas las necesidades del operador. Estos son el modo de ejecución relativo y el absoluto. [Cap 4.7]

6. El operador debe tener la capacidad de controlar la cantidad de movimiento que se realiza durante la teleoperación para obtener una mayor precisión en sus ejecuciones. [Cap 4.6]

7. Acoplar una garra robótica al *Lwr* de modo que se puedan realizar tareas de *grasping* durante la teleoperación. [Cap 4.9]

8. Realizar un control supervisado con el fin de aportar información extra al operador para que muestre una telepresencia máxima facilitando en gran medida el proceso de teleoperación. Para lograr una telepresencia máxima el control debe de estar dotado de las siguientes medidas: [Cap 4.12]

    a) El operador debe sentir en el *Omni* la retroalimentación o *feedback* de las fuerza resultante obtenida mediante la proyección de todas las fuerzas acaecidas en el manipulador sobre el último efector del *Lwr*.
    Para ello se utiliza los sensores de par que posee el manipulador en cada una de sus articulaciones.

    b) El operador debe sentir diferentes fuerzas en el *Omni* que le indiquen que se están produciendo desfases entre las posiciones deseadas por el operador y las que adopta *Lwr*. Nos referiremos a este control como, **Control de error en posición cartesiana**.

    c) El operador debe ser capaz de percibir e interpretar datos informativos del estado del manipulador entre los que se destacan la posición cartesiana y articular que posee en cada instante.

9. El manipulador debe ser capaz de detectar y evitar autónomamente posibles colisiones con un entorno establecido y con el propio manipulador.

10. El operador puede enseñar diferentes trayectorias al manipulador, las cuales se traducen en tareas que el manipulador debe reproducir autónomamente y sin variación con la original. Esta habilidad se denomina *Programing by demonstration*.

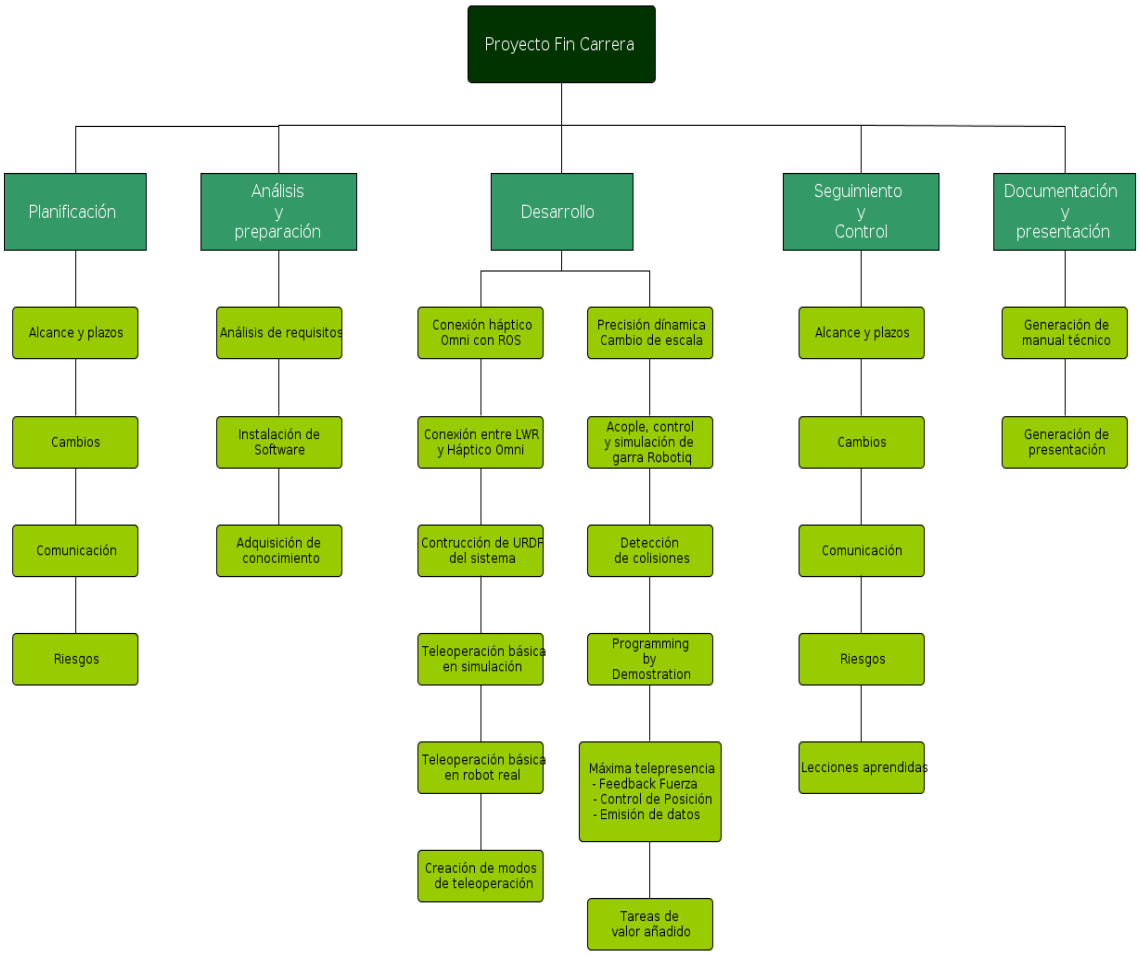
El objetivo último de la corporación Tecnalia es utilizar esta tecnología en manipuladores industriales e implantarlo en empresas dedicadas a diversos sectores de producción.

Como objetivo futuro se pretende aunar el trabajo desarrollado en este proyecto con el de otro compañero del departamento para utilizar teleoperación junto con cámaras *Microsoft Kinect* con el fin de conseguir una visión del espacio de trabajo del brazo y realizar un asistente de guiado de trayectorias. Este objetivo queda exento de los objetivos preliminares de este proyecto.

2 Planificación

2.1 Análisis del proyecto

El siguiente apartado muestra la relación existente entre el tiempo planificado y el real que se ha invertido en la realización de las diferentes tareas del proyecto.
Mediante el siguiente EDT (Estructura de descomposición del trabajo) se representan todas las tareas que han sido realizadas. A continuación se explican brevemente las más relevantes.



2.2 Alcance y plazos

El alcance del proyecto se basa en desarrollar el control software necesario para teleoperar de manera eficiente el robot *Lwr* mediante el dispositivo háptico *Omni*. Para ello se utilizan herramientas software como el *framework ROS* y *Orocos* sobre los cuales se hablará en los próximos capítulos.
Todo el desarrollo del proyecto se va a realizar bajo el sistema operativo Ubuntu, en su versión 10.04 LTS.
Se cuenta con una licencia académica de investigación para la utilización del háptico *Omni*.
La comunicación entre *Lwr* y *Omni* se realiza gracias a un computador de Tecnalia conectado mediante Ethernet al controlador del robot.
Tras la consecución de los primeros objetivos se va a acoplar a *Lwr* una garra industrial para realizar tareas de *grasping*.
Principalmente se van a realizar pruebas mediante la simulación del modelo del manipulador con posterior implantación en el robot real.
Se despreciarán los retardos de comunicación y de respuesta entre dispositivos, aunque serán conocidos y medidos.

Los plazos establecidos mediante hitos y estructurados por actividades se pueden observar en la Tabla 1. En ella se se aprecian las diferentes actividades que se han realizado a lo largo de todo el proyecto indicando por cada una de ellas su periodo de realización expresado en meses junto con las horas planificadas y reales.

Actividad	Periodo de realización	Horas planificadas	Horas reales
Planificación	Febrero	20	15
Análisis y preparación	Febrero	100	104
Desarrollo	Febrero - Marzo - Abril	160	190
Seguimiento y control	Febrero - Marzo - Abril	20	16
Documentación y Presentación	Marzo - Abril	70	75
TOTAL		370	400

Tabla 1: Tabla de plazos

2.3 Comunicación

Las reuniones se extienden a lo largo de todo el proyecto, si bien al principio son más bien escasas, una vez clarificadas las actividades a desempeñar se intensifican en su número. El mayor volumen de reuniones se dan en la etapa de desarrollo.

Las tareas de comunicación han sido más costosas de lo que en un primer momento se esperaba, ya que el responsable directo del proyecto por parte de Tecnalia, Urko Esnaola, ha tenido que realizar un viaje a Japón por un periodo de 6 meses debido a cuestiones internas del centro. Esta situación no ha repercutido directamente en ningún objetivo del proyecto, no obstante, la forma de comunicación ha variado considerablemente.
Las reuniones de seguimiento [Cap 11.1], que en un primer momento han sido presenciales y mediante el uso de correos electrónicos internos, han pasado a ser semanales y realizadas mediante videoconferencias entre España y Japón, utilizando para ello la herramienta *Google+* debido a su fiabilidad y estabilidad.
Para compartir el código fuente generado en el proyecto se ha utilizado el asistente *subversion,* el cual simplifica en gran modo el mantenimiento y depuración del código.

2.4 Cambios

Durante todo el proyecto se han producido numerosos cambios en diversos factores del proyecto. Algunos de ellos han sido necesarios debido a las circunstancias que se han dado a lo largo del desarrollo, en cambio, otros se han dado de modo inesperado. A continuación se muestran una serie de cambios ordenados por su nivel de repercusión:

- Poca repercusión
    - Cambios en la localización de los recursos del proyecto. Durante diferentes etapas se han producido cambios de posición de mi lugar de trabajo y de los recursos robóticos.

- Media repercusión
    - Cambios de los recursos del proyecto. Debido a que diferentes proyectos trabajan conjuntamente con el mismo manipulador se han alternado, en muchas ocasiones, los efectores finales, siendo estos la garra robótica industrial (*Robotiq*) y la mano robótica de precisión *Prensilia*.

- Gran repercusión
    - Cambios en versiones software. Debido a la necesidad de actualizar el proyecto a las versiones últimas de software se han producido cambios en las versiones de ROS y de Ubuntu.

2.5 Riesgos

En las siguientes tablas se muestran los posibles riesgos que pueden acaecer a lo largo del desarrollo del proyecto, poniendo éste en peligro. Para su correcto estudio se describen, se mide su impacto y la probabilidad de que ocurran. Para cada uno de ellos se adjunta un plan de contingencia en caso de que se de dicha situación, con el fin de mitigar los sus efectos.

Riesgo	Ordenador de desarrollo deje de funcionar
Descripción	El ordenador que se utiliza para el desarrollo del proyecto se estropee
Probabilidad	Baja
Impacto	Alto
Consecuencias	Perdida de información y de datos del proyecto
Contingencia	Habilitar otro ordenador de las mismas características para continuar con el desarrollo del proyecto. Realizar copias de seguridad de todo el proceso software como de la documentación generada

Riesgo	Robot o componente robótico deje de funcionar
Descripción	Alguna parte física del proyecto (robot, garra, háptico) se estropee
Probabilidad	Media
Impacto	Alto
Consecuencias	No se podría continuar con la fase de desarrollo del proyecto
Contingencia	Llamar al servicio técnico del recurso estropeado e intentar repararlo o sustituirlo con la mayor brevedad posible. Mientras tanto, intentar continuar la fase de desarrollo utilizando el simulador

Riesgo	Responsable de proyecto abandone las instalaciones de Tecnalia por un periodo prolongado
Descripción	El responsable del proyecto deba realizar una estancia en otro país durante un tiempo prolongado por cuestiones internas de Tecnalia
Probabilidad	Alta
Impacto	Medio
Consecuencias	Las comunicaciones, seguimiento y control del proyecto se verían afectadas. Ralentizarían el desarrollo del proyecto

Contingencia	Realizar videoconferencias todas las semanas. Comunicación mediante correo electrónico. Compartir código fuente con software de control de versiones.

Riesgo	Perdida de documentación
Descripción	Perdida total o parcial de la documentación generada a lo largo de toda la vida del proyecto
Probabilidad	Baja
Impacto	Alto
Consecuencias	Volver a redactar toda la documentación importante a tener en cuenta en posteriores etapas del proyecto: documentación de cambios y justificación de decisiones
Contingencia	Se realizarán copias de seguridad de toda la documentación tanto en *Dropbox* como en el ordenador personal

Riesgo	Descalibración del háptico *Omni*
Descripción	Por diversos motivos el háptico *Omni* puede llegar a descalibrarse durante la fase de desarrollo del proyecto
Probabilidad	Alta
Impacto	Medio
Consecuencias	Imposibilidad de continuar con la fase de desarrollo ya que es un recurso imprescindible para el proyecto
Contingencia	Contactar con el servicio técnico de *SensAble* e intentar volver a calibrar de forma manual el háptico. De no ser posible, enviarlo a los fabricantes para su recalibrado y puesta en marcha

Riesgo	Número elevado de Demos con los robots
Descripción	Cuantiosas demos dentro y fuera de las instalaciones de Tecnalia
Probabilidad	Alta
Impacto	Medio
Consecuencias	Durante el periodo de Demos se imposibilita la continuación del desarrollo
Contingencia	Procurar continuar con las fase de desarrollo utilizando la simulación del robot y sin hacer demasiado cambios en el código fuente.

2.6 Análisis y Preparación

Esta es una parte fundamental del proyecto ya que en ella se definen los requisitos técnicos necesarios para afrontarlo y se analiza el proyecto en su plenitud.

Se pretende teleoperar, mediante un dispositivo háptico, un manipulador industrial *KUKA Lightweight.* Para ello es necesario conocer los diferentes tipos de hápticos que se pueden obtener en el mercado por un precio no muy elevado.
Se ha optado por el dispositivo háptivo *Phantom Omni* de la compañía *SensAble* debido a diversos factores que se indicarán en la sección de fundamentos tecnológicos y a los consejos de los expertos de Tecnalia. Un valor añadido es su compatibilidad con diferentes sistemas operativos.

La conexión entre manipulador y háptico se realiza mediante las conexiones *Firewire* en el caso de la conexión entre el *Omni* y el computador y Ethernet para el caso de la conexión entre el computador y el controlador del manipulador.

El sistema operativo utilizado es Ubuntu en su versión 10,04 LTS.

Como IDE para la implementación se emplea el editor de texto *Gedit* debido a su agilidad y simplicidad.

Como referentes en la programación de software se utilizarán los lenguajes *C++* y *Python* utilizándolos indistintamente según indique el diseño del programa.

A lo largo de todo el proyecto se utilizan herramientas que ayudan y facilitan la generación de código software, como es el caso de ROS [Cap 9,1]*, Robot Operating System,* que se utiliza como *framework* de desarrollo ya que ofrece herramientas y librerías para el desarrollo de sistemas robóticos, u OROCOS [Cap 3.2.3] que facilita los cálculos de cinemáticas.

El computador utilizado para calcular cinemáticas y otros valores posee las siguientes características:

Marca: Dell
Procesador: Intell CORE 2 DUO, 6600 @ 2,40GHz
Memoria RAM: 2GiB

Parte III

FUNDAMENTOS TECNOLÓGICOS

3 Tecnologías empleadas

En el siguiente capítulo se va a hacer referencia a las diferentes tecnologías utilizadas en el desarrollo del proyecto. Desde los lenguajes de programación utilizados hasta las librerías de cálculo de cinemáticas, pasando por las características del sistema maestro esclavo, manipulador y háptico.

3.1 Lenguajes de programación

Para el desarrollo del proyecto se han utilizado dos lenguajes de programación muy comunes en el campo de la robótica, *Python y C++*. A pesar de que ambos son muy potentes y adecuados para este tipo de programación se ha optado por seleccionar a *Python* como lenguaje predeterminado por diversos motivos:

Se trata de un lenguaje interpretado o de *script*, multiplataforma, con tipado dinámico y fuertemente tipado, multiparadigma y orientado a objetos. La sintaxis es muy limpia y favorece a la legibilidad del código. Se ahorra mucho tiempo a la hora de ejecutar el código ya que no se a de compilar.

Por otra parte, el uso de *C++* ha sido necesario debido a que gran cantidad del código existente en *ROS* y *Orocos* esta implementado en dicho lenguaje.

3.2 Herramientas utilizadas

3.2.1 Interprete de comandos

Al trabajar sobre Ubuntu y ROS es necesario familiarizarse con la terminal del sistema ya que todo se maneja mediante uso de comandos. Esto facilita el movimiento entre directorios, ejecución de programas, cambio de permisos, mostrar variables del sistema, etc.

Debido a que es necesario utilizar diferentes comandos Linux y/o ROS [Cap 9.1] simultáneamente y observar los numerosos resultados se ha optado por utilizar una terminal más sofisticada que la que trae por defecto la distribución de Ubuntu.
Esta nuevo interprete de comandos tiene el nombre de *Terminator* y se encuentra en los repositorios de Ubuntu.
Como se puede observar *Terminator* permite dividir una ventana de comandos en numerosas sub-ventanas de diferentes tamaños y posiciones.
Esto hace que sea mucho más sencillo contrastar datos, lanzar varias aplicaciones a la vez y capturar resultados. Además permite la creación de nuevas pestañas, en las cuales también incluir sub-ventanas. Por todo ello se ha convertido en una herramienta indispensable el manejo de comandos en ROS/Linux.

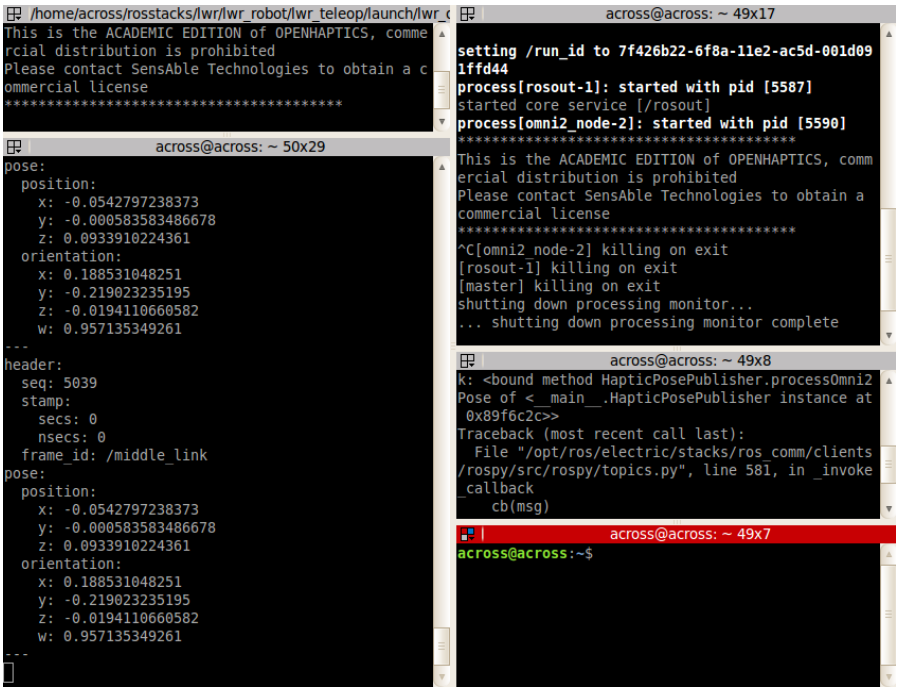


Ilustración 1: Interprete de comandos Terminator

3.2.2 Entorno de desarrollo integrado

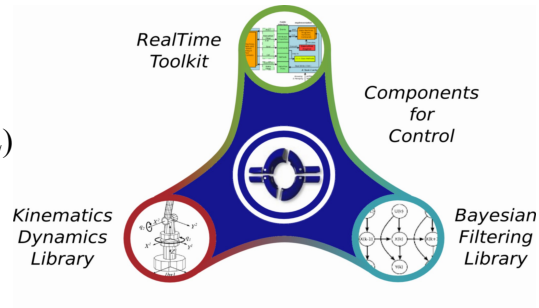
Existe gran cantidad de entornos de desarrollo (IDE) compatibles con Ubuntu. Genéricos, dedicados a lenguajes específicos, más o menos pesados, con una interfaz amigable, etc.
Dentro de este gran abanico de posibilidades se ha elegido uno adaptado a nuestras necesidades, *Gedit Text Editor*.
Con mayor similitud a un editor de texto que a un entorno de desarrollo, *gedit* ha sido la decisión última ya que es un entorno muy ágil, simple y dinámico. Esta herramienta que

parece muy simple, si se añaden los *plugins* adecuados, se convierte en un IDE muy potente y con el que la tarea de programar se simplifica en gran medida. No obstante, su hándicap es que no dispone de un *debugger* para depurar el código y solucionar errores, por lo que en muchas ocasiones la solución más rápida es colocar trazas a lo largo del código.

3.2.3 Orocos

Orocos es el acrónimo de *Open Robot Control Software.* Su objetivo principal es desarrollar software libre de propósito general que se basa en un *framework* para el control de dispositivos robóticos.
Soporta cuatro bibliotecas desarrolladas en C++:

- Real-Time Toolkit (RTT)
- Kinematics and Dynamics Library (KDL)
- Bayesian Filtering Library (BFL)
- Orocos Component Library (OCL)



La librería *Kinematics and Dynamics Library (KDL)* integrada en ROS [Cap 10.1.4] contiene clases y métodos implementados en su mayoría en C++, no obstante, también existen partes escritas en *Phyton*. Mediante la utilización de la librería somos capaces de calcular las soluciones al problema cinemático directo e inverso. Estas soluciones se calculan de manera iterativa mediante el método de obtención de raíces *Newton-Fourier /Rapson.*
Los métodos software utilizados son *CarToJnt()* para calcular la cinemática inversa y *JntToCart()* para calcular la directa. En la Ilustración 2 se puede observar un ejemplo de la *API* de *KDL* donde se aprecia el método *CartToJnt()*.

int KDL::ChainIkSolverPos_NR_JL::CartToJnt (const JntArray & q_init,
const Frame & p_in,
JntArray & q_out
) [virtual]

Calculate inverse position kinematics, from cartesian coordinates to joint coordinates.

Parameters:
q_init initial guess of the joint coordinates
p_in input cartesian coordinates
q_out output joint coordinates

Returns:
if < 0 something went wrong

Implements **KDL::ChainIkSolverPos**.

Ilustración 2: API KDL, método CartToJnt → Cinemática inversa

El proceso para calcular la cinemática inversa se compone de diferentes pasos. En primer lugar se debe conocer la configuración *q* del manipulador.
Llamamos *q* a la posición articular del manipulador en el instante actual. En otras palabras, la posición articular se refiere a los ángulos formados por cada articulación del brazo robótico a partir de un eje de referencia expresado en radianes. Estos ángulos se obtienen directamente de los sensores, también conocidos como *encoders*, del manipulador.

En el Código 1 se puede observar el pseudocódigo utilizado para calcular la cinemática inversa. Como primer paso se establece un máximo de iteraciones para que el algoritmo intente calcular una solución cinemática, ya que de no ser así, el algoritmo podría no converger nunca e interrumpir el funcionamiento del sistema.

La posición cartesiana del manipulador indica el lugar en el espacio cartesiano donde se encuentra el último eslabón o efector final del manipulador expresado en coordenadas cartesianas *x, y, z.*
El problema cinemático directo en posición es capaz de calcular la posición cartesiana del manipulador a partir de una posición articular. Representado en el pseudocódigo como *pos_fk()*, éste se calcula directamente componiendo las matrices de transformación [Cap 7.1] desde la base. Las matrices que definen los parámetros del manipulador se pueden establecer mediante el método *Denavit Hanterberg* (DH), pero en la práctica no siempre es así ya que los parámetros *DH* no son suficientemente generales para representar cualquier configuración, *p.ej.* no manejan bien cadenas que se ramifican, como un torso con varios brazos.

Existe un error de posición *delta_x* entre la posición cartesiana actual del manipulador calculada mediante la cinemática directa, y la posición deseada. Si el error es menor que una tolerancia definida previamente, la solución se considera válida.

Entrada: posición articular actual del manipulador y posición cartesiana deseada.
Salida: posición articular del manipulador hallada acorde a la posición deseada, *x_des*.

```
q = q_ini
   for i = 0 to max_iter
      x = pos_fk(q)
      delta_x = diff(x, x_des)
      if delta_x ~= 0
          break
      delta_q = vel_ik(delta_x, q)
      q = q + delta_q
return q
```

Código 1: Código fuente cinemática inversa

La cinemática inversa en velocidad, lo que en el pseudocódigo aparece como *vel_ik(),* se calcula utilizando la matriz jacobiana del manipulador. Esta matriz posee las derivadas parciales de cada coordenada cartesiana respecto a cada coordenada articular y depende de la posición articular. Con esta matriz se puede escribir la cinemática directa en velocidad.

En la práctica no se suele utilizar la inversa tal cual, ya que es muy común que *J* sea haga singular en algún valor de *q*, esto es, tenga determinante igual a 0, y por tanto no exista su inversa. Existen diferentes soluciones para esto, y todas suelen implicar una u otra forma de pseudo-inversa, que tenga solución definida en las singularidades.

En la Ilustración 3 se puede observar la relación, mediante jacobianas, entre la velocidad articular del manipulador y la velocidad cartesiana.

Directa: $dx/dt = J(q) * dq/dt$ Inversa: $dq/dt = inv(J(q)) * dx/dt$

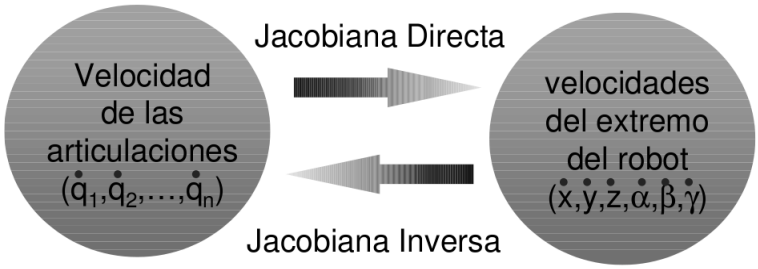


Ilustración 3: Relación entre Jacobianas

3.3 Cinemática Inversa

El problema cinemático inverso consiste en encontrar los valores que deben adoptar las coordenadas articulares del robot q = [q1, q2, . . . , qn] para que su extremo se posicione y oriente según una determinada localización espacial.

3.3.1 Definición

Un brazo articulado consta de un conjunto de segmentos rígidos unidos mediante diferentes articulaciones [Cap 8.1]. Los múltiples ángulos que pueden adoptar dichas articulaciones permiten un gran número de configuraciones para el brazo. La cinemática inversa tiene en cuenta el tipo de articulación y el número de ellas para encontrar una posición angular para cada una de ellas y, por tanto, posicionar el efector final en la posición cartesiana deseada.
Un problema de cinemática inversa puede no tener solución, tenerla, o tener muchas. La existencia o no de la solución lo define el espacio de trabajo del robot. La ausencia de una solución significa que el robot no puede alcanzar la posición y orientación deseada porque se encuentra fuera del espacio de trabajo del robot o fuera de los rangos permisibles de cada una de sus articulaciones.

El caso de tener más de una solución se da en robots redundantes, los cuales poseen más grados de libertad de los necesarios para posicionar y orientar el efector final en el espacio. Dicha característica les permite alcanzar una posición y una orientación determinada con diferentes configuraciones articulares [Ilustración 4]. Esto es necesario cuando el entorno es cambiante, con muchos obstáculos o de difícil acceso.

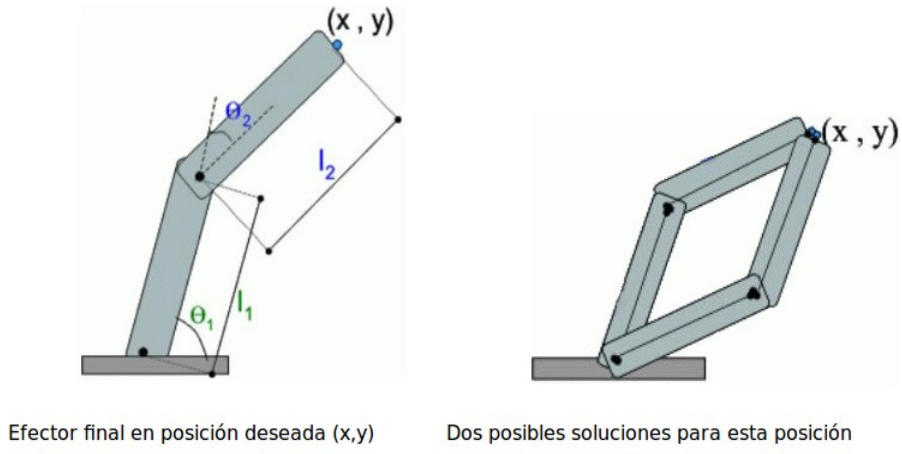


Ilustración 4: Solución del problema cinemático inverso para un robot planar con dos articulaciones. Puede tener 0,1 o 2 soluciones

3.3.2 Métodos de resolución

La obtención de soluciones puede realizarse de manera iterativa mediante el método de obtención de raíces *Newton-Fourier/Rapson*, éste tiene el inconveniente que sus cálculos son lentos. En aplicaciones reales el número de iteraciones o el tiempo máximo es acotado para reducir dicha lentitud.

A su vez existen soluciones analíticas mediante el uso de métodos geométricos. Consisten en la utilización de las relaciones trigonométricas y la resolución de los triángulos formados por los elementos y articulaciones del robot.
Existen otros métodos de resolución de la cinemática inversa:

1. Métodos geométricos

   - Mediante el uso de relaciones geométricas y trigonométricas (resolución de triángulos). Se obtienen parámetros del manipulador mediante la medida de la longitud de cada eslabón y el ángulo que forman entre ellos. Los parámetros del siguiente ejemplo corresponden con los parámetros de la Ilustración 4.

$$x = l_1 cos(\theta_1) + l_2 cos(\theta_1 + \theta_2)$$

$$y = l_1 sin(\theta_1) + l_2 sin(\theta_1 + \theta_2)$$

$$x^2 + y^2 = l_1^2 + l_2^2 + 2 l_1 l_2 cos(\theta_2)$$

$$cos(\theta_2) = \frac{x^2 + y^2 - l_1^2 - l_2^2}{2 l_1 l_2}$$

Ilustración 5: Relaciones geométricas

   - Método de reducción polinómica. Se basa en la transformación de las ecuaciones obtenidas por métodos algebraicos o geométricos para que adopten forma polinómica, la cual es más fácil de resolver.

2. Resolución a partir de las matrices de transformación homogénea

   - *Denavit y Hartenberg* propusieron un método sistemático para descubrir y representar la geometría espacial de los elementos de una cadena cinemática con respecto a un sistema de referencia fijo. Este método utiliza una matriz de transformación homogénea para descubrir la relación espacial entre dos elementos rígidos adyacentes.

Siendo A_x las matrices de transformación de cada eslabón del manipulador, T_y la matriz de transformación para toda la cadena cinemática y C_i, S_i, seno y coseno, respectivamente.

$$A_1 = \begin{bmatrix} C_1 & -S_1 & 0 & a_1C_1 \\ S_1 & C_1 & 0 & a_1S_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_2 = \begin{bmatrix} C_2 & -S_2 & 0 & a_2C_2 \\ S_2 & C_2 & 0 & a_2S_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^0 = A_1^0 A_2^1$$

$$T_2^0 = \begin{bmatrix} C_{12} & -S_{12} & 0 & a_1C_1 + a_2C_{12} \\ S_{12} & C_{12} & 0 & a_1S_1 + a_2S_{12} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ilustración 6: Matriz de transformación homogénea

3. Desacoplamiento cinético

   - Mediante la separación de la orientación y el posicionamiento del último efector del manipulador, se puede resolver de forma explícita los 3 grados de libertad que definen la orientación del efector final. Existe una solución analítica por métodos algebraicos mediante el *Método de Pieper*.

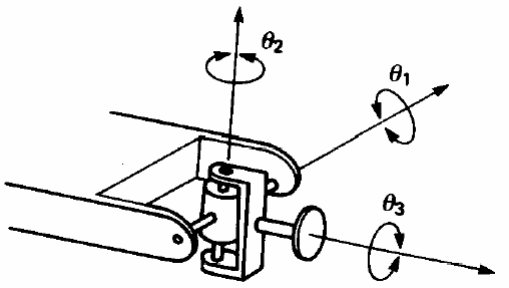


Ilustración 7: Posición y orientación del efector final

4. Otros métodos de resolución:

   - Álgebra de tornillo, métodos iterativos...

3.3.3 Observaciones

En muchas ocasiones, para acelerar los cálculos se emplean tablas calculadas previamente, conocidas como *look-up tables*.

El coste de calcular *n* soluciones no es necesariamente *n* veces el de calcular una única solución ya que existen configuraciones singulares del manipulador que son más difíciles de calcular sus cinemáticas, hasta el punto de no existir soluciones o que el tiempo de cálculo supera un umbral previamente definido. Por ello son preferibles las soluciones cerradas explicitas que las iterativas.

3.3.4 Ejemplo práctico

En el siguiente ejemplo se va a mostrar, paso a paso, cómo calcular la cinemática inversa de un manipulador planar con tres grados de libertad dibujada en la Ilustración 8.

En primer lugar se calculan las matrices de transformación del manipulador, las cuales se representan mediante la notación ${}^{j}_{i}A$ y corresponden a los eslabones comprendidos entre el *i* y el *j*.

$$\mathbf{A}_1^F = \begin{bmatrix} cos\theta_1 & -sin\theta_1 & 0 \\ sin\theta_1 & cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}_2^1 = \begin{bmatrix} cos\theta_2 & -sin\theta_2 & d_1 \\ sin\theta_2 & cos\theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}_3^2 = \begin{bmatrix} cos\theta_3 & -sin\theta_3 & d_2 \\ sin\theta_3 & cos\theta_3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}_P^3 = \begin{bmatrix} 1 & 0 & d_3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

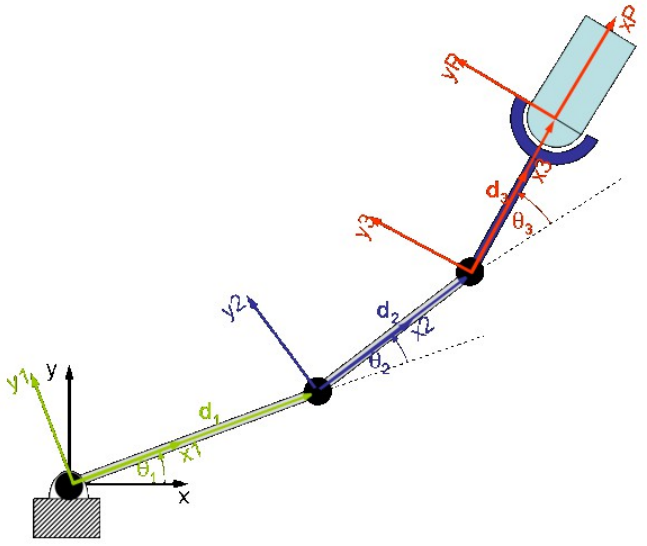


Ilustración 8: Matrices de transformación de robot planar con 3 grados de libertad

La matriz de transformación para toda la cadena cinemática, es el producto de las matrices individuales.

$$\mathbf{T}_P^F = \mathbf{A}_1^F \mathbf{A}_2^1 \mathbf{A}_3^2 \mathbf{A}_P^3$$

$$\mathbf{T}_P^F = \begin{bmatrix} c(\theta_1+\theta_2+\theta_3) & -s(\theta_1+\theta_2+\theta_3) & d_3c(\theta_1+\theta_2+\theta_3)+d_2c(\theta_1+\theta_2)+d_1c\theta_1 \\ s(\theta_1+\theta_2+\theta_3) & c(\theta_1+\theta_2+\theta_3) & d_3s(\theta_1+\theta_2+\theta_3)+d_2s(\theta_1+\theta_2)+d_1s\theta_1 \\ 0 & 0 & 1 \end{bmatrix}$$

A partir de la matriz de transformación anterior, obtenemos:

$$\begin{aligned} \theta &= (\theta_1+\theta_2+\theta_3) \\ \mathsf{x} &= d_3 cos(\theta_1+\theta_2+\theta_3)+d_2 cos(\theta_1+\theta_2)+d_1 cos\theta_1 \\ \mathsf{y} &= d_3 sin(\theta_1+\theta_2+\theta_3)+d_2 sin(\theta_1+\theta_2)+d_1 sin\theta_1 \end{aligned}$$

Para obtener los valores de Θ_1, Θ_2, y Θ_3 para los valores de posición cartesianos dados x, y, Θ, bastaría con:

$$\begin{aligned} \theta_2 &= arccos\left[\frac{(\mathsf{x}-d_3c\theta)^2+(\mathsf{y}-d_3x\theta)^2-d_2^2-d_1^2}{2d_1d_2}\right] \\ \theta_1 &= arccos\left[\frac{(\mathsf{x}-d_3c\theta)(d_1+d_2c\theta_2)+(\mathsf{y}-d_3x\theta)(d_2s\theta_2)}{d_1^2+d_2^2+2d_1d_2c\theta_2}\right] \\ \theta_3 &= \theta-(\theta_1+\theta_2) \end{aligned}$$

3.4 Cinemática Directa

El problema cinemático directo consiste en calcular la posición y orientación de partes de una estructura articulada a partir de sus componentes fijas y las transformaciones inducidas por las articulaciones de la estructura.

3.4.1 Definición

Determinar la posición y orientación del extremo final, *end-effector*, del robot en el espacio cartesiano, con respecto a un sistema de coordenadas de referencia, conocidos los ángulos de las articulaciones y los parámetros geométricos de los elementos del robot .

3.4.2 Métodos de resolución

Para construir el modelo directo de un determinado robot existen dos alternativas:

Utilización de relaciones específicas del robot, lo que consiste en el empleo de las relaciones geométricas que puedan establecerse en el robot de que se trate.

Método general, el cual trata de emplear matrices de transformación que relacionan sistemas de referencia. A continuación se muestran los pasos que se deben seguir para la resolución del problema cinemático inverso mediante el método general:

- Establecer para cada eslabón del robot un sistema de coordenadas cartesiano ortonormal (x_i,y_i,z_i) donde $i = 1,2,...,n$ siendo *n* el número de grados de libertad del robot. Cada sistema de coordenadas corresponderá a la articulación (i+1) y estará fijo en el elemento i-ésimo.

- Encontrar los parámetros *Denavit-Hartenberg* [Cap 7.1] de cada una de las articulaciones.

- Calcular las matrices A_i de proyección de la posición del elemento i al (i+1)

- Calcular la matriz $T^0_n = A^0_1\ A^1_2\ ...\ A_n^{n-1}$ la posición del primer eslabón a la posición del eslabón final.

3.4.3 Ejemplo práctico

En el siguiente ejemplo se muestra, paso a paso, cómo calcular la cinemática directa de un manipulador SCARA serie con cuatro grados de libertad [Ilustración 9].

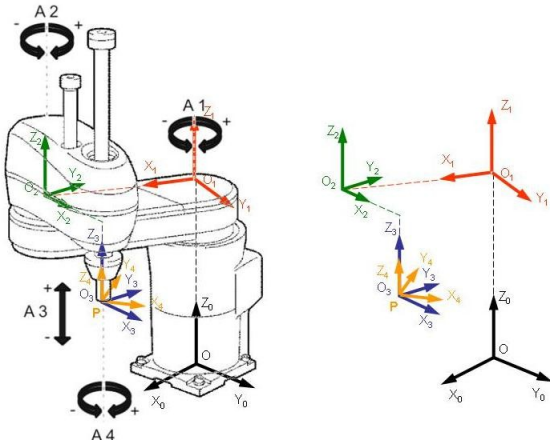


Ilustración 9: Robot SCARA serie con cuatro grados de libertad

En primer lugar se calculan las matrices de transformación del manipulador:

$$A_1^0 = \begin{bmatrix} \cos\alpha_1 & -\sin\alpha_1 & 0 & 0 \\ \sin\alpha_1 & \cos\alpha_1 & 0 & 0 \\ 0 & 0 & 1 & L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_3^2 = \begin{bmatrix} 1 & 0 & 0 & L_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -a_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_2^1 = \begin{bmatrix} \cos\alpha_2 & -\sin\alpha_2 & 0 & L_2 \\ \sin\alpha_2 & \cos\alpha_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_4^3 = \begin{bmatrix} \cos\alpha_4 & -\sin\alpha_4 & 0 & 0 \\ \sin\alpha_4 & \cos\alpha_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Calcular la matriz de transformación para toda la cadena cinemática:

$$T_4^0 = A_1^0 A_2^1 A_3^2 A_4^3$$

$$T_4^0 = \begin{bmatrix} \cos\theta_Z & -\sin\theta_Z & 0 & X \\ \sin\theta_Z & \cos\theta_Z & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A partir de la matriz de transformación anterior, obtenemos la cinemática directa, esto es, la posición cartesiana del último efector en el espacio.

$$
\begin{aligned}
X &= L_2 \cos\alpha_1 + L_3 \cos(\alpha_1 + \alpha_2) \\
Y &= L_2 \sin\alpha_1 + L_3 \sin(\alpha_1 + \alpha_2) \\
Z &= L_1 - a_3 \\
\theta_Z &= (\alpha_1 + \alpha_2 + \alpha_4)
\end{aligned}
$$

3.5 Dispositivo Háptico

Los dispositivos hápticos proporcionan realimentación de fuerza al sujeto que interactúa con entornos virtuales o remotos. Tales dispositivos trasladan una sensación de presencia al operador que se denomina telepresencia y es necesaria para que el operador pueda desarrollar correctamente sus funciones.
Esta tecnología abarca multitud de sectores en los que se puede aplicar y facilitar su ejecución como: medicina, realidad virtual y aumentada, modelado 3D, robótica, fuerzas armadas, etc.
Por otra parte, se encuentra la figura del teleoperador, operadores que controlan el dispositivo de manera remota y que necesitan sentir las fuerzas de contacto resistentes.

Diferentes estudios han clarificado cómo un individuo ha de explorar un objeto para que éste pueda ser identificado y diferenciado de otros objetos. Para ello se han descrito ocho técnicas comunes para la percepción de los objetos:

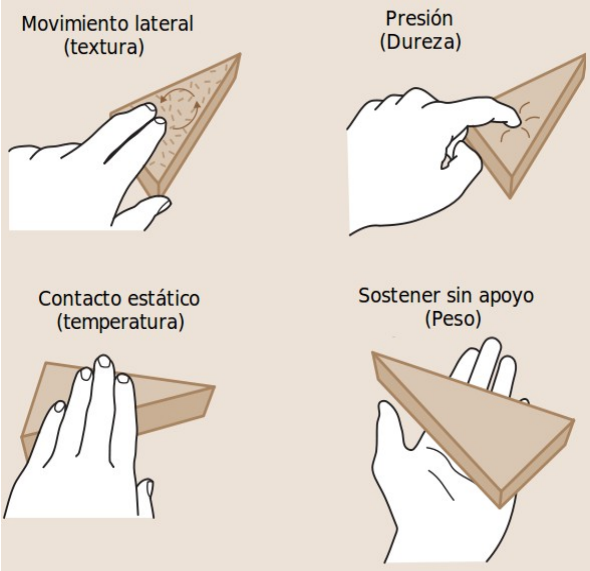


Ilustración 10: Cuatro de las ocho formas de exploración de objetos

1. Movimiento lateral (textura).
2. Presión (dureza).
3. Contacto estático (temperatura).
4. Sostener sin apoyo (peso).
5. Recinto (forma global, volumen).
6. Seguimiento de contornos (forma exacta, volumen).
7. Prueba de partes móviles (partes móviles).
8. Función de pruebas (función específica).

Es necesario que toda o parte de esta información sea enviada de alguna manera al teleoperador mediante fuerzas de contacto resistentes, lo que denominamos telepresencia.

El proceso de teleoperación en entornos virtuales se realiza siguiendo los siguientes siete pasos [Ilustración 11]:

1. Desplazamientos articulares Θ del dispositivo. El operador mueve el háptico alrededor de su espacio de trabajo. Cada posición adquirida por los eslabones del háptico es recogida en forma de posición articular mediante los sensores de movimiento o *encoders.* Estos expresan sus mediciones en forma de ángulos representados en radianes.

2. Procesados mediante cinemática directa. Las posiciones articulares anteriormente obtenidas son utilizadas para hallar la cinemática directa, esto es, la posición cartesiana del eslabón final del háptico, también conocida como el *tip* del *stylus*. Se obtiene, por tanto, la posición y velocidad cartesiana $x, \dot{x}$ respectivamente.

3. Detección de colisiones. Para este caso, el software debe determinar si la posición del punto de interfaz háptica (PIH) o *tip,* se encuentra en contacto con un objeto virtual en el instante actual del tiempo. En la práctica, esto se utiliza generalmente para determinar si el PIH está penetrando la superficie del objeto.

4. Determinación de puntos de la superficie. La superficie del objeto se representa mediante un modelo geométrico que puede componerse de polígonos o *splines*.

5. Cálculo de fuerzas de contacto. Las fuerzas de contacto se calculan comúnmente mediante la **ley de Hooke** expresada del siguiente modo:

   $$f = k \cdot x$$

   Donde x es el vector que representa la posición del PIH y k es un valor constante mayor que cero. Cuando k es suficientemente grande, la superficie del objeto se sentirá como una pared perpendicular a x. Este muro virtual, o superficie de impedancia, es un elemento fundamental para la mayoría de los entornos virtuales hápticos. Debido a que el muro virtual sólo se muestra si existe una colisión entre el PIH y el objeto virtual, el muro es una restricción unilateral, gobernada por una condición no lineal.
   El valor máximo de k esta limitado por la fuerza máxima que puede proporcionar el dispositivo.

   Para mejorar la sensación de rigidez se añade un amortiguador al sistema, que además, mejora la estabilidad y puede simular fricción con el medio. Este nuevo componente añadido, se refiere a la velocidad con la que se esta desplazando el PIH en el momento de su medición.

   $$f = k \cdot x + d \cdot v_{PIH}$$

6. Cinemáticas. La fuerza calculada en el espacio cartesiano debe ser transformada en momento de torsión o *torque* en el espacio del actuador.
   Típicamente, el cálculo corresponde con la siguiente fórmula:

$$\tau = J^{\top} f$$

   Donde τ es el comando de par para los actuadores, f es el vector de fuerza deseada, y $J^{\top}$ es la transpuesta de la matriz jacobiana del dispositivo háptico.
   En situaciones ideales donde los actuadores son perfectos, el usuario final puede sentir las fuerzas deseadas exactas, pero debido a retardos, a la dinámica del dispositivo y a otros factores, las fuerzas percibidas por el usuario difieren de las deseadas.

7. Amplificación y actuación sobre dispositivo. Se amplifican las fuerzas calculadas y se envían en forma de señales eléctricas al dispositivo háptico para que se encargue de transformar dichas señales en fuerzas que el operador es capaz de sentir y reconocer.

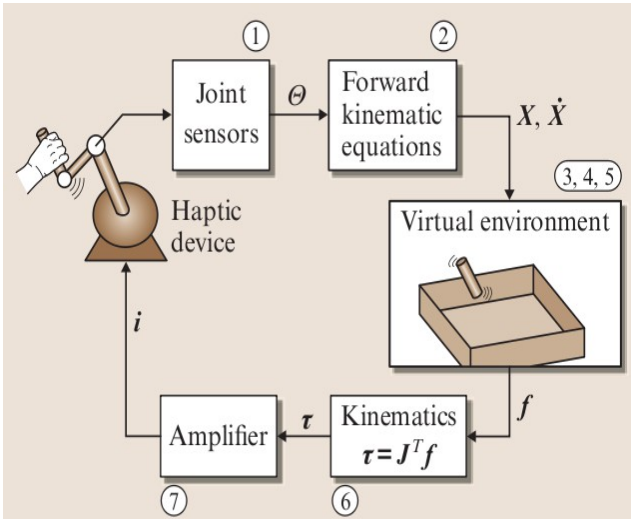


Ilustración 11: Telepresencia en entornos virtuales

A la hora de manipular un dispositivo háptico existen dos tipos de control, de impedancia y de admitancia:

- **Impedancia**: el usuario mueve el dispositivo, y el dispositivo reaccionará con una fuerza si es necesario.

    - Entrada: desplazamiento
    - Salida: fuerza

    El rango de impedancia ideal será:

    - Cero en movimiento libre
    - Infinito cuando hay contacto

- **Admitancia**: el dispositivo mide las fuerzas que el usuario ejerce sobre él, y reacciona con el movimiento (la aceleración, la velocidad, la posición).

    - Entrada: fuerza
    - Salida: desplazamiento

El caso más común, el de impedancia, sigue los siguientes criterios para el control:

- Los sensores de desplazamiento recogen el movimiento generado.
- Se envía este desplazamiento, mediante controlador, al simulador software.
- Se produce la interacción con el mundo virtual y se calculan las fuerzas.
- Se envía señal al controlador.
- El dispositivo produce la fuerza necesaria.
- El háptico realiza la fuerza necesaria para ello.

3.6 Phantom Omni

El háptico *Phantom Omni* del fabricante *SensAble* que se muestra en la Ilustración 12, es un dispositivo capaz de tocar y sentir objetos virtuales con una gran precisión. Posee un precio moderado para sus capacidades y características técnicas [Tabla 2].

Ilustración 12: Háptico Phantom Omni - SensAble

Espacio de trabajo de realimentación de fuerza	~6.4 an. x 4.8 al. x 2.8 prof. in > 160 an. x 120 al. x 70 prof. mm
Huella (Área que ocupa la base)	6 5/8 an. x 8 prof. in ~168 an. x 203 prof. mm
Peso	3 lbs 15 oz
Rango de movimiento	Movimiento de la mano giratoria en la muñeca
Resolución de la posición nominal	> 450 dpi ~ 0.055 mm
Fricción de retroceso	< 1 oz (0.26 N)
Fuerza máxima de posición nominal (brazo ortogonal)	0.75 lbf (3.3 N)
Fuerza continua (24 horas)	> 0.2 lbf (0.88 N)
Rigidez	X eje > 7.3 lbs / en (1.26 N / mm) Y eje > 13.4 lbs / en (2.31 N / mm) Z eje > 5.9 lbs / en (1.02 N / mm)
Inercia (masa aparente en la punta)	~0.101 lbm (45 g)
Realimentación de fuerzas	x, y, z
Sensores de posición [Stylus cardán]	x, y, z (encoders digitales) [Pitch, roll, yaw (± 5% potenciómetros lineales)
Interfaz	IEEE-1394 FireWire® port: 6-pin to 6-pin
Compatible con OpenHaptics®Toolkit	Sí

Tabla 2: Características técnicas del dispositivo háptico Phantom Omni

Al trabajar con dispositivos hápticos, es importante distinguir entre el espacio de trabajo nominal y el real [Ilustración 13]:

Espacio de trabajo nominal: es el volumen en el cual el fabricante del dispositivo garantiza la precisión y realimentación de fuerza especificada en la hoja de características. En el caso del *Phantom*, se trata de un prisma rectangular de dimensiones (160 ancho, 120 alto, 70 profundidad en mm).

Espacio de trabajo real: se trata del volumen que es posible recorrer con el extremo del manipulador. Pero este espacio real incluye una zona marginal donde el comportamiento del dispositivo puede ser inaceptable para algunas aplicaciones, además de inestable.

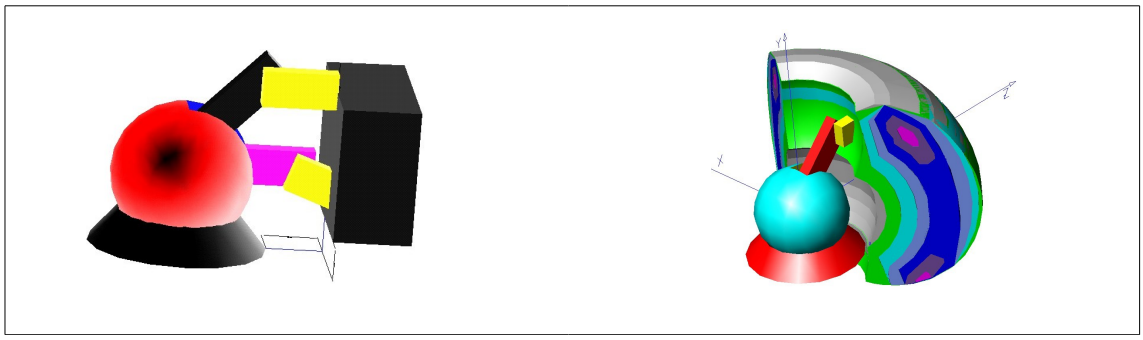


Ilustración 13: Espacio nominal y real del Phantom Omni

Phantom Omni es un dispositivo de retroalimentación de fuerza de propósito general, con control de impedancia, esto es, el usuario mueve el dispositivo, y el dispositivo reacciona con diferentes fuerzas.
Por su morfología lo podemos catalogar como tipo lápiz ya que posee un *stylus* acoplado en su extremo. Dicho *stylus* posee dos botones con los que se pueden desempeñar diferentes tareas. Mecánicamente posee 6 grados de libertad.

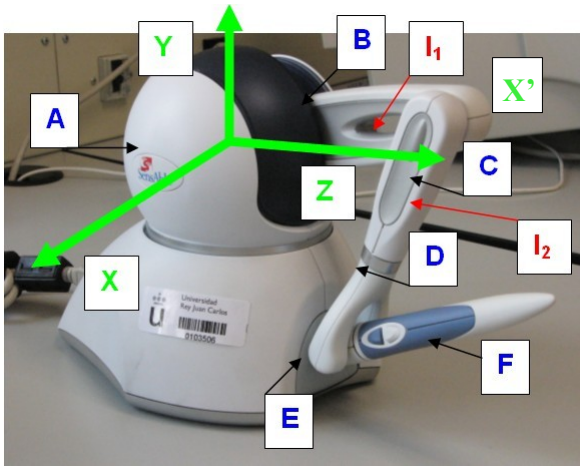


Ilustración 14: Desglose de eslabones y rotaciones del Omni

A gira alrededor del eje **Y** definiendo el ángulo $\boldsymbol{\theta_1}$.

B gira alrededor del eje **X** definiendo el ángulo $\boldsymbol{\theta_2}$.

C gira alrededor del eje **X'** relativo definiendo el ángulo $\boldsymbol{\theta_3}$.

D, **E** y **F** son ejes ortogonales situados en el elemento final y son ángulos Gimbal.

Una de las características más importantes de los hápticos con retroalimentación es cómo el operador percibe las fuerzas y la información táctil enviada. Para lograr una correcta sensación táctil y una telepresencia adecuada se debe cumplir:

- El rango de impedancia ideal debe ser cero cuando el movimiento está libre de contactos e infinito cuando se detecta un contacto o colisión.

- La información percibida de manera visual prevalece sobre la háptica y corrige las limitaciones de los dispositivos.

- Para lograr sensación de continuidad durante la teleopertación, la frecuencia de muestreo con la que el háptico debe trabajar debe ser de 1000Hz. Esta frecuencia es necesaria para percibir las fuerzas de forma continua, máxime cuando se va a interactuar con sólidos rígidos y en menor medida cuando los objetos son deformables.
  El problema de alcanzar estas frecuencias tan altas de muestreo es que limitan la cantidad de cálculos que se pueden realizar.

3.7 KUKA LightWeight

El robot *KUKA LightWeight*, es el resultado de una colaboración de investigación bilateral entre *KUKA Roboter GmbH* y el Instituto de Robótica y Mecatrónica del Centro Aeroespacial Alemán (DLR).

Proporciona muchas características únicas para los investigadores robóticos. Para dar pleno acceso a estas características, se ha desarrollado una nueva interfaz que permite el acceso a bajo nivel directamente al controlador del robot (KRC), a altas velocidades de hasta 1 kHz y en tiempo real. Por otro lado, proporciona características de innovación industrial, como la capacidad de aprendizaje, los sensores de par en cada articulación, un lenguaje de *scripting* integrado KRL (*KUKA Robot Language*), además de emplear la tecnología estándar *socket UDP* para la comunicación.

3.7.1 Características

El *hardware* del controlador se ha desarrollado de manera que la fuente de alimentación, la tarjeta del controlador y la lógica de seguridad se encuentran en una caja común,lo que se conoce como *all-in-a-box*.
Principalmente, el *hardware* se compone de tres partes, señaladas en la Ilustración 15:

Ilustración 15: KUKA LightWeight

1. Manipulador robótico *KUKA LightWeight*. (*Lwr*)

2. Panel de Control KUKA, conocido como *teach pendant* (KCP)

3. Controlador KUKA Robot (KRC)

El manipulador es un brazo articulado de 7 ejes. Cada articulación está equipada con un sensor de posición en la entrada y sensores de par y posición en el lado de salida. El robot, por tanto, puede ser operado mediante posición, velocidad y control de par.

En la Ilustración 16 se pueden observar todas las rotaciones que es capaz de realizar el manipulador, teniendo éste 7 grados de libertad, uno por cada rotación, representadas mediante una vocal (A, E) y un número de identificación (1...6).
Debido a sus numerosos grados de libertad, podemos denominar al manipulador como redundante.

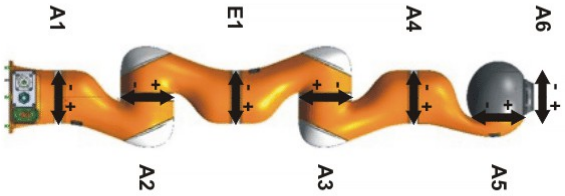


Ilustración 16: Ejes del manipulador Lwr

Los fabricantes de *Lwr* han desarrollado una nueva interfaz completamente integrada en el controlador del manipulador (KRC), para que investigadores y desarrolladores puedan desarrollar código e interactuar con el robot de manera más sencilla. Esta interfaz que recibe el nombre de *Fast Research Interface* (FRI), está basada en el sencillo protocolo de comunicación UDP. La interfaz permite al usuario controlar el estado del robot desde un PC externo.
Cuando se establece una conexión, la velocidad de muestreo de la interfaz puede ser seleccionada libremente entre 1 y 100 ms.

Debido a que las instrucciones de desplazamiento se publican con mayor lentitud que los ciclos de mili-segundos, éstas son preprocesadas e interpoladas por la unidad de control KUKA. Debido a que se utiliza Ethernet UDP como tecnología de conexión, la interfaz puede ser portada a una amplia variedad de sistemas operativos y computadores.

En la Ilustración 17 se puede observar de manera global cómo se establecen las comunicaciones y relaciones entre los componentes del *Lwr*, y cómo FRI ayuda a las comunicaciones con el controlador desde un PC ajeno al sistema KUKA.
En la parte superior izquierda de la ilustración se observa el KCP que corresponde con el *teach pendant*, o mando a distancia con pantalla informativa de cristal liquido donde se observa la interfaz gráfica (GUI) del sistema. KCP conecta con el controlador y se puede programar tareas mediante el lenguaje dedicado KRL, que es interpretado y ejecutado por el controlador. Otro posible modo de programar tareas es mediante código fuente LUA o KRL y FRI que se encarga de enviar el dicho código al interprete de *Lwr* mediante UDP.

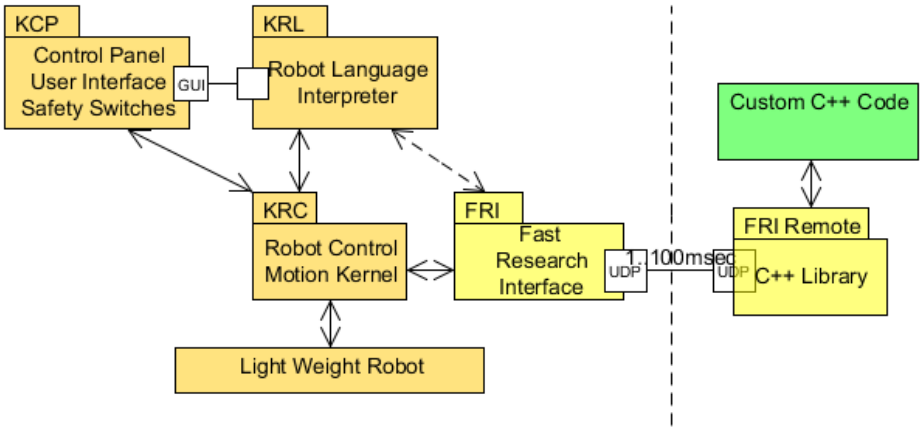


Ilustración 17: Visión general del la arquitectura del sistema de control FRI

El robot se caracteriza principalmente por la capacidad que posee para trabajar con y para humanos. Esto se debe a su morfología y a su novedosa tecnología. Se puede observar que el manipulador está diseñado con formas redondeadas evitando, por tanto, posibles lesiones a personas que interaccionen con el robot y puedan ser golpeadas por éste.

En el caso de recibir un impacto por parte del robot, éste posee sensores de par en cada una de sus articulaciones. Gracias a esta tecnología el robot es capaz de "sentir" el impacto y reaccionar de tal manera que el impacto recibido sea mucho menor, llegando incluso a parar.

Otra de las bondades del manipulador es su capacidad para compensar la gravedad. Cuando se activa esta opción, el robot es capaz de mantener una posición establecida sin necesidad de la utilización de los frenos mecánicos. Esto lo realiza enviando el par gravitacional al controlador de par y realizando cálculos para que compense dicho par. Esto es así para cada una de las articulaciones.
En el caso de aplicarle una fuerza externa, por ejemplo, la fuerza de la mano de un usuario, el robot seguirá la trayectoria que se le indique. Tras finalizar el guiado, el robot mantendrá la última posición establecida.
Gracias a esta tecnología, se hace muy sencillo enseñar trayectorias al robot, que es capaz de aprender y repetir con una gran exactitud.

Por todo ello este manipulador ha sido seleccionado por los expertos de Tecnalia para el departamento de I+D+i e incluirlo en proyectos dedicados a la interacción persona-robot, también conocido como colaboración segura.
Las características técnicas principales del manipulador *Lwr* se describen en las tablas 3 y 4:

Número de ejes	7
Volumen del espacio de trabajo	1.84 m^3
Repetitividad	±0.05 mm
Peso	aprox. 16 kg
Nivel de sonido	<75 dB (A) fuera del marco de trabajo
Posición de montaje	Suelo, techo, pared
Acabado de la superficie, pintura	Aluminio Tipo: plata, Pintura: naranja, base marco: naranja

Tabla 3: Características técnicas del manipulador Lwr

Ejes	Rango de movimiento	Velocidad sin carga	Par máximo
A1 (J1)	+/-170°	112.5 °/s	200 Nm
A2 (J2)	+/-120°	112.5 °/s	200 Nm
E1 (J3)	+/-170°	112.5 °/s	100 Nm
A3 (J4)	+/-120°	112.5 °/s	100 Nm
A4 (J5)	+/-170°	180.0 °/s	100 Nm
A5 (J6)	+/-120°	112.5 °/s	30 Nm
A6 (J7)	+/-170°	112.5 °/s	30 Nm

Tabla 4: Características de los ejes del manipulador Lwr

En cada una de sus articulaciones el manipulador posee un freno mecánico que entra en acción ante una parada repentina del robot como medida de seguridad, o en el momento de apagar el robot ya que los servos no reciben el voltaje necesario para mantener la ultima posición adoptada.

En caso de parada o de que alguna de las articulaciones supere su amplitud máxima será necesario desbloquear los frenos utilizando el cable que aparece en la Ilustración 18. Mediante este cable se envía una señal eléctrica al freno para que se desbloquee y se pueda, por tanto, mover la articulación en cuestión. No se trata de una tarea fácil ya que para acceder al cable es necesario desmontar la carcasa del codo del manipulador.

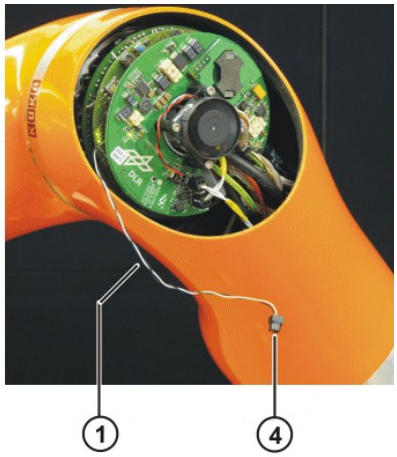


Ilustración 18: Interior manipulador

Si se desea realizar un análisis más exhaustivo del manipulador se adjunta, a este trabajo, la guía de usuario "*KUKA System Software 5.6 lr*", que incluye toda la información suministrada por el fabricante.

3.8 Librerías Reflexxes

Las librerías de movimiento *Reflexxes* proporcionan la capacidad de generar, mediante diferentes algoritmos, trayectorias instantáneas para sistemas de control de movimiento. Genera trayectorias suaves y continuas de manera determinista dentro de menos de una milésima de segundo, por lo que los robots y máquinas pueden reaccionar a las señales de los sensores y otros eventos instantáneamente durante cualquier movimiento.
Poseen diferentes algoritmos encargados de generar trayectorias en linea. Las tres principales características de estas librerías son:

1. Los movimientos del robot se calculan a partir de los estados iniciales arbitrarios de movimiento (ej. durante cualquier movimiento).

2. Los nuevos movimientos se calculan dentro de un ciclo de control de bajo nivel (normalmente en menos de 1 ms).

3. La interfaz es muy simple y clara, de tal manera que se puede integrar fácilmente en los sistemas existentes.

Dados dos puntos, *Reflexxes* es capaz de realizar una interpolación para generar una trayectoria que se enviará al robot. Esto lo realiza de manera segura respetando tanto los límites físicos del robot, como la velocidad y la aceleración máxima de cada articulación.

3.9 Garras robóticas

Se han empleado dos tipos de garras robóticas, una de carácter industrial y otra más sofisticada para usos más específicos.
La primera mano está fabricada por la compañía *Robotiq*, consta de tres dedos adaptativos, los cuales son capaces de coger cualquier tipo de objeto ya que sus dedos se acomodan al objeto en cuestión. La conexión con el computador se realiza mediante *EtherCAT*. Es óptima para procesos industriales y de automatización.

La segunda se trata de una mano antropomórfica de la compañía *Prensilia*, la cual puede ser utilizada en robots humanoides, en prótesis de manos, para experimentos neurocientíficos, etc. Se comunica mediante puerto serie, RS232. Es óptima para investigación.

Ilustración 19: Garras robóticas

Parte IV

DESARROLLO TÉCNICO

E

IMPLEMENTACIÓN

4 Desarrollo e implementación

El proyecto comienza debido a la adquisición del háptico *Phantom Omni* por parte de Tecnalia y a las diferentes propuestas de teleoperación que se han dado en proyectos internos. Tecnalia, a su vez, posee un manipulador semi-industrial *Kuka Lwr* con el que se están desarrollando otros proyectos y con el que es interesante realizar estudios de teleoperación. Por ello, surge la idea de teleoperar el manipulador *Lwr* mediante el dispositivo háptico *Phantom Omni*.

4.1 ROS y Phantom Omni

Se propone la utilización del *framework ROS* [Cap 9.1] como herramienta software principal, ya que ofrece diferentes herramientas y librerías para el desarrollo de sistemas robóticos.
Como primera aproximación cabe indicar que ROS es un sistema multiplataforma, no obstante, en este proyecto se ha instalado sobre el sistema operativo Ubuntu 10.04 ya que para este sistema está más depurado y optimizado.
La idea principal de ROS es la comunicación de diferentes programas, también conocidos como nodos, entre sí para generar estructuras de nodos más complejas que ejecuten numerosas tareas. Los programas pueden estar escritos en diferentes lenguajes, siendo los más usuales *Python* y *C++* por sus ya demostradas cualidades a la hora de programar tareas robóticas y debido a su robustez y eficiencia.
La comunicación entre dichos nodos se realiza mediante paso de mensajes o como se conoce en ROS, *topics*. En un *topic* se puede leer o escribir información que previamente ha sido depositada por otros nodos. Puede haber varios publicadores y suscriptores sobre el mismo *topic* concurrentemente y un único nodo puede publicar y suscribirse a múltiples *topics*.

Además de las mencionadas existen otras herramientas que asisten al programador en su tarea, tales como la herramienta de visualización y simulación *Rviz*, que muestra modelos virtuales de robots incluyendo los ejes de coordenadas que toma como referencia en el tiempo. Dichos ejes toman el nombre de *TF* y están representados por x, y, z con los colores rojo,verde y azul respectivamente [Ilustración 20].

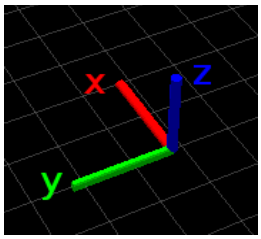


Ilustración 20: Ejemplo de TF visualizado en rviz

Por otra parte, se deben instalar los controladores del *Phantom Omni* [Cap 10.1.3]. Una vez realizado se han instalado automáticamente los directorios y paquetes necesarios para su uso, por lo que ya se adaptado el háptico en ROS.
Dentro del paquete Phantom_omni definimos y adaptamos el nodo *omni.cpp* a nuestras necesidades. Este nodo se encarga, entre otras cosas, de controlar la ejecución del *Omni* a la vez de obtener información necesaria para publicar su estado en diferentes *topics*. Entre ellos podemos destacar el *topic omni2_pose*, donde se publica la posición y orientación de la punta del *stylus*, y el *topic omni2_button*, donde se publica si alguno de los botones del *stylus* ha sido pulsado.

Tras ello, el objetivo es tener la capacidad de visualizar en *rviz* los ejes de referencia del *Omni*, concretamente el eje que representa la punta del bolígrafo *stylus* que se encuentra en el último eslabón del mismo, basándonos para ello en sus ejes de coordenadas o *TFs*.
Es necesario realizar transformaciones en los ejes de coordenadas entre la base del *Omni* y la punta del *stylus,* que es el *TF* que nos interesa.
En primer lugar se crean los *TF* que se van a utilizar, los cuales corresponden con los grados de libertad del *Omni*. Cada uno de ellos se coloca en el espacio siguiendo la geometría del dispositivo real.
Como el *Omni* posee seis grados de libertad se crean seis *TFs* de manera obligatoria, no obstante, se crean dos más ya que se necesita generar la base estática del *Omni* y la proyección de todos los *TF* hasta llegar al último que se corresponde con la punta del *stylus* al que llamaremos L0_6 [Ilustración 21].

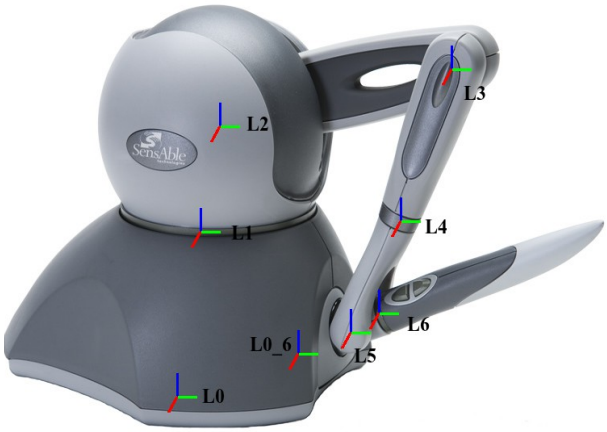


Ilustración 21: Representación de los TF en el Omni

En el siguiente código fuente [Código 2] escrito en C++ se puede observar cómo se crean los *TF* y se realizan las transformaciones necesarias entre ellos siguiendo la geometría del modelo real.

```
tf::Transform l0, l1, l2, l3, l4, l5, l6, l0_6;
// Indica el origen en X,Y,Z
l0.setOrigin(tf::Vector3(0., 0, 0.15));
// Indica la rotación en row, pich, yaw
l0.setRotation(tf::createQuaternionFromRPY(0, 0, 0));

l1.setOrigin(tf::Vector3(0., 0, 0.));
l1.setRotation(tf::createQuaternionFromRPY(0, 0, -state->thetas[1]));

l2.setOrigin(tf::Vector3(0., 0, 0.));
l2.setRotation(tf::createQuaternionFromRPY(0, state->thetas[2], 0));
...
(hasta l0_6)
```

Código 2: Creación de TF del Phantom Omni

Cada uno de los *TF* se genera respecto al anterior, obteniendo una estructura de árbol.
Una vez definidos todos, es momento de publicarlos para que puedan ser visualizados en el espacio y tiempo por *rviz*.
En la función de publicación es necesario incluir el nombre del *TF*, el tiempo del sistema y el *TF* padre e hijo.

```
br.sendTransform(tf::StampedTransform(l0,ros::Time::now(),
omni_name.c_str(), link_names[0].c_str()));

l0_6 = l0 * l1 * l2 * l3 * l4 * l5 * l6;

br.sendTransform(tf::StampedTransform(l0_6,ros::Time::now(),
link_names[0].c_str(), link_names[6].c_str()));
```

Código 3: Publicación de los TF para que rviz pueda visualizarlos

En la primera linea del Código 3, se ha publicado el *TF* de la base del *Omni* que es estático en el tiempo. En la segunda se obtiene la proyección de todos los *TF* sobre el último y se publica de igual manera. En este caso, el padre *link_names[0]* es la propia base del *Omni*.

En la Ilustración 22 se puede observar los *TF* del *Omni* representados en el espacio tiempo por la herramienta *rviz*. El *TF* con nombre */omni2_link6* corresponde con la punta del *stylus* y se observa como se mueve ésta a lo largo del tiempo. En la ilustración tan solo se presentan tres de los 8 *TF* generados ya que son los más representativos.

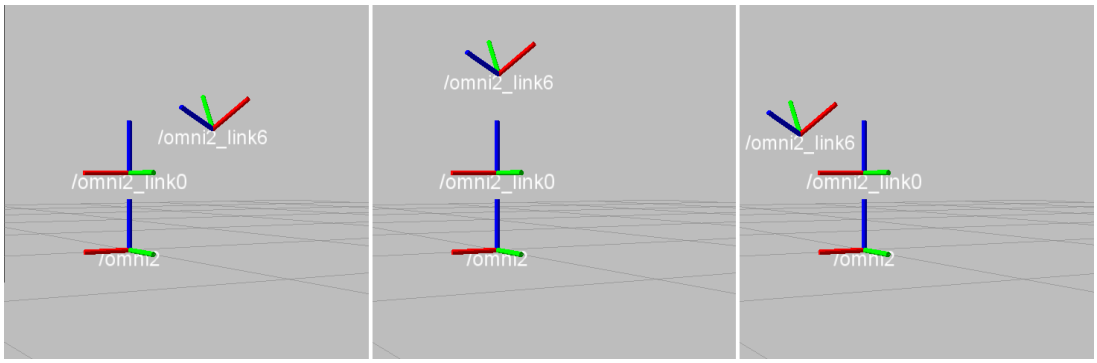


Ilustración 22: TFs del Phantom Omni. omni2_link6 corresponde a la punta del stylus

4.2 Modelos URDF

Una vez que podemos visualizar el *Omni* en el espacio y conocer su posición y orientación, es el momento de crear el modelo del manipular KUKA *Lwr* y de su lugar de anclaje.

Para la generación de modelos se suelen utilizar programas destinados a tal fin como son *Maya* o *Blender*. No obstante, ROS incluye una manera de crear e interpretar modelos que recibe el nombre de *Unified Robot Description Format*, *URDF*.
La sintaxis del código fuente para describir el modelo del robot es muy similar a la utilizada en *XML*. El paquete *URDF* contiene un analizador escrito en *C++* que interpreta el código escrito.
Mediante las opciones del paquete *URDF* se pueden definir diversos tipos básicos de eslabones, esto incluye esferas, cubos, cilindros y mallas. Utilizando esos eslabones básicos y unidos mediante articulaciones se puede generar un modelo rudo del robot.

Como se puede observar en la Ilustración 23, un *joint* o articulación enlaza dos eslabones, uno como padre y otro como hijo. Estos eslabones pueden tomar las diferentes formas indicadas arriba. Las articulaciones también pueden ser muy diferentes y ajustarse a cada robot especifico, (revolución, prismática, fija, planar...) [Cap 8.1].

Los eslabones también incluyen diferentes propiedades, entre ellas la geometría, el tipo de material, la textura, las propiedades inerciales y las colisiones. Mediante la etiqueta *collision* se puede sobreponer al eslabón una figura geométrica, normalmente un rectángulo, que lo envuelva de manera transparente para el usuario. Ésta se encarga de detectar colisiones alrededor de todo el eslabón.

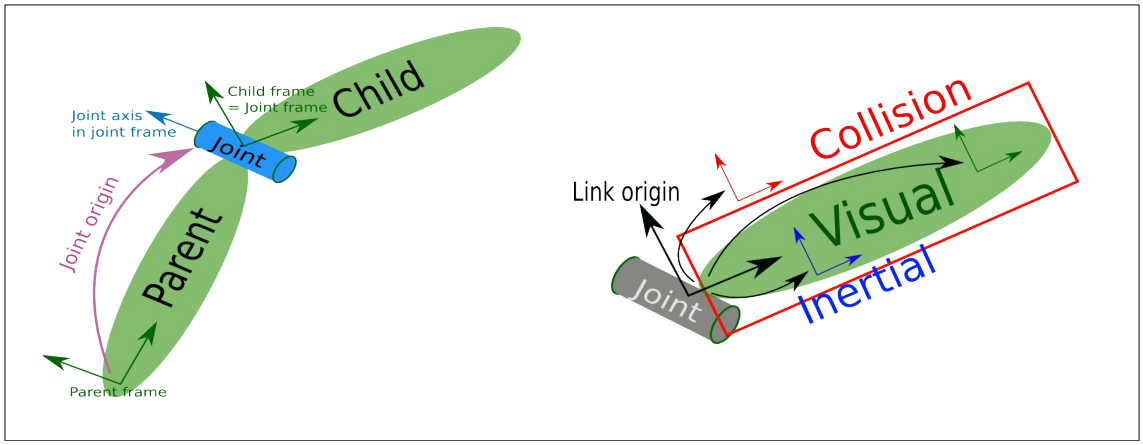


Ilustración 23: Articulaciones, eslabones y zonas de colisión

Para generar modelos de robots más elaborados se utilizan mallas que suelen ser generadas por los programas más específicos. *URDF* es capaz de incluir dichas mallas en su sistema e interpretarlas.

La malla del manipulador *Lwr* se ha obtenido de la Universidad de TUM (*Technische Universität München*) mediante un contrato de cooperación con Tecnalia. No es el caso de la mesa donde está colocado el manipulador, que se ha modelado mediante los tipos básicos del paquete *URDF*.

Tras obtener las medidas reales de la mesa se ha procedido a generar los primeros eslabones mediante cubos de diferentes medidas unidos por *joints*.
Tras lograr el aspecto real de la mesa mediante la unión de todos los cubos generados, se han aplicado diferentes texturas de aluminio para dar un mayor realismo a la simulación.

En el Código 4 se puede observar la programación necesaria para generar la mesa. A cada eslabón o *link* se le asigna un nombre y unas propiedades visuales y de colisión para su creación:

```
<joint name="${name}_table_floor_top_fixed_joint" type="fixed" >
  <parent link="${name}_table_floor_link" />
  <origin xyz="0 0 ${table_height}" rpy="0 0 0" />
  <child link="${name}_table_top_link" />
</joint>

<link name="${name}_table_top_link">
  <visual>
    <origin xyz="0 0 ${-table_top_thickness/2}" />
    <geometry>
      <box size="${table_depth} ${table_width} ${table_top_thickness}
"/>
    </geometry>
    <material name="gray_texture1">
      <texture filename="package://lwr_model/textures/aluminio.jpg"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 ${-table_top_thickness/2}" />
    <geometry>
      <box size="${table_depth} ${table_width} ${table_top_thickness}
"/>
    </geometry>
  </collision>

</link>
```

Código 4: Creación de la mesa que sostiene el manipulador

Con el fin de que ROS sea una herramienta genérica para la creación y definición de diferentes tipos de robots, ofrece una aplicación llamada W*izard* que se encarga de generar una cadena cinemática del robot. Mediante la creación del modelo *URDF* y la asignación de parámetros en la aplicación W*izard*, se obtienen los siguientes resultados de modo automático:

- Configuración de los grupos de planificación, principalmente de las cadenas cinemáticas para manipuladores.

- Generación de ficheros de configuración para comprobar los posibles choques entre eslabones del robot.

- Generación de ficheros de lanzamiento que resuelven la cinemática inversa de la cadena cinemática.

En la Ilustración 24 se observa la herramienta W*izard* en la que aparece un listado con todos los eslabones existentes en el modelo del robot. En los tres últimos campos denominados *Chain Name*, *Base Link* y *Tip Link* se debe introducir el nombre de la nueva cadena cinemática, del primer eslabón y del último eslabón respectivamente. En el campo *Tip Link* no se debe introducir el efector final ya que suele ser variable según el uso o tarea que se desee desempeñar por el manipulador.

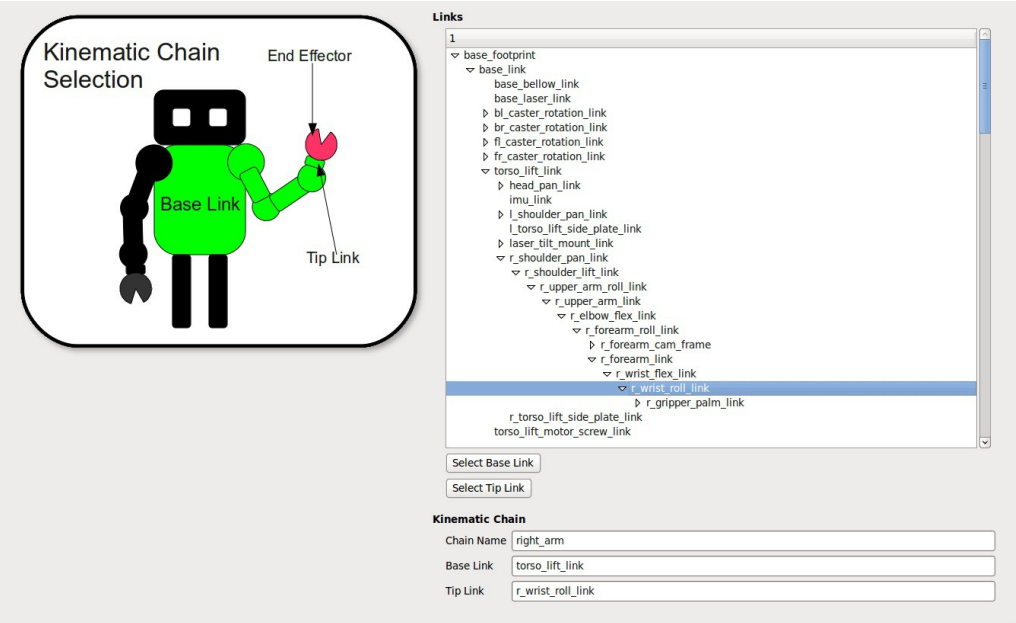


Ilustración 24: Creación de cadena cinemática mediante la herramienta Wizard

Tras la creación de la mesa y el acoplamiento en el *URDF* del modelo de manipulador se obtiene la siguiente visualización en *rviz*:

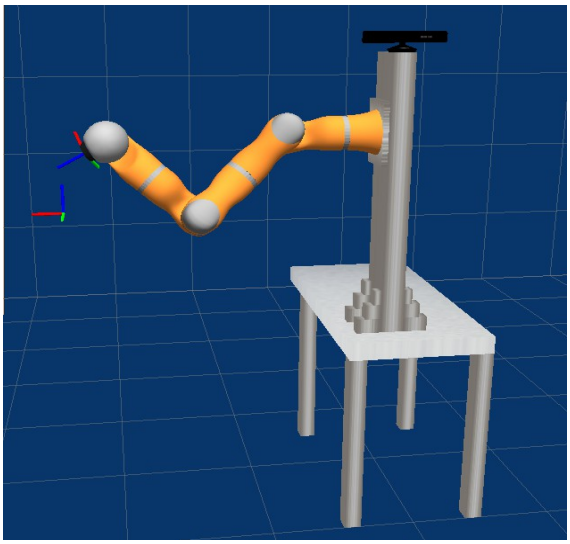


Ilustración 25: Modelo de la mesa y del manipulador Lwr visualizado en rviz

Con objeto de lanzar la aplicación de una manera más sencilla se ha generado un fichero de lanzamiento para que con una sola llamada de este fichero se pueda lanzar la visualización del robot [Cap 9.1/Comandos]. Esto incluye todos los ficheros *URDF* necesarios, además de la aplicación *rviz*.
El fichero recibe el nombre de *lwr_omni_teleop.launch.*

```
<launch>
  <param name="robot_description" command="$(find xacro)/xacro.py '$
(find lwr_model)/urdf/scenes/mast_mounted_lwr.urdf.xacro'"/>

  <node pkg="robot_state_publisher" type="state_publisher"
name="joint_state_to_tf"/>

  <!-- Start rviz -->
  <node name="rviz" pkg="rviz" type="rviz" respawn="true" args="-d $
(find lwr_model)/launch/omni_teleop_rviz.vcg" />
</launch>
```

Código 5: Lazador de modelo del manipulador y visualizador rviz

Tecleando en una terminal el siguiente comando, seremos capaces de visualizar en nuestro monitor la representación de la Ilustración 25.

roslaunch $\underbrace{lwr\_teleop}_{\text{paquete}}$ $\underbrace{lwr\_omni\_teleop.launch}_{\text{lanzador}}$.

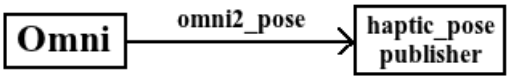
4.3 Gestión de la Cinemática Inversa. Modo Absoluto

Tras crear el modelo completo del robot y del háptico es el momento de profundizar en la teleoperación como tal. El primer objetivo es que el *Omni* envíe la posición y orientación de la punta de su *stylus* mediante un *topic* a otro nodo que se encargue de calcular la cinemática inversa de dicha posición en el sistema de referencia del *Lwr*. Por ello, se requiere que el manipulador y el háptico compartan un espacio de trabajo casi similar.

Para obtener la posición y orientación del *Omni* se utiliza el *topic omni2_pose*, que se publica y se alimenta de mensajes enviados por el nodo *omni* del paquete phantom_omni. La frecuencia de muestreo y de publicación de la posición y orientación del háptico es de 100 Hz.

Se ha creado un nuevo nodo que se suscribe a dicho *topic omni2_pose* en el que se reciben los datos de la localización del *Omni*.
Los datos son del tipo *PoseStamped* que son un tipo de datos que utiliza ROS para representar localizaciones en el espacio.

Estos pertenecen al paquete *geometry_msgs* e incluyen la posición dada en coordenadas cartesianas y la orientación dada en cuaterniones.



Este nuevo nodo al que llamamos *haptic_pose_publisher* se encarga de transformar los datos recibidos para obtener la posición de la punta del *Omni* vista desde el marco de referencia que nos interese. Para ello se ha creado un nuevo *TF* estático llamado *middle_link* el cual está colocado en la mitad del espacio de trabajo del *Omni* y, a su vez, en la mitad del de *Lwr*.

El motivo de este diseño se debe a que es necesario colocar el centro del espacio de trabajo del *Omni* en el centro del espacio de trabajo de *Lwr* para apartados futuros cuando hablemos de escalas aplicadas al *Omni*. No obstante, sobre éste *TF* se han realizado diversos cambios de localización en busca de la posición óptima.

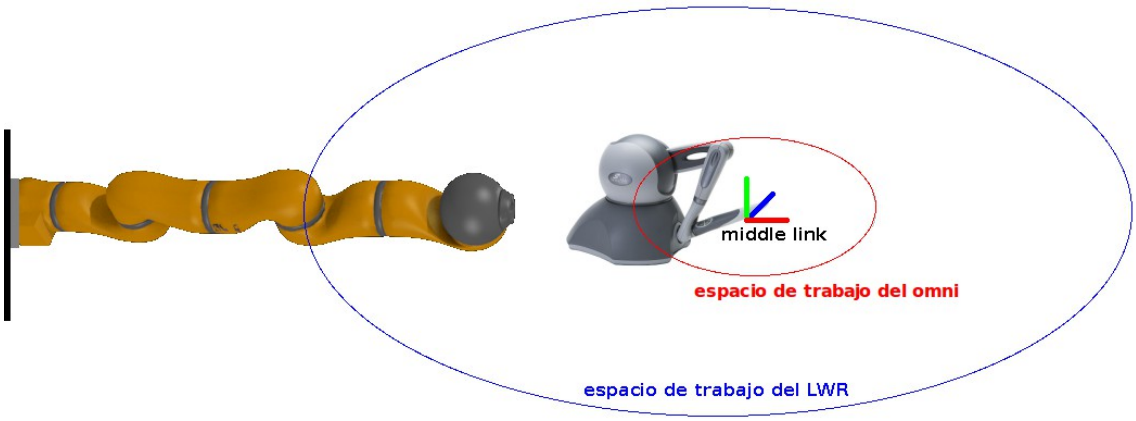


Ilustración 26: Espacio de trabajo Omni y Lwr. Posición del TF de referencia middle_link

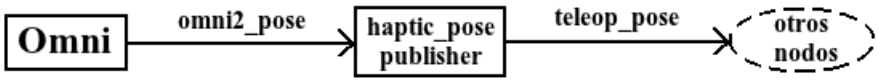
Por otra parte, cabe destacar que en ROS cada vez que se publica un mensaje en un *topic*, éste es recibido por todos los nodos que estén suscritos a dicho *topic*. Los nodos reciben los mensajes con la misma frecuencia que éstos hayan sido publicados, no obstante, el programador puede seleccionar los mensajes que le interesen o la frecuencia de muestreo de dichos mensajes.
El modo de obtener los mensajes se realiza mediante lo que en ROS se denomina *callback*, que es una función sencilla que recibe los datos.

En el siguiente fragmento de código [Código 6] se puede apreciar la suscripción al *topic omni2_pose* y cómo se tratan los datos recibidos para cambiar el marco de referencia del *Omni*. El objetivo es obtener en el eje de referencia *(pose_link0)* el resultado obtenido tras observar la posición del *Omni* (*omni2_pose)* vista desde el *frame* (*middle_link).*

```
 rospy.Subscriber("omni2_pose", PoseStamped, self.processOmni2Pose)

//suscripción base_link = "middle_link"
self.listener.waitForTransform("/"+base_link,"/omni2_link6",
rospy.Time(), rospy.Duration(4.0))

// middle_link --> /omni2_link6
pose_link0 = self.listener.transformPose("/"+base_link, pose_link6)

pose_link0.header.stamp = rospy.Time.now()
self.haptic_publisher.publish(pose_link0)
```

Código 6: Suscripción, tratamiento y publicación de la posición del Omni

Seguidamente se publica el resultado obtenido en *pose_link0,* en el nuevo *topic* creado llamado *teleop_pose*.



El nuevo objetivo se basa en calcular la cinemática inversa para *Lwr* obtenida a partir de la posición de *teleop_pose.* Para ello, el primer paso consiste en lanzar los servicios de cinemáticas que ofrece *Orocos KDL* para que puedan ser utilizados.
Es necesario conocer diversos parámetros para utilizar correctamente los servicios de KDL, entre los que destacamos:

- Los nombres de todas las articulaciones del *Lwr*: *lwr_arm_joint_0, lwr_arm_joint_1, ..., lwr_arm_joint_6*
- La posición articular del manipulador en el instante de la petición (*joint_state*)
- La posición cartesiana que se desea alcanzar (*teleop_pose*)

Esto es así ya que el servicio utilizado necesita conocer las propiedades y parámetros del manipulador para solucionar el problema cinemático. Tras conocerlos procede a calcular las posibles soluciones por un procedimiento iterativo. En el caso de que exista más de una solución para la configuración del robot (esto se da en robot redundantes como es el caso de *Lwr*), se elige la configuración articular del robot más próxima a la actual mediante la utilización de la métrica euclidiana.
En caso de no haber ninguna solución factible, se informa con un mensaje de error y el robot permanece inmóvil.

El siguiente fragmento de código muestra cómo se conecta con el servicio y se realiza la petición al servicio `GetPositionIK`.

```
 try:
   get_ik = rospy.ServiceProxy('kuka_lwr_kinematics/get_ik',
GetPositionIK)
   ik_res = get_ik(kin_req1)
   return ik_res
 except rospy.ServiceException, e:
   print "Service call failed: %s"%e
   return ik_res
```

Código 7: Utilización del servicio de cinemática inversa ofrecido por KDL

En la variable *ik_res* se encuentra la solución al problema cinemático. Puede albergar dos posibles soluciones, la nueva posición articular del robot y un número entero que indica el tipo de error que se ha producido.
Los errores más comunes aparecen en la siguiente tabla, siendo los marcados en negrita los mas frecuentes:

# kinematics errors	# general errors
NO_IK_SOLUTION = -31	INVALID_TIMEOUT = -36
INVALID_LINK_NAME = -32	# overall behavior
IK_LINK_IN_COLLISION = -33	PLANNING_FAILED = -1
NO_FK_SOLUTION = -34	**SUCCESS = 1**
KINEMATICS_STATE_IN_COLLISION = -35	**TIMED_OUT = -2**

En la siguiente columna de datos se puede observar la posición cartesiana que se desea alcanzar por el manipulador cuya configuración corresponde con el ejemplo observado en la Ilustración 25. Dicha posición se ha obtenido interceptando un mensaje del *topic teleop_pose.*

```
pose:
  position:
    x: -0.0543202359067
    y: -0.000268538224786
    z: 0.0932872392256
  orientation:
    x: -0.150945484986
    y: -0.0131531020742
    z: 0.987930340867
    w: -0.0321884771759
```

Para la posición posición anterior se ha calculado la siguiente cinemática inversa:

```
name: ['lwr_arm_joint_0', 'lwr_arm_joint_1', 'lwr_arm_joint_2', 'lwr_arm_joint_3', 'lwr_arm_joint_4',
       'lwr_arm_joint_5', 'lwr_arm_joint_6']

position: [-2.231468313863743, 0.73843458679997853, -0.065683451129718406, 1.5670172411097878,
           0.28745399017239126, 0.63441010534579667, -1.0084691472916245]
velocity: [ ]
effort: [ ]
```

El atributo *position* almacena la posición articular de cada una de las siete articulaciones del manipulador, expresada en radianes.
Esta solución se envía al controlador del robot con el fin de que el robot se mueva a esa configuración.
Concretamente la solución se envía en forma de mensaje ROS escribiendo la información en el *topic* con nombre *target_joint_state.* El tipo de mensaje corresponde con *joint_state* y guarda la posición articular de cada una de las articulaciones del robot.
Esta información es recibida por *Reflexxes* que es el encargado de realizar una interpolación de todas las soluciones dadas para que el robot se mueva de forma continua y no discreta en el tiempo ya que pasa por multitud de puntos en un intervalo de tiempo muy pequeño. Con ello se consigue un suavizado de la trayectoria y un mayor realismo al teleoperar ya que no se aprecian saltos ni movimientos bruscos durante la operación.
Reflexxes respeta tanto los límites físicos como la velocidad y aceleración máxima de cada articulación.

4.4 Pruebas cinemáticas con robot real

Una vez realizadas diversas pruebas con el modelo en la simulación, y visto que el comportamiento y dinámica del robot es correcta y no pone en peligro la integridad del robot real ni del personal de alrededor, procedemos a conectar el PC con el controlador del manipulador o *KRC* mediante Ethernet.

El primer requisito para establecer la conexión es lanzar el *FRI* que, como ya se ha comentado, es la interfaz rápida de desarrollo que KUKA ha implantado en el controlador para que éste sea accesible desde un PC externo.
Se lanzan los comandos *Fri_open* y *Fri_start,* para que la información del robot se envíe al PC y viceversa.
Seguidamente se lanza *Reflexxes* cuya finalidad es que se encargue de las trayectorias del manipulador y de la interpolación de los datos enviados desde el PC que calcula las cinemáticas.
El controlador, a su vez, envía datos sobre el estado del manipulador, temperatura de las articulaciones, posición cartesiana, posición articular, par en cada articulación y par transmitido al último efector, etc.

Nos basamos en los datos recibidos por el manipulador para calcular la cinemática inversa en nuestro PC. Las posiciones articulares calculadas, también conocidas como *q,* son enviadas al robot pasando previamente por *Reflexxes* para que sean interpoladas.

Existe un riesgo inherente en la aplicación ya que al comenzar la teleoperación el manipulador se encuentra en posición de reposo, esto es, completamente extendido horizontalmente. El *Omni*, por el contrario, se encuentra en otra configuración completamente diferente debido a su fisonomía.
Es por ello que en el momento de arrancar la teleoperación la primera posición articular deseada que es enviada al robot es muy diferente a la que posee en ese momento el manipulador. Por ello se teme que *Lwr* realice un salto brusco entre su posición actual y la deseada.
No obstante, esto no supone un problema para *Reflexxes* ya que desarrolla una trayectoria correcta entre ambos puntos y el robot se mueve de manera continua y sin saltos o brusquedades.
Para las próximas posiciones articulares deseadas no existe mayor problema para *Reflexxes* ya que son enviadas 40 veces por segundo y mediante interpolación se alcanza la cantidad de datos necesaria.

Tras la realización de la primera prueba, los resultados han sido satisfactorios ya que *Lwr* se mueve de manera continua siguiendo todas las trayectorias dictadas por el *Omni*.

4.4.1 Resultados y conclusiones

La configuración de los componentes *hardware* del sistema se conectan mediante Ethernet del siguiente modo:

KRC → router → PC de cálculo

Tras realizar diversas pruebas se ha considerado que la configuración de los componentes *hardware* no es la óptima, ya que se originan retardos producidos debido a que el PC consta de dos tarjetas de red, de las cuales una está conectada a la red Internet y la otra, mediante un *router*, al controlador KRC.
Esta configuración genera mucho tráfico porque existen en torno a cinco equipos conectados a dicho *router* al mismo tiempo. El motivo de este diseño se debe a que el manipulador *Lwr* se emplea en diferentes proyectos de Tecnalia y de esta manera no es necesario reiniciar las conexiones de red cada vez que un ordenador quiere conectarse con el manipulador.
Como solución a esta cantidad de datos de red y posterior gestión en cada tarjeta de red, se ha cambiado la configuración de la red, conectándose ahora el robot directamente al PC y eliminando la conexión a Internet durante el proceso de teleoperación.

Otro problema a considerar es la velocidad de cálculo de las cinemáticas por parte de nuestro PC. Tras calcular la cinemática ésta se envía al robot con una frecuencia determinada.
En una primera aproximación la frecuencia utilizada ha sido de 10Hz, pero ha sido algo insuficiente si consideramos la frecuencia con la que trabaja *Lwr* que es 1KHz. Por lo que se ha aumentado la frecuencia de envío a de 10Hz a 100Hz.

ROS incluye un comando que nos permite conocer la frecuencia a la que se están escribiendo los mensajes en un *topic*:

rostopic hz *nombreTopic*

Tras realizar un muestreo del *topic target_joint_state*, en el cual se escriben los resultados obtenidos tras el cálculo de la cinemática inversa, se ha observado que la velocidad de escritura no supera los 40Hz. Esto es debido a que el computador usado para el cálculo es limitado en cuanto a su capacidad y no está orientado para este fin. El procesador del PC no tiene el potencial necesario para calcular cinemáticas de manera iterativa con una frecuencia de 100 Hz.
Por tanto, el cuello de botella de la aplicación se origina en el PC de cálculo.

4.5 Gestión de la Cinemática Directa

El siguiente objetivo se basa en conocer si las posiciones articulares calculadas son correctas y si el manipulador se posiciona realmente donde le estamos indicando desde el *Omni*. Para ello es necesario obtener la cinemática directa del manipulador [Cap 3.4].

Para este caso *Orocos KDL* ofrece un servicio que calcula la cinemática directa. Es necesario conocer el estado de las articulaciones en el momento de la petición e indicar los siguientes parámetros:

- Eslabón desde el cual se va a calcular la cinemática directa indicando el origen de la cadena cinemática.

- Eslabones de la cadena cinemática que nos interese conocer su posición cartesiana. En este caso, es suficiente con conocer la posición en el espacio de la última articulación.

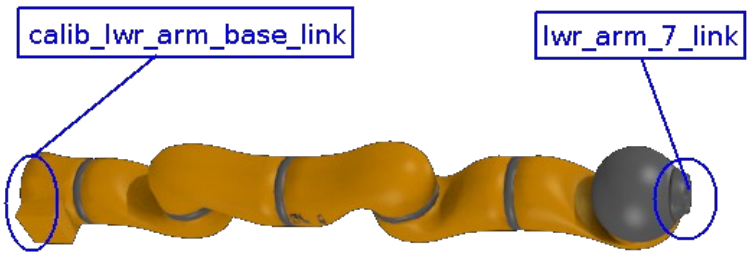


Ilustración 27: Eslabones necesarios para cinemática directa

Tras comprobar la funcionalidad de este nuevo nodo se puede observar que los cálculos realizados son correctos y que *Lwr* se posiciona fielmente en las posiciones indicadas por el *Omni*.

4.6 Escalas

Como se puede observar en la Ilustración 26, el espacio de trabajo del *Omni* es considerablemente menor que el de *Lwr*, es por ello que no podremos abarcar todas las posiciones posibles, limitando las capacidades y funcionalidades del robot.

Por esta razón, se ha optado por incorporar un factor de escala adaptable que posea la opción de reducir o aumentar la posición deseada.
Por defecto, el factor de escala es igual a 1, ya que no deseamos notar variaciones al inicio de la teleoperación. Esta se puede variar en un rango de reducción, desde 0,1 a 0,9 y de aumento, desde 1,1 hasta 4,0. Con este valor máximo de 4,0 aseguramos que todo o casi todo el espacio de trabajo del *Lwr* se cubre desde el *Omni*. Se ha aplicado esta restricción del valor máximo por seguridad, ya que de lo contrario los movimientos se volverían inestables y peligraría el entorno de trabajo y la mecánica del propio robot.

La variación de amplitud se consigue mediante una simple operación que se basa en una multiplicación de cada componente de la posición cartesiana multiplicada por dicha escala.

Con ello conseguimos no solo abarcar todo el espacio de trabajo del *Lwr*, sino que también dotamos al sistema de mayor precisión, ya que, en modo reducción, un movimiento grande en el *Omni* se traduce en un movimiento muy pequeño en el robot. Esto puede ser interesante para operaciones que requieran una gran precisión en un espacio de trabajo muy reducido, o a la inversa, para operaciones rápidas donde se requiera alcanzar puntos muy dispares del espacio de trabajo en poco tiempo.

Anteriormente se ha comentado de la importancia de seleccionar correctamente el punto desde el cual se va a calcular la posición de la punta del *Omni*, en otras palabras, del *middle_link*.

Por defecto, *Omni* emite la posición de la punta de su *stylus* desde una referencia interna del dispositivo que se encuentra en su centro. Con lo que una posición de la punta cercana a esta referencia del dispositivo emitiría una posición igual o muy cercana a cero en su coordenada y, por el contrario, una posición de la punta alejada del centro daría un valor mayor a cero. Con esta configuración no existen los valores negativos en esa coordenada ya que por su configuración la punta siempre se desplaza desde el centro para adelante.

En cambio con la referencia *middle_link* obtenemos valores con rango (-x , 0 , +x), (-y , 0 , +y) (-z , 0 , +z)

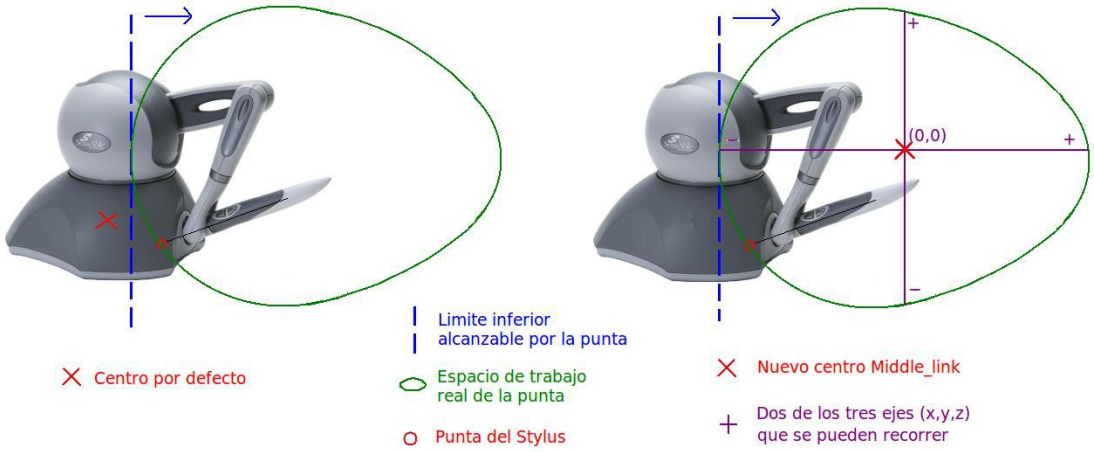


Ilustración 28: Cálculo de posición respecto a nueva referencia middle_link

Este cambio de referencias es necesario ya que al aplicar la escala al *Omni*, el espacio recorrido crecía en forma de pseudo-óvalo, generando movimientos muy pequeños cuando la punta del *Omni* se situaba cerca del centro y movimientos muy grandes cuando la punta se situaba en posiciones lejanas al centro.

En cambio, en este momento el espacio recorrido crece en forma de circunferencia/esfera por lo que todos los puntos los espacio de trabajo son equidistantes. Cualquier movimiento en el *Omni* con escala aplicada se va a apreciar correctamente en el robot ya que los dos centros son coincidentes.

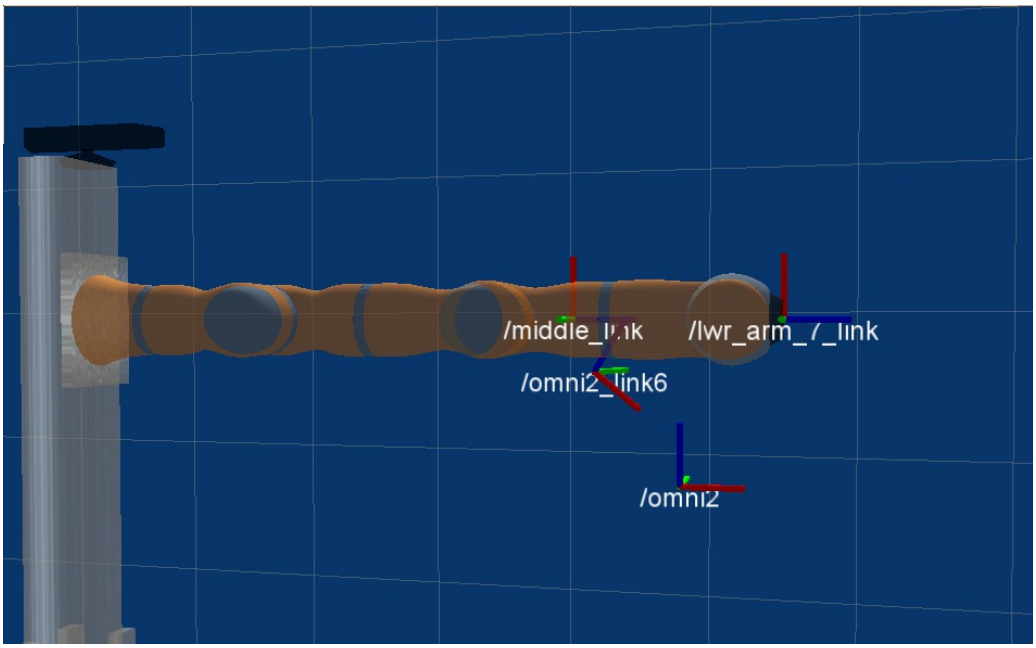


Ilustración 29: Visualización mediante rViz del modelo de Lwr y de los TF del Omni y middle_link

4.7 Gestión de la Cinemática Inversa. Modo Relativo

En el siguiente apartado se va a describir una manera diferente de teleoperar un manipulador. Hasta ahora sólo se había documentado acerca del modo absoluto, el dispositivo háptico envía posiciones y el robot reproduce fielmente esas posiciones una a una.
Para una teleoperación rápida, sin demasiada complejidad, esta opción es muy recomendable. Pero existen casos para los que hay que llegar más allá. Debido a los siguientes factores, se ha programado la teleoperación en modo relativo:

1. Por seguridad, es muy común que en aplicaciones donde interaccionan humanos con robots exista un botón en el sistema, llamado *hombre-muerto,* que ha de ser pulsado en todo momento para que el robot este operativo. En caso de que este botón sea liberado por el operador, el robot se para automáticamente. Esto es importante ya que si el operador sufre un accidente, desmayo u otra circunstancia, con el simple gesto automático de liberar el botón el robot se parará, y por tanto, no supondrá un peligro.

2. Poder abarcar zonas del espacio de trabajo que mediante la manera absoluta no se pueden alcanzar.

3. La teleoperación resulta mucho más cómoda. Debido a que se pueden realizar

descansos en diferentes momentos de una misma tarea.
El *stylus* del *Omni* posee dos botones que pueden ser programados para realizar diferentes funciones, en nuestro caso vamos a programarlos como botones *hombre-muerto.* Para dar más facilidades al usuario y por ergonomía, se va a ofrecer la posibilidad de utilizar cualquiera de los dos botones indistintamente.
El objetivo es que el *Omni* envíe las posiciones deseadas sólo cuando un botón esté pulsado.
Esto puede parecer sencillo si se capturan los eventos de los botones y sólo se publica en el *topic* en el momento preciso, no obstante, surge un problema, ya que si se pulsa un botón del háptico en una posición, se suelta, y se vuelve a pulsar en una posición lejana a la primera, al robot le van a llegar dos posiciones muy dispares y puede que *Reflexxes* no pueda generar una trayectoria o, peor aún, generar una trayectoria que entre en colisión con un objeto del entorno, dañando por tanto al objeto o al propio robot.

Por ello, se pretende implementar el modo de teleoperación relativo.
Todos los movimientos realizados por el *Omni*, mientras un botón está pulsado, son relativos a un punto definido en el instante de pulsar uno de los botones. Para conocer el estado de los botones nos suscribimos al *topic* con nombre *omni2_button,* el cual posee una variable de tipo *boolean* por cada botón que cambia su estado a *True* cuando se ha pulsado un botón. El valor *boolean* de esta variable se recoge en un *flag*.

A continuación se van a incluir y a explicar diferentes fragmentos del código utilizado en la programación para entender mejor su funcionamiento:

```
def calculateAX(self):
# There aren't any button pressed
  if self.buttonPressed.white_button == 0 and
self.buttonPressed.grey_button == 0:
    self.flag = False
    self.flag_publisher.publish(False)
    self.x_to_ik = self.getFk().pose_stamped[0]
# Here the robot doesn't move
# If one button is pressed, the first iteration, 'x_omni' pose saves
as a 'x_omni_last'
  else:
    if self.flag == False:
      self.x_omni_last = self.x_omni
      self.x_robot_last = self.getFk().pose_stamped[0]
      self.flag = True
```

Código 8: Control de pulsación de botones y guardado de primeros valores

Esta primera parte del código [Código 8] comprueba si se ha pulsado algún botón. En el caso de no haber ninguno, el *flag* toma el valor *False* y se almacena la posición cartesiana actual del *Lwr*, para ello se utiliza el servicio de cinemática directa.
En cambio, si se pulsa un botón, se almacena la posición cartesiana en ese instante tanto del *Omni* como del *Lwr* y el *flag* toma el valor *True*.

```
elif self.flag == True:

  Ax_omni = PoseStamped()     # Get the Ax_omni

  Ax_omni.pose.position.x = self.x_omni.pose.position.x –
self.x_omni_last.pose.position.x

  Ax_omni.pose.position.y = self.x_omni.pose.position.y –
self.x_omni_last.pose.position.y

  Ax_omni.pose.position.z = self.x_omni.pose.position.z –
self.x_omni_last.pose.position.z

# R1 * R2 = R_result  --> R2 = R1^-1 * R_result
  rotation_omni_1 =
PyKDL.Rotation.Quaternion(self.x_omni_last.pose.orientation.x,
self.x_omni_last.pose.orientation.y,
self.x_omni_last.pose.orientation.z,
self.x_omni_last.pose.orientation.w)

  rotation_omni_result =
PyKDL.Rotation.Quaternion(self.x_omni.pose.orientation.x,
self.x_omni.pose.orientation.y, self.x_omni.pose.orientation.z,
self.x_omni.pose.orientation.w)

  rotation_omni_2 = rotation_omni_1.Inverse() * rotation_omni_result
```

Código 9: Se aplica al Lwr la rotación generada por el Omni

Esta segunda parte [Código 9] se calcula el incremento de posición (Δx) en el *Omni*. Para ello se accede a cada componente x, y, z por separado y se halla la diferencia entre la posición actual y la posición almacenada en el momento de pulsar el botón.

Para hallar el incremento o la diferencia en la orientación del *Omni*, el procedimiento difiere del anterior ya que es necesario utilizar matrices de rotación para dicho fin [Cap 4.8]. La librería *PyKdl* nos ayuda a calcularlas, no obstante se ha de crear una estructura del tipo *PyKDL.Rotation.Quaternion* para la rotación inicial y final.

Se entiende *R1* como la orientación inicial del háptico y *R_result* como la orientación final. *R2,* en cambio, es cuánto ha rotado el *Omni* desde la posición inicial hasta la posición final.

Para conocer el valor de *R2* es necesario despejar la siguiente ecuación:

$$R1 * R2 = R\_result \rightarrow R2 = R1^{-1} * R\_result$$

El resultado obtenido en R2 es la cantidad de rotación que se debe aplicar a *Lwr* para que realice la misma rotación que el *Omni* ha realizado.

```
self.x_to_ik.pose.position.x = (Ax_omni.pose.position.x * self.alfa) +
self.x_robot_last.pose.position.x

self.x_to_ik.pose.position.y = (Ax_omni.pose.position.y * self.alfa) +
self.x_robot_last.pose.position.y

self.x_to_ik.pose.position.z = (Ax_omni.pose.position.z * self.alfa) +
self.x_robot_last.pose.position.z
```

Código 10: Se aplica a Lwr el movimiento realizado por el Omni

Se ha de añadir a la posición almacenada del *Lwr,* el diferencial de posición del *Omni* previamente calculado y multiplicado por el factor de escala deseado. De este modo el desplazamiento del manipulador será el mismo que el del *Omni*.

```
rotation_robot_1 =
PyKDL.Rotation.Quaternion(self.x_robot_last.pose.orientation.x ,
self.x_robot_last.pose.orientation.y ,
self.x_robot_last.pose.orientation.z ,
self.x_robot_last.pose.orientation.w)

rotation_robot_result = rotation_robot_1 * rotation_omni_2
self.x_to_ik.pose.orientation.x = rotation_robot_result.GetQuaternion[0]
self.x_to_ik.pose.orientation.y = rotation_robot_result.GetQuaternion[1]
self.x_to_ik.pose.orientation.z = rotation_robot_result.GetQuaternion[2]
self.x_to_ik.pose.orientation.w = rotation_robot_result.GetQuaternion[3]
self.flag_publisher.publish(True)
```

Código 11: Se aplica el resultado de la rotación R2 al Lwr

Del mismo modo que en el Código 10, se le aplica al *Lwr* la cantidad de rotación realizada por el *Omni* y previamente calculada en la rotación R2.

Mediante este procedimiento se ha resuelto el modo de teleoperación relativo. En este momento somos capaces de dirigir el *Lwr* durante el periodo de pulsación de uno de los botones del *Omni* con lo que nos aseguramos llegar a todas las zonas del espacio de trabajo. El teleoperador dispone de seguridad a la vez que comodidad y ergonomía a la hora de teleoperar.

4.8 Matrices de Rotación

Una matriz de rotación representa una rotación en el espacio euclídeo. Es ortogonal, cuadrada, con valores reales y de determinante igual a uno.

El motivo por el que el cálculo de incrementos en el movimiento de traslación y en el de orientación difiere, se debe a que la traslación posee la propiedad conmutativa y la orientación en cambio, no, entendiendo que la propiedad conmutativa se da cuando el resultado de una operación es el mismo, cualquiera que sea el orden de los elementos con los que se opera.
Sea E un conjunto en el cual se ha definido una operación binaria o ley de composición interna *, es decir una aplicación:

$$\begin{aligned} * &: E \times E \rightarrow E \\ (x, y) &\rightarrow z = x * y \end{aligned}$$

Se dice que * es conmutativa si verifica para todo *(x , y)* de $E \times E$ la siguiente igualdad $x * y = y * x$.

$$\forall (x, y) \in E^2, \quad x * y = y * x$$

Por ejemplo, si para el caso de traslación, un objeto se desplaza 3 unidades hacia la derecha seguido de otro desplazamiento de 8 unidades hacia arriba, nos lleva al mismo lugar que si efectuáramos primero un desplazamiento de 8 unidades hacia arriba seguido de una traslación de 3 unidades hacia la derecha. Como con ambos movimientos se produce el mismo resultado se puede decir que se cumple la propiedad conmutativa.
Por el contrario, si realizamos una rotación en un eje y a continuación otra rotación en otro eje, y realizamos esta misma operación cambiando el orden de ejecución, esto nos llevaría a orientaciones diferentes. En la Ilustración 30 se puede observar de manera gráfica la no conmutatividad en el movimiento de rotación.

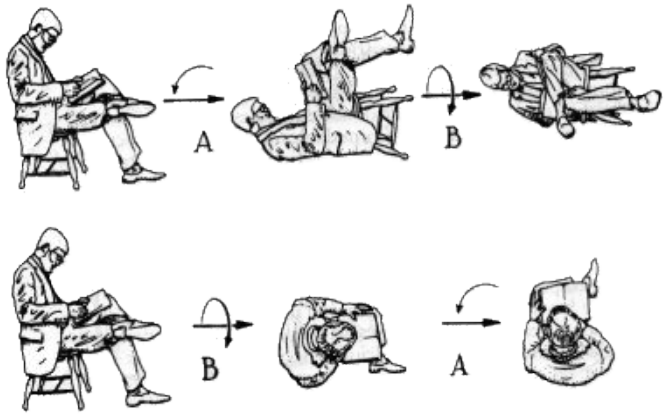


Ilustración 30: Demostración de la no conmutatividad en la Rotación

La rotación más sencilla es la que tomamos un vector situado en el plano x-y girándolo en sentido antihorario, lo cual equivale a tomar un eje-z invisible y perpendicular al plano aplicándole un giro por un ángulo θ, moviendo el vector **v0** a su nueva posición **v'**
La longitud del objeto rotado debe permanecer inalterada.
En la Ilustración 31 se puede observar la rotación vista desde cada plano.

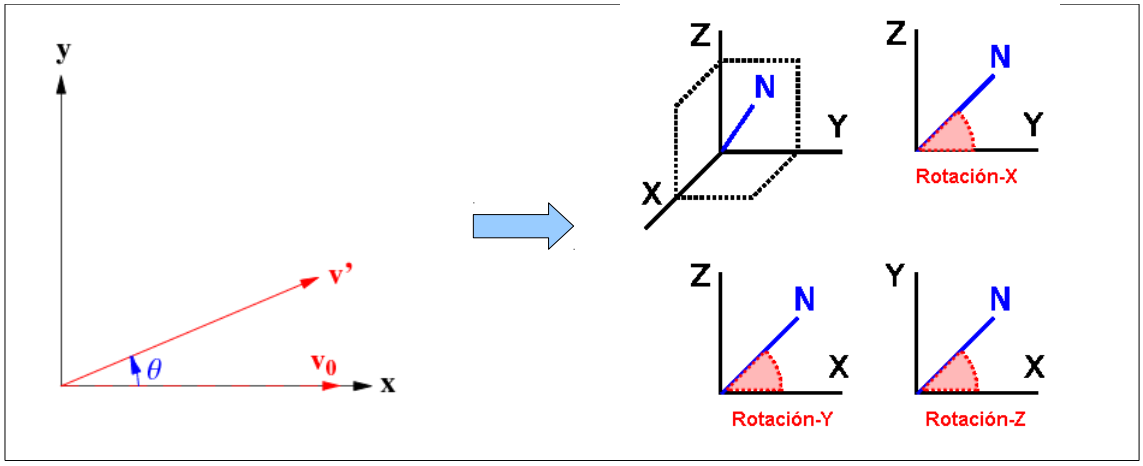


Ilustración 31: Proyección en el plano de una rotación

Para el caso de un solo plano, por ejemplo el x-y, es suficiente con utilizar una matriz de rotación 2*2.

$$R^T = R^{-1} \;\; y \;\; det\, R=1$$

$$R_z(\theta)=\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}$$

Esta matriz de rotación, aplicada como un *operador* a un vector **v**, nos produce un vector **v'** cuya longitud es la misma que la del vector original, aunque las proyecciones verticales y horizontales hayan cambiado:

$$\mathbf{v'} = \mathbf{R_z}\,\mathbf{v}$$

Para un rotación de 30 grados, por ejemplo, la matriz de rotación será la siguiente:

$$R_z(\theta)=\begin{bmatrix}\cos(30) & -\sin(30) \\ \sin(30) & \cos(30)\end{bmatrix}=\begin{bmatrix}0{,}86 & -0{,}5 \\ 0{,}5 & 0{,}86\end{bmatrix}$$

Si las componentes horizontal y vertical del vector original **v** son a=3 y b=1, sus nuevas componentes a' y b' después de la rotación serán:

$$a' = (0{,}86) \cdot a - (0{,}5) \cdot b = 2{,}58 - 0{,}5 = 2{,}08$$

$$b' = (0{,}5) \cdot a + (0{,}86) \cdot b = 1{,}5 + 0{,}86 = 2{,}36$$

Si lo que vamos a describir son rotaciones en un sistema de coordenadas tridimensional, entonces no nos basta con una sola matriz 2x2, necesitamos tres matrices 3x3.

$$R_x(\theta)=\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad R_y(\theta)=\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta)=\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Estas matrices de rotación realizan rotaciones de vectores alrededor de los ejes *x*, *y*, o *z*, en el espacio de tres dimensiones.

4.9 Instalación Mano Robótica

Una vez superado el problema de las cinemáticas y de los modos de teleoperación podemos observar cómo el manipulador se posiciona y orienta fielmente conforme a las instrucciones del teleoperador.
Tras el cumplimiento de estos objetivos es interesante buscar una situación real en la que aplicar esta tecnología, por lo que Tecnalia ha adquirido una mano robótica con la finalidad de realizar tareas de manipulación e intercambio de objetos, también conocido como *grasping* y *pick and place*.

La mano es de la compañía *Robotiq*, concretamente el modelo S. Dispone de multitud de opciones para agarrar objetos por lo que le convierte en un modelo muy versátil y robusto. En la Ilustración 32 se pueden observar los distintos modos de agarre que posee la mano, empezando por agarre a lo ancho, el modo tijera, el pellizco y por último el agarre básico para tareas más sencillas.

Ilustración 32: Diferentes modos de agarre ajustables para cada tipo de objeto

Para instalarla a *Lwr* se ha creado un anillo mecanizado de un diámetro preciso que hace de unión entre la parte inferior de la mano y el último eslabón de *Lwr*. De esta manera queda bien fijada al robot.
Existe un pequeño hándicap en la instalación, ya que el cableado de la mano queda suspendido en el aire hasta el PC de control produciendo un efecto visual no deseable a la vez que posibles enganches con el manipulador y con objetos del entorno. Con lo que no respeta las directrices especificas para que pueda considerarse un robot apto para trabajar en colaboración con humanos.

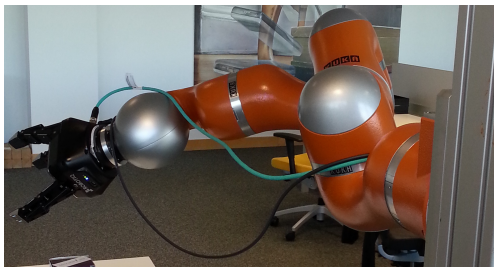


Ilustración 33: Cableado exterior

No obstante y gracias al un buen diseño, *Lwr* posee en su interior un cableado genérico que va guiado a través de los cilindros metálicos que componen su estructura. En el extremo inicial y final del manipulador sobresalen dichos cables que se han utilizado para generar nuevos conectores compatibles con los que utiliza la mano robótica para, de esa manera, enviar los datos de control desde el PC sin necesidad de cables colgantes en el entorno.

Para utilizar la mano es necesario realizar una comunicación previa entre la mano y el PC mediante *EtherCat*. Todos los requisitos que a la comunicación se refiere los ha gestionado otra división de Tecnalia, quedando, por tanto, transparentes para nosotros.

Tras la comunicación inicial ya nos encontramos en situación de enviar ordenes a la

mano. Para ello, utilizamos comandos en forma de mensajes que se escriben en un *topic ROS*. Es necesario rellenar todos los campos del cuerpo de los mensajes para que éstos sean interpretados por el controlador de la mano.
A continuación se muestra un ejemplo de mensaje que realiza una apertura total de la mano:

```
rostopic pub /robotiq/command soem_robotiq_drivers/RobotiqSModelCommand
'{activate: True, finger_a_position_req: 254, finger_a_speed: 254,
individual_finger_control: True, go_to_position: True}'
```

Código 12: Comando para apertura total de la mano robótica

Como se puede observar se publica mediante el comando *rostopic pub* seguido del nombre del *topic* deseado.

El resto es el cuerpo del mensaje que incluye:

activate	Activar la mano
finger_a_position_req	Indica la posición deseada para el dedo A
finger_a_speed	Velocidad deseada para alcanzar una posición
individual_finger_control	Activado cuando se desea utilizar un único dedo
go_to_position	Activado para que la mano se dirija a la posición deseada

Si se desea utilizar un modo de agarre de los especificados en la Ilustración 32, se debe indicar en el propio mensaje. Por ejemplo, si se necesita utilizar el modo tijera para coger un objeto pequeño se debería incluir lo siguiente:

```
activate: True, scissor_position_req: 0, scissor_speed: 100,
individual_scissor_control: True, go_to_position: True}
```

Código 13: Comando para posicionar la mano en modo tijera

Los mensajes también se pueden enviar desde programas escritos en código *python* creando estructuras dedicadas para ello, como se puede apreciar en el ejemplo:

```
cmd = RobotiqSModelCommand()  # Hand command
cmd.individual_scissor_control = False
cmd.activate =  True
cmd.go_to_position = True
cmd.gripper_position_req = 0
cmd.gripper_speed = 120
cmd.gripper_force = 100
self.command_pub.publish(cmd)
```

Código 14: Código Python para el control de la mano robótica

La mano robótica es capaz de suministrar información de su estado al computador, la cual la obtenemos del *topic /robotiq/feedback* que nos aporta información acerca de la posición de los dedos, el tipo de agarre, la velocidad utilizada, etc.

Las valores de las posiciones de los dedos y la velocidad de movimiento están escalados entre 0 y 255 unidades, siendo 0 completamente cerrados y velocidad lenta, y 255 completamente abiertos y con velocidad máxima.
El comando más importante y que más se ha ajustado a lo largo del proyecto es el de *gripper_force* ya que de él depende la fuerza de agarre de los dedos. Para cada tipo de aplicación se debe retocar este parámetro en función del tipo de objeto que se vaya a manipular.

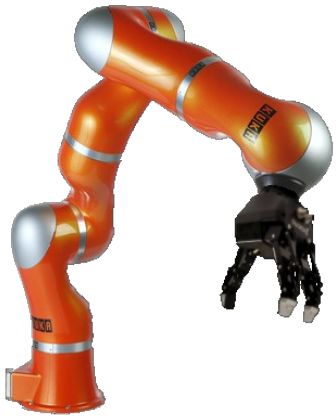


Ilustración 34: Mano robótica acoplada a Lwr utilizando los cables internos del robot

4.10 Método de Aprendizaje del manipulador

Una tecnología que está en auge en la robótica actual es la conocida como *Programing by Demonstration* [Calinon & Billard, 2007], [Zöllner & Dillmann , 2004]. Se basa en la capacidad para enseñar tareas al robot de manera sencilla y, en consecuencia, el robot sea capaz de repetir esas tareas autónomamente.
Es muy utilizado en tareas repetitivas donde el manipulador ejecuta siempre los mismos movimientos. O también, para tareas más complicadas donde estas lecciones aprendidas forman parte de un gran número de primitivas que el robot utilizará para tomar sus propias decisiones.

El objetivo es que un operador sea capaz de mostrar una tarea al manipulador mediante teleoperación y que éste sea capaz de repetirla autónomamente.
Esta capacidad tiene una gran acogida en las empresas, ya que a la hora de programar un robot para que desempeñe una función ya no es necesario que el trabajador posea conocimientos de programación o que conozca características especificas del robot.

La idea inicial es capturar las posiciones articulares del *Lwr* a lo largo de toda la tarea. Para ello, es necesario suscribirse al *topic joint_state,* donde el robot publica su estado y posición.
Todos los datos capturados se almacenan en un fichero especial creado gracias a la librería *numpy* que aporta *python*. Por cada tarea enseñada se crea un directorio con extensión `.npz` que recibe el nombre que considere el usuario y dentro de él siete ficheros con el estado de cada una de las articulaciones de *Lwr*. Estos siete ficheros que se ordenan de cero hasta seis tienen extensión `.npy`. Son de tamaño variable, dependiendo del tiempo de ejecución de la tarea.

En el caso de ejecutar un tarea de aprendizaje con la mano *Robotic* instalada, es necesario crear un nuevo fichero `.npy` que almacene la posición de cada uno de los dedos. Éste corresponde con el fichero número siete, en un total de ocho ficheros de aprendizaje.
Debido a que el manipulador y la mano son independientes en cuanto a frecuencia de mensajes que se publican en los *topics*, surge el problema de sincronizar ambos para que se capturen los movimientos de la mano en el momento exacto. Para solucionarlo, se almacenan los datos de posición de los dedos de la mano tantas veces como mensajes se hayan recibido del manipulador, para de ese modo, el número de datos de los ficheros del manipulador y de la mano sea el mismo.
En el preciso momento que la mano varía su posición, se almacenan los datos recibidos de la mano durante la ejecución del movimiento con la misma frecuencia que la utilizada por el *Lwr.* Con lo que, mediante esta sincronización, al final de la ejecución tendremos la misma cantidad de datos para los ocho ficheros.

Para reproducir los datos, en primer lugar se posiciona el *Lwr* en la posición inicial de la tarea con una velocidad controlada. El siguiente fragmento de código se encarga de ello:

```
rospy.loginfo('Slowly moving the robot to the initial position (in 5
sec.): ' + str(initial_position))

current_position = list(current_position.position)
joint_position_command.position = list(current_position)

rate = rospy.Rate(50)
for t in range(5*50):
  for i in range(7):
    joint_position_command.position[i] = current_position[i] +
(initial_position[i] – current_position[i])*(t/(5.*50.))

    joint_position_command.header.stamp = rospy.Time.now()
    joint_position_command_pub.publish(joint_position_command)
    rate.sleep()
```

Código 15: Posicionamiento del manipulador en la posición incial de la tarea

Una vez alcanzada la posición inicial, se procede a acceder a la totalidad de los datos en masa para leerlos, tratarlos y publicarlos en el *topic target_joint_state* al que está suscrito *Reflexxes* para repetir la trayectoria. Con ello, el robot comienza a realizar la tarea previamente aprendida con una excelente exactitud.

4.11 Detección de Colisiones

En ocasiones cuando se teleopera no se dispone de una telepresencia adecuada, el usuario no es capaz de ver o interpretar el entorno de trabajo y pueden surgir problemas. Por ello, se introducen mejoras al sistema como pueden ser cámaras, retroalimentación de fuerzas, guiados por computador, etc.

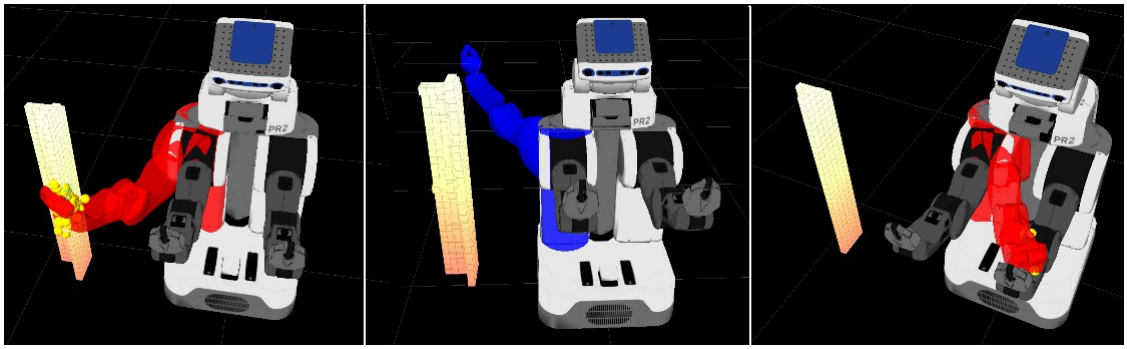
Como primer paso se pretende que el manipulador nunca pueda colisionar con el entorno estático en su espacio de trabajo. Con esto nos referimos a la mesa donde está anclado, a las cámaras *Kinect*, y al propio robot, ya que cuando se ha instalado la mano es posible que ésta colisione con otros eslabones del robot.
Tal como se ha mencionado anteriormente, la creación de un buen modelo del robot es muy importante, ya que ROS utiliza ese modelo para diferentes tareas.
Al crear un modelo *URDF* y una cadena cinemática con el asistente de ROS, *Wizard*, se crean internamente ficheros de configuración de extensión `.yaml` muy útiles en diferentes ámbitos.
El fichero *joint_limits.yaml*, por ejemplo, indica los nombres de todas las articulaciones del robot, incluyendo restricciones de velocidad, aceleración y posición.

En primer lugar se crea una escena de la situación actual del robot como si de una foto 3D se tratara, en la que se incluye todo el contenido del modelo *URDF*. Es necesario utilizar dos servicios ROS para crear la escena: *set_planning_scene_diff* y *get_state_validity.*
Se crea una estructura de datos con la información necesaria para realizar las peticiones, incluyendo el nombre y estado de las articulaciones. Para cada solución de cinemática inversa calculada se comprueba que es válida y que está fuera de colisión. Si la nueva posición articular calculada es válida, se le envía al robot para que éste realice la trayectoria, en cambio, si la posición no es válida, es decir, entra en colisión, no se envía al robot, por lo cual no se mueve y se muestra un mensaje al usuario indicándole la posible colisión.

Mediante este procedimiento evitamos cualquier tipo de colisión entre el robot y su entorno estático, por lo que se ayuda en gran manera al teleoperador al poder obviar estas situaciones de riesgo para la realización de la tarea.

No obstante, mediante este procedimiento también se pueden añadir objetos no estáticos, como por ejemplo cuando la mano robótica coge un objeto y lo desplaza a

otro lugar. Durante el desplazamiento el objeto formaría parte de la cadena cinemática y el sistema que hemos programado evitaría colisiones entre el nuevo objeto y el resto del robot previamente definido. También se pueden añadir objetos en tiempo de ejecución utilizando para ello cámaras y programas de percepción.
En la Ilustración 35 se puede apreciar como el robot PR2 detecta colisiones con objetos del entorno y con él mismo.



1. Colisión con otros objetos 2. Respetar límites articulares 3. Colisión con el propio robot

Ilustración 35: Detección de colisiones

4.12 Construcción de Interfaz Gráfica

En teleoperación, la telepresencia es un objetivo de gran importancia si se desean obtener buenos resultados y una alta tasa de éxito en tareas teleoperadas. Por ello nos ha parecido interesante desarrollar una interfaz gráfica con diferentes detalles que muestren el estado del robot y la relación de éste con el háptico.

Tras el estudio de diferentes alternativas de implementación de *GUIs* para *python* como *wxPython, PyGTK* o *Tkinter,* se ha seleccionado *PyQt4*, ya que posee una estructura organizada y completa, con un conjunto de módulos con más de 300 clases y 6000 métodos.

En primer lugar, se ha de definir qué es adecuado representar en la interfaz y que características gráficas se le van a dar. Un usuario que no haya experimentado previamente la teleoperación con este sistema debería interpretar y asimilar fácilmente su funcionamiento. Por ello, se han utilizado iconos e imágenes grandes y descriptivas. Si se coloca el cursor del ratón encima de cualquier elemento de la interfaz se muestra una breve explicación a modo informativo, para guiar al usuario y facilitar su uso.
Buscando la ergonomía para el usuario se ha considerado que solo dispone de una mano libre para realizar tareas como cambiar de modo de ejecución, abrir y cerrar la mano robótica, etc. Pero dichas tareas no se pueden realizar cómodamente con sólo una mano, ya que es necesario enviar comandos mediante consola, navegar con el ratón, distraerse de la tarea que se esta teleoperando durante un largo periodo, …

Como solución se ha optado por automatizar dichas tareas y embeberlas en diferentes teclas del teclado que pueden ser pulsadas por el usuario cuando considere necesario. Éstas están ordenadas de manera precisa para que el usuario no necesite mirar el teclado y perder de vista el robot, consiguiendo, por tanto, una aplicación muy intuitiva.

Al comenzar, la aplicación le pregunta al usuario si desea ejecutar *Reflexxes*, en caso de negativa, posteriormente se puede iniciar mediante la barra espaciadora del teclado.

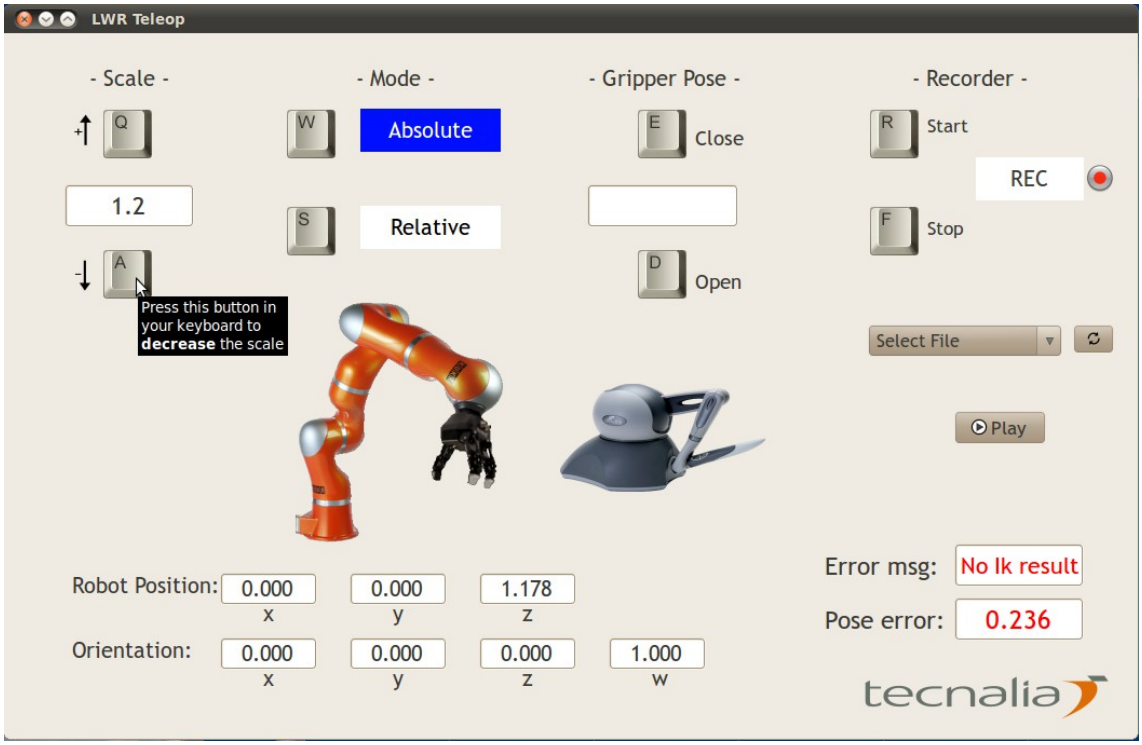


Ilustración 36: Interfaz gráfica de usuario

Comenzando por la izquierda se puede observar el factor de escala que se está aplicando al *Omni* en dicho instante. Puede ser modificado de manera sencilla pulsando las teclas "Q" y "A" del teclado. Incrementando hasta un máximo de 4,0 y decrementando hasta un mínimo de 0,1.

Siguiendo por la derecha, se puede observar el modo de ejecución en el que nos encontramos. Este puede ser absoluto o relativo. Las teclas destinadas para el cambio se modo son "W" y "S", mediante un panel que cambia de color en función del modo podemos conocer en qué modo estamos teleoperando. Para automatizar esta tarea ha sido necesario crear otro nodo ROS que se encarga de lanzar y finalizar los modos independientemente en función de la tecla pulsada.

Seguidamente se observa el estado de la mano robótica. Para manipularla es necesario pulsar las teclas "E" y "D" con las que se le ordena a la mano abrir o cerrar, respectivamente. La amplitud máxima de la mano es una valor de 255 y la mano

completamente cerrada se corresponde con el valor 0. Internamente, se envían comandos al controlador de la mano vía Ethernet.

En la parte derecha de la interfaz se observa el modo de aprendizaje del manipulador. Esta parte esta muy elaborada en lo que control de errores se refiere. Las teclas de teclado correspondientes son “R” y “F” las cuales se pueden recordar mediante técnicas nemotécnicas ya que la R se puede asociar con la palabra Rec que se utiliza en el vocabulario anglosajón para definir “grabando”, y la F con la palabra “Fin”. Además esta tecla tiene otra utilidad que es finalizar el modo de ejecución, por lo que la teleoperación también finalizaría hasta seleccionar un nuevo modo.

En el momento de pulsar la tecla R de grabación, aparece una nueva ventana en el que se debe indicar el nombre de la tarea que se va a realizar. En caso de no ingresar ningún nombre la tarea pasará a llamarse *default.npz*. Tras ello, los movimientos teleoperados que realice el robot serán almacenados en un fichero. Una vez finalizado el proceso se ha de pulsar la tecla F, con lo que finalizará la grabación y el proceso de teleoperación.

En la parte derecha de la Ilustración 36, se aprecia un botón de refresco, un desplegable y un botón con la palabra *Play* en su interior.
El desplegable incluye todas las tareas de teleoperación que se han realizado y el robot ha aprendido.
El botón de refresco se utiliza para incluir la última tarea aprendida por el robot y actualizar la lista desplegable.
El botón Play ejecuta la tarea seleccionada. En caso de no seleccionar ninguna, muestra un mensaje de error. Durante la ejecución de la tarea almacenada la interfaz se bloquea para que no se puedan realizar cambios en el robot, por lo tanto, si se utiliza el *Omni* durante este periodo las posiciones deseadas no le serán comunicadas al robot.

En el centro de la interfaz se observa una imagen del manipulador *Lwr* con la mano incorporada y otra del *Phantom Omni*.

La parte inferior de la interfaz aporta información al usuario acerca del robot, ya que muestra la posición y la orientación del último eslabón del robot. De esta manera se puede recrear mentalmente un posicionamiento espacial del robot. También se ha incorporado un cuadro de texto que indica el error o desfase existente entre la posición cartesiana del *Lwr* y del *Omni*. Se trata de un control muy útil para conocer si el manipulador está respondiendo bien a las ordenes del operador.

En otro cuadro de texto se incluyen mensajes de alerta y de error que se puedan producir en la aplicación. Estos se muestran en color rojo para que llamen la atención del operador.

4.13 Control del Manipulador en Posición

Los sentidos más utilizados en teleoperación son la vista y el tacto. Con el primero podemos sacar conclusiones rápidas de la localización del robot, de la distancia a nuestro objetivo, del entorno de trabajo, etc. Pero, en ocasiones, no es suficiente el uso de sólo este sentido, ya que no siempre se tiene una vista óptima del entorno, y es muy difícil determinar si el acoplamiento del robot sobre el háptico es absoluto o se producen retardos o desfases.

Por ello, se ha implementado otro tipo de opción sensorial: el tacto. Mediante éste podemos conocer si el manipulador sigue fielmente las ordenes del háptico, si el robot ha colisionado con algún objeto o está sufriendo fuerzas o pares que no se consideraban.

Existen diversos factores por los cuales el manipulador puede no seguir correctamente las posiciones dictadas por el usuario mediante el *Omni*:

- Se produce un retardo en el envío de posiciones o pérdida de datos por motivos de la red Ethernet.
- *Reflexxes* tarda más de lo esperado en interpolar los datos recibidos.
- *KDL* no es capaz de hallar una solución al problema cinemático inverso/directo.
- El operador desplaza el *Omni* a un velocidad que supera los limites de velocidad máxima del *Lwr*.
- El manipulador ha colisionado con algún objeto o se encuentra fuera de sus límites máximos articulares.

Como primer objetivo se pretende que en el *Phantom Omni* se sienta una fuerza dinámica que dependa del error o desfase acaecido entre la posición deseada del *Omni* y la posición actual de *Lwr*. Para ello, se debe conocer la posición cartesiana del último eslabón de la cadena cinemática de ambos robots. Esto se ha realizado mediante la cinemática directa del manipulador.

Seguido, se obtiene la diferencia entre las componentes x, y, z de cada uno y se aplica una ganancia. El resultado obtenido se puede interpretar como una fuerza que se debe aplicar a los motores del *Omni* para que lo sienta el usuario.

La fuerza resultante debe estar comprendida entre unos máximos y mínimos establecidos y acordes a las características físicas del háptico. Ya que aplicar una fuerza resultante al háptico mayor de la que puede soportar, aunque sea durante un instante muy pequeño de tiempo, puede dañar los actuadores e incluso los sensores.
Por ello se han realizado pruebas de estabilidad del *Omni* y de fuerzas máximas soportadas. En el siguiente apartado se explica con mayor claridad.

Por lo tanto, mediante software se controla la fuerza que se envía al *Omni* en cada eje. Ésta puede ser variable dependiendo de la fuerza que se desee sentir en el *Omni* alterando la ganancia aplicada.

En el modo relativo de ejecución sólo se sentirán las fuerzas de control de posición durante el periodo de tiempo que se pulse el botón del *Omni* (hombre-muerto). La diferencia de posición será relativa al primer instante en el que se pulse el dicho botón. La fuerza resultante cuando el botón no esté pulsado será cero.

En situaciones ideales, si en un momento dado la posición del *Lwr* no es consecuente con la del *Omni* se debería sentir una fuerza infinita en la mano del operador, como si de una pared se tratara.
En la práctica, se pretende aplicar una fuerza lo suficientemente grande como para que el operador sienta que algo está ocurriendo y el dispositivo háptico no se dañe.

4.14 Retroalimentación o FeedBack de Fuerzas

Tras conseguir el objetivo de implementar un controlador en posición entre *Lwr* y *Omni*, es interesante detectar cuándo el manipulador colisiona contra algún objeto o está sufriendo pares o fuerzas en sus articulaciones.
En aplicaciones reales donde el robot se emplea para manipular objetos, es de gran interés que el operador sienta en el dispositivo háptico las fuerzas proporcionales que está sintiendo el robot en cada momento. Esto puede evitar que se sostengan pesos superiores a los máximos del robot y evitar configuraciones articulares dañinas debido a dichos pesos.
Otra propiedad es que el operador sienta si el robot colisiona con objetos del entorno. Éste puede determinar si la fuerza que está ejerciendo sobre un objeto es suficiente o, en cambio, es excesiva.

El manipulador *Lwr* posee un sensor de par por cada una de sus articulaciones. Los datos aportados por cada uno de ellos pueden ser visualizados e interpretados.
Existe un *topic* denominado */cartesian_wrench* que a partir de los pares obtenidos en cada uno de los sensores calcula la fuerza resultante en la última articulación.

Existe una relación directa entre par y fuerza. Para explicarla de manera gráfica se va a utilizar el siguiente ejemplo.
Se supone que un ciclista quiere comenzar su marcha, para ello debe realizar una fuerza determinada sobre los pedales para poder avanzar. Esa fuerza, multiplicada por la distancia de los pedales al eje donde está alojado el plato, produce un momento de fuerza sobre el eje, o par motor.

Póngase por caso que el ciclista en cuestión hace una fuerza F sobre los pedales y que éstos se encuentran a una distancia r del eje del plato. Esta fuerza genera un momento de fuerza, llamado par motor:

$$\tau = F \cdot r$$

Al igual que ocurre con las velocidades articulares y las velocidades en el espacio cartesiano, los pares articulares y las fuerzas/pares en el último elemento del robot también se relacionan a través de la matriz jacobiana.

$$\tau = \mathrm{J(q)}^{\mathrm{T}} \cdot F \quad ; \qquad F = \begin{pmatrix} n \\ f \end{pmatrix}$$

Donde τ es un vector *n-dimensional* de las fuerzas/pares aplicados en un manipulador de n grados de libertad. F es el vector de fuerza espacial en el cual n y f son vectores de par y fuerza, respectivamente, aplicados al último efector. Ambos expresados en el marco de referencia con respecto a la que también se expresa la jacobiana. Esto se da en el caso estático. En el caso dinámico entran también en juego los pares en las articulaciones que provienen del movimiento del robot. En todo caso, suponiendo que se tienen los pares con la componente dinámica filtrada, la anterior fórmula también se aplica.

Tras entender como se obtiene la fuerza en el último efector, podemos interpretar los datos recogidos en el *topic* para asignar dicha fuerza al *Omni*, siempre respetando la fuerza máxima soportada por el háptico.

Realizando pruebas en el robot real se ha podido observar que si se aplica una fuerza controlada hacia cualquier dirección, el *Omni* se comporta de igual manera desplazándose en el sentido de la fuerza aplicada. El operador se da cuenta perfectamente del sentido y la intensidad de dicha fuerza.

4.15 Pruebas de contacto y control con Phantom Omni

La teleoperación se realiza en entornos no estructurados por lo que, en ocasiones, es posible que el manipulador interactúe con objetos de manera esperada o inesperada.
Por tanto la principal motivación para la realización de estas pruebas radica en la propia interacción con objetos. Existen diferentes tipos de objetos clasificados según su dureza, objetos rígidos, elásticos, plásticos, moldeables, etc. Para este proyecto sólo se van a tener en consideración los objetos rígidos, concretamente paredes virtuales.
La ecuación que representa la mayoría de los cuerpos rígidos es la siguiente:

$$F = K \cdot x$$

donde F es la fuerza resultante de aplicar la ganancia K a una posición x conocida.

La ecuación anterior no aporta una sensación adecuada al tacto en teleoperación ya que no tiene en consideración el modo en el que se incide en un objeto, por lo que se debe incluir la velocidad con la que se acerca al límite del objeto multiplicado por un factor *b* de amortiguamiento. Este nuevo componente amortigua la fuerza resultante y el operador logra una sensación más realista de la rigidez de los objetos.

$$F = kx + b\dot{x}$$

El cambio de rigidez producido durante el movimiento en el espacio vacío y el movimiento en contacto con objetos rígidos produce inestabilidad.
Se pretende estudiar hasta dónde se puede controlar esa inestabilidad por lo que se han desarrollado pruebas de contacto y de estabilidad en el *Omni*.

En primer lugar se ha creado una pared virtual a lo largo de toda la vertical. En una primera aproximación se ha utilizado la ecuación $F = K \cdot x$ para modelarla, donde x es la posición del *Omni* y K es la ganancia que queremos estimar.
Damos a la K de diferentes valores para estudiar cómo se comporta el *Omni* en las proximidades de la pared virtual.

Para un valor de K = 1,0 la fuerza resultante no se nota en exceso por el usuario. Esto es, la pared virtual colocada en la vertical (y = 4,0) se sobrepasa con facilidad.

Para un valor de K = 1,5 la sensación de pared empieza a acercarse a lo buscado. Con mayor impedancia que la anterior, pero sin ser una pared robusta. Con un poco de presión la pared es sobrepasada.

Para un valor de K = 2,0 la pared toma un carácter sólido a efectos del sentido del tacto. Cuesta mucho sobrepasarla por lo que se toma como una buena ganancia.

Para valores de K superiores a 2,0 en las proximidades de la pared virtual el *Omni* comienza a oscilar rebotando con la pared, mostrando síntomas de inestabilidad. Esta vibración aporta mucha información ya que nos indica qué valor de ganancia no se puede sobrepasar. Tras esta prueba es necesario reiniciar el *Omni* ya que quedan diferentes fuerzas residuales que hacen imposible el manejo del háptico.

Con el uso de esta prueba de contacto se calcula empíricamente la ganancia óptima del controlador.
Tras concluir la ganancia óptima aplicable en el *Omni*, se han llevado estas pruebas al *Lwr*, con el fin de estudiar el comportamiento del manipulador cuando es teleoperado con una pared virtual y diferentes ganancias.
Por lo que se han repetido las pruebas con las ganancias anteriores durante la teleoperación.
Los resultados obtenidos se han monitorizado mediante un *script* y se han generado gráficas para su estudio y comprensión.
El *script* del Código 16 se basa en la obtención de los mensajes de los *topics* que nos aportan información relevante en el ensayo.

Posición articular y cartesiana del robot y posición cartesiana del *Omni*.

```
#!/bin/bash
rostopic echo /joint_states >
~/Desktop/control_omni_lwr/Ensayos/Segundo_Ensayo/lwr_articular.txt&

rostopic echo /get_fd_kinematic >
~/Desktop/control_omni_lwr/Ensayos/Segundo_Ensayo/lwr_cartesian.txt&

rostopic echo /teleop_pose >
~/Desktop/control_omni_lwr/Ensayos/Segundo_Ensayo/omni_pose.txt&
wait
```

Código 16: Script para obtener información del Omni y Lwr

En la Ilustración 37 se pueden observar tres gráficas, una por cada eje cartesiano.
El eje de abscisas corresponde con el tiempo medido en segundos y el eje de ordenadas con la posición del *Omni* en cada eje, x, y , z :

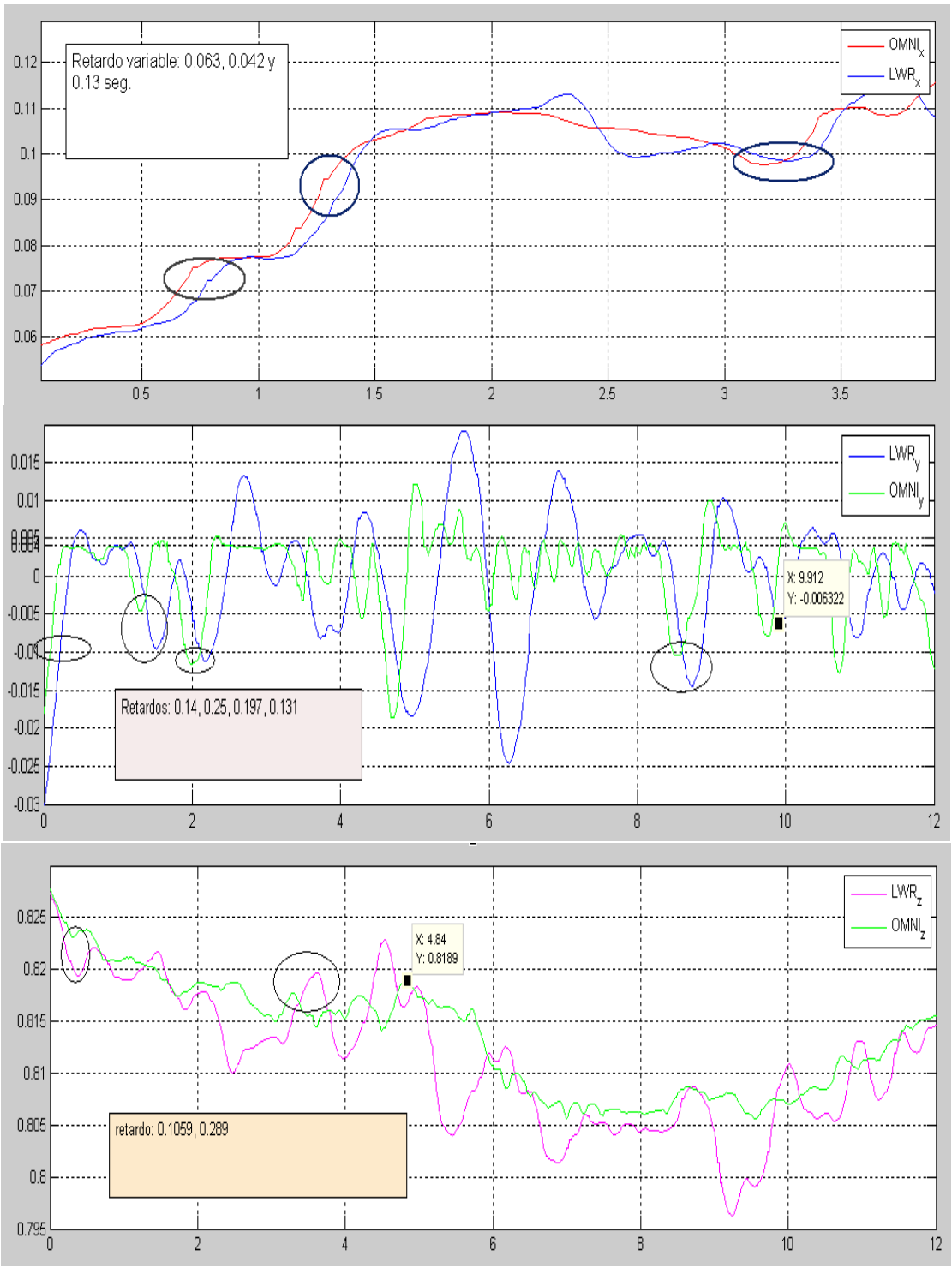


Ilustración 37: Resultados de las pruebas de contacto y control con Phantom Omni

Observando las gráficas se pueden apreciar dos problemas:

1. Retardo variable

   Se observa en la primera y segunda gráfica. Puede ocurrir por diversos factores:

   - Retardos en la comunicación
   - Tiempo variable y no controlado del cálculo cinemático inverso
   - Velocidad del robot limitada y menor que la del *Omni*

   Solución:

   - Utilizar equipos en tiempo real
   - Nuevo solucionador de cinemática inversa más rápido y eficiente (p.ej. *IkFast*)
   - Alcanzar velocidad máxima del Robot o disminuir la del *Omni*

2. Problemas de sincronización

   - Cuando el robot repite una trayectoria dictada en modo relativo, éste se mueve de manera no continuada, como a saltos y con pequeñas paradas

   Solución:

   - Filtrado de trayectorias mediante técnicas de filtrado tipo splines cúbicos
   - Examinar los datos recogidos en los ficheros de aprendizaje

Para aplicaciones reales en la industria es necesario garantizar unos tiempos de respuesta fijos y controlados. Por ello es interesante el estudio de otras técnicas para calcular cinemáticas como puede ser *IkFast* de *OpenRave* [Rosen Diankov & Kei Okada, 2011] que mediante técnicas analíticas mejora y mucho los tiempos de cálculo actuales.
En este proyecto se ha comenzado ha estudiar la técnica de *OpenRave* pero no se ha llegado a implantar en el sistema debido a que se desconoce sobre la completa compatibilidad con ROS y los modelados de robots como *URDF*.

Se ha aumentado la velocidad del *Lwr* hasta su máximo permitido por el fabricante con el fin de obtener una respuesta mejor.

Con respecto al segundo problema se ha pensado en añadir técnicas de suavizado de trayectorias, también conocido como *smoothing* pero esto no ha resultado ya que las trayectorias almacenadas incluyen todos los puntos por donde ha pasado el *Omni* y es obligatorio pasar por cada uno de ellos, ya que si por ejemplo se ha enseñado una

trayectoria que dibuja un cuadrado, ésta no se puede redondear, ni evitar ciertos puntos para suavizarla ya que el robot podría entrar en colisión con objetos que antes se esquivaban.
El *smoothing* tiene validez para generar trayectorias en medios estructurados o vigilados, donde las posibles colisiones con el entorno cambiante son controladas.
Por lo que, tras su estudio, se desestimó esta técnica.

Otra solución se ha basado en estudiar los datos almacenados en los ficheros de aprendizaje. Tras ello nos hemos dado cuenta que se almacenan todos los datos enviados y procesados por el robot, incluyendo los instantes en los que el operador no pulsa el botón del *Omni* con el fin de volver a colocar el *Omni* en una posición más cómoda para el operador o para alcanzar una mayor distancia. Durante esas paradas el robot no se mueve.
Una vez filtradas las paradas el manipulador ser mueve con una mayor soltura y dinamismo.

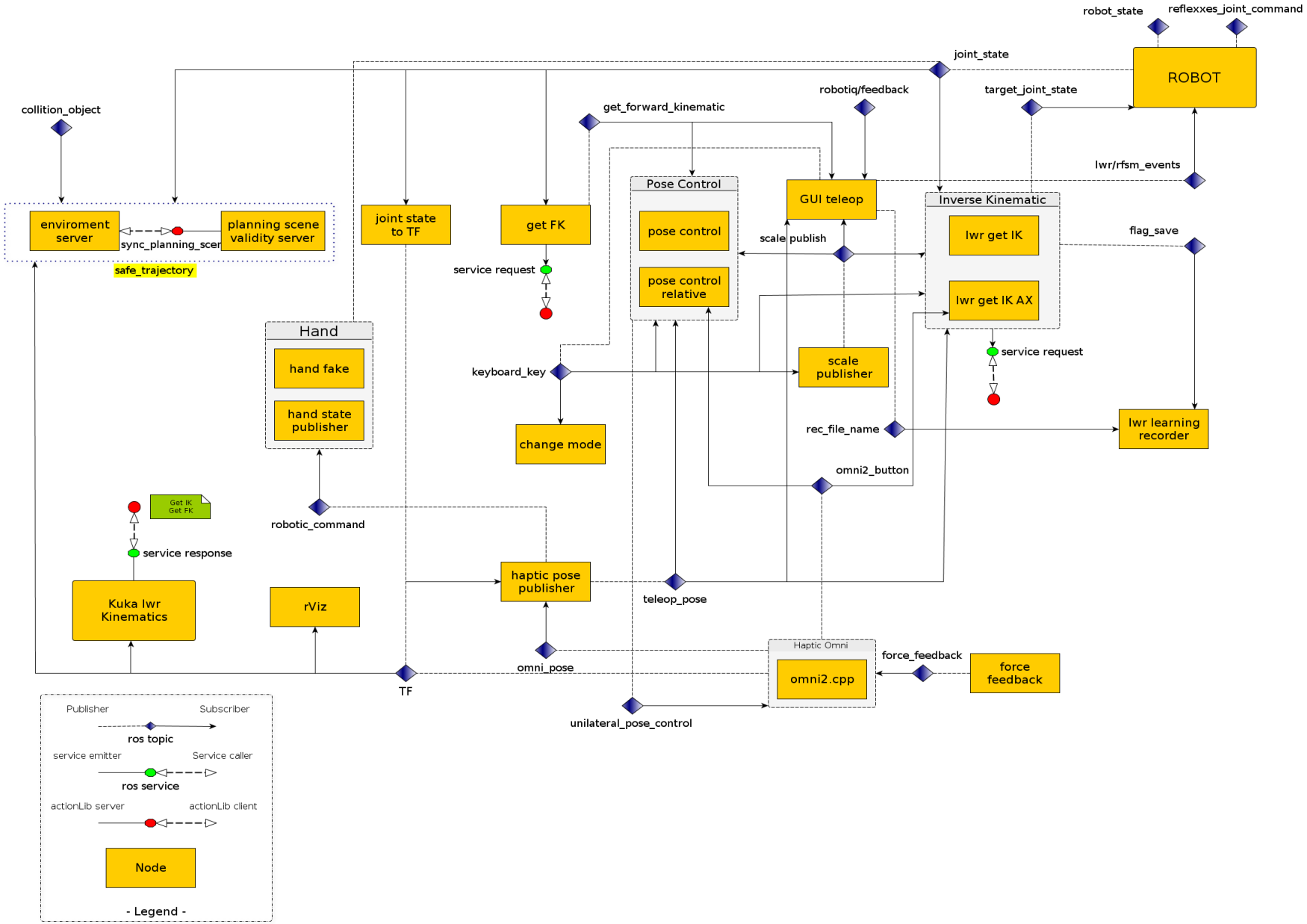
Parte V

VISIÓN GLOBAL
Y
CONCLUSIONES

5 Visión Global del Proyecto

Tras la finalización de todos los hitos y tareas propuestas, es interesante abstraerse para observar el proyecto en su totalidad.
Como primera aproximación se pretendía teleoperar de modo sencillo el manipulador *Lwr* mediante el dispositivo háptico *Phantom Omni*. Para ello ha sido necesario instalar los dispositivos y entender su funcionamiento.
Por otra parte, si nos referimos al apartado software, nos hemos ayudado de herramientas que facilitan la programación de robots y manipuladores industriales, como son *ROS, Orocos, Reflexxes, etc.* También nos hemos ayudado de un modelo del manipulador y del entorno, que utilizamos mediante simulación para realizar pruebas y generar nuevas demos. Todas estas pruebas realizadas mediante simulación han supuesto un avance y una ayuda para después trasladarlas al robot real. También han sido necesarias para darnos cuenta de la necesidad de añadir mejoras al sistema, como son los dos modos de teleoperación, absoluto y relativo, la posibilidad de cambiar durante la ejecución el factor de escala aplicado a las posiciones del *Omni*, para de ese modo alcanzar posiciones más lejanas o tener una mayor precisión en tareas de espacio reducido. Otras mejoras han sido la capacidad autónoma del manipulador para no colisionar con obstáculos definidos en el entorno de trabajo y la capacidad de aprendizaje de tareas que han sido fácilmente ilustradas por un usuario sin conocimientos de programación.
Con el fin de ayudar y asistir al usuario en el proceso de teleoperación se ha diseñado y una interfaz gráfica intuitiva y que aporta mucha información sobre el estado del manipulador. Como asistencia al sentido de la vista, se le proporciona al usuario información sensorial mediante el control de posición del manipulador que se basa en el envío de fuerzas al *Omni* en caso de que el manipulador actúe de forma diferente a la esperada. El usuario también es capaz de sentir lo que captan los sensores de par que posee *Lwr*, siendo esto de gran importancia para detectar posibles choques del manipulador.

La siguiente ilustración muestra un esquema general del proyecto, basado en los nodos que se ejecutan durante el proceso de teleoperación. Se ha estructurado en cuatro grandes grupos de nodos, siento estos, *Phantom Omni*, Mano robótica, cinemática inversa, y robot, que se refiere al manipulador *Lwr* [Ilustración 38].
El resto de los nodos se corresponden con *rviz*, con los nodos de aprendizaje de tareas, con la interfaz gráfica, etc.


robot_state
reflexxes_joint_command
ROBOT
joint_state
target_joint_state
robotiq/feedback
get_forward_kinematic
collition_object
lwr/rfsm_events
Pose Control
GUI teleop
Inverse Kinematic
enviroment server
sync_planning_scen
planning scene validity server
safe_trajectory
joint state to TF
get FK
pose control
pose control relative
scale publish
lwr get IK
lwr get IK AX
flag_save
service request
Hand
hand fake
hand state publisher
keyboard_key
scale publisher
service request
change mode
rec_file_name
lwr learning recorder
omni2_button
Get IK
Get FK
robotic_command
service response
haptic pose publisher
Kuka lwr Kinematics
rViz
teleop_pose
Haptic Omni
omni2.cpp
force_feedback
force feedback
omni_pose
TF
unilateral_pose_control
Publisher
Subscriber
ros topic
service emitter
Service caller
ros service
actionLib server
actionLib client
Node
- Legend -

Con el fin de clarificar el bloque denominado ROBOT del esquema, se muestra la Ilustración 38 donde se observa, con más detalle, los mensajes, *topics* y nodos que intervienen en el funcionamiento del manipulador.

Como se ha comentado, FRI [Cap 3.7.2] realiza la comunicación entre el controlador *KRC* y el computador, en la Ilustración 38 representado como *lwr_fri.* En su interior posee diferentes nodos encargados de interceptar, tratar y reenviar mensajes entre las dos partes. Los *topics* más utilizados a lo largo del proyecto han sido:

/target_joint_state	El usuario envía las posiciones articulares deseadas, éstas van a *reflexxes* para ser interpoladas y enviadas al controlador mediante el topic *JointPositionCommand*.
/joint_state	Muestra el estado de las articulaciones del manipulador, posición, velocidad y esfuerzo
/cartesian_position	Indica la posición cartesiana de los todos los eslabones de la cadena cinemática
/cartesian_wrench	Indica el par que está sufriendo el último efector del manipulador
/robot_state	Resumen del estado general del robot, temperatura, posiciones

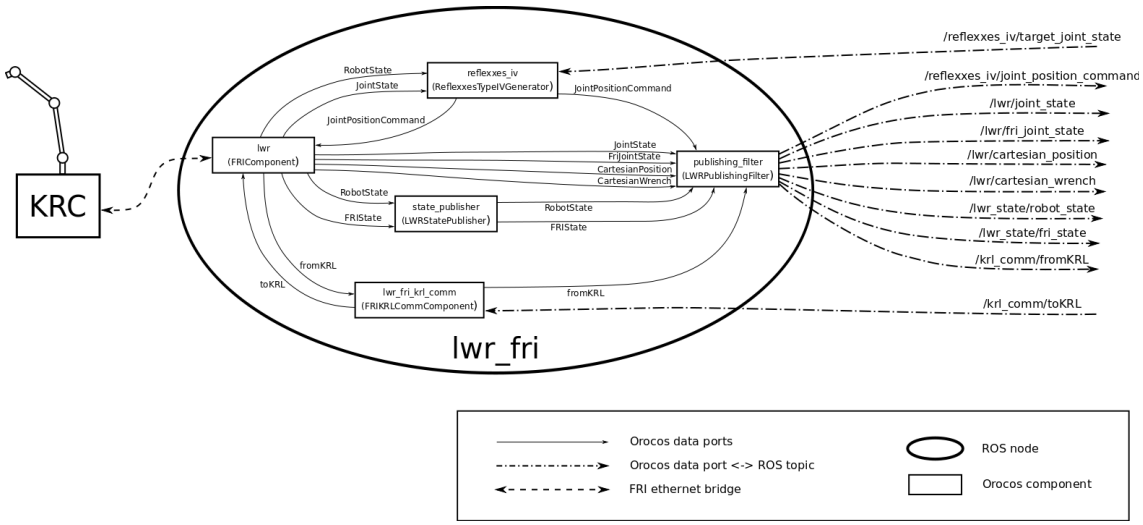


Ilustración 38: Comunicación interna entre KRC y computador

6 Conclusiones

Según se planteó el proyecto en sus orígenes se debía desarrollar la tecnología necesaria para el controlar y teleoperar un manipulador *KUKA Lwr* mediante un dispositivo háptico *Phantom Omni* utilizando para ello herramientas software como *ROS* y *Orocos.*
Voy a realizar una valoración a dos niveles:

A nivel de objetivos, el proyecto lo considero satisfactorio ya que se han cumplido todos los objetivos impuestos en la planificación del proyecto, superando los mínimos en muchos aspectos y aportando valor añadido. Incluso se han generado nuevos objetivos durante la fase de desarrollo que han supuesto mejoras significativas en el sistema, como pueden ser la interfaz gráfica que asiste al operador aportándole información relevante del robot, el control de posición del manipulador, que logra una gran telepresencia, etc.

En este momento, un usuario sin conocimientos de robótica puede ser capaz de realizar tareas complejas teleoperando el manipulador y obteniendo buenos resultados de precisión, rapidez y eficiencia.

A nivel personal considero el proyecto como una gran oportunidad de aprendizaje debido a toda la infraestructura y el material del que he dispuesto, como son los robots *KUKA Lwr, Kadawa Hiro, Fanuc*, el háptico *Omni*, etc.
Debido también al ambiente de trabajo y de investigación en el que me he encontrado. He tenido la oportunidad de conocer en profundidad herramientas software como *ROS* y *Orocos* que facilitan la labor del programador.

Principalmente me ha servido de fuerte motivación para continuar mis estudios en el campo de la robótica y de la investigación con vistas a realizar un doctorado en el próximo año.

7 Lineas futuras de Investigación

Durante el desarrollo del proyecto han surgido diversas lineas de investigación de gran interés que pueden dar lugar a futuros proyectos dentro del campo de la robótica. Algunas de estas lineas de han surgido como respuesta o apoyo a los objetivos planteados en el proyectos, otras, en cambio, se deben a ideas interesantes propuestas por todos los que investigamos.

Un objetivo de la corporación Tecnalia se basa en reutilizar los conocimientos y resultados adquiridos en este proyecto para teleoperar otros manipuladores, estableciendo principal interés en el área de los robots industriales.
Durante unas semanas hemos estado investigando y desarrollando de forma paralela a los objetivos de este proyecto la forma de teleoperar un robot industrial *Fanuc* mediante el dispositivo *Phantom Omni.*
En este caso la mayor diferencia radica en que es el propio controlador del robot quien calcula la cinemática directa e inversa del manipulador. Por lo que se deben enviar al robot las posiciones cartesianas deseadas indicadas de forma manual por el háptico *Omni*.
Otra gran diferencia reside en la comunicación ya que en *Fanuc* se realiza mediante *ModBus.* Para ello se ha realizado un estudio de la librería *libModbus* compatible con Linux.
Como último hito en esta línea se ha establecido un protocolo de comunicación entre el PC que controla el *Omni* y el controlador de *Fanuc,* ya que hay que establecer cuándo una posición enviada ha sido recibida y/o ejecutada por el manipulador. Para ello y para el control de posiciones negativas respecto a una referencia fija, se utilizan bits de control enviados en posiciones de memoria concretos del controlador de *Fanuc*.

A partir de una idea ha surgido la linea de investigación basada en la teleoperación de un robot mediante el mando a distancia que utiliza la video-consola *Wii* de *Nintendo*, *WiiMote*.
Tras el estudio de esta tecnología, se ha comprobado que el mando *WiiMote* obtiene su posición en el espacio gracias a una barra denominada *SensorBar* que posee *LEDs* en cada uno de sus lados que emiten luz infrarroja. Ésta luz es recogida por el *WiiMote,* y mediante unos cálculos matemáticos conoce su posición en el espacio.
Como alternativa a la *SensorBar* se puede colocar en su defecto dos velas encendidas ya que también aportan luz infrarroja, no obstante, se conoce que a mayor número de

emisores infrarrojos mayor será la precisión de posicionamiento del mando. Por lo que se ha decido construir una *SensorBar* casera.
Tras construirla colocando *LEDs* infrarrojos en serie sobre una placa de metacrilato y alimentándolos con un conjunto de pilas, y conectando el *WiiMote* a un PC mediante *Bluetooth* y un software especifico, se ha logrado obtener las posiciones del mando en el espacio. Las orientaciones del mando se obtienes mediante acelerómetros que trae el mando instalados.
El problema reside en el momento de obtener las posiciones, ya que en los movimientos de izquierda-derecha y arriba-abajo, el mando nos indica las posiciones de manera acertada en un rango definido. En cambio, en los movimientos hacia delante y hacia atrás, los resultados no son los esperados ya que el rango es otro muy diferente al anterior y los datos no son estables.

Como alternativa a la teleoperación mediante el mando a distancia *WiiMote,* se pretende hacerlo utilizando el propio cuerpo humano. Para el caso de teleoperar el manipulador *KUKA Lwr,* como es antropomórfico, se utilizaría un brazo humano para enviar las posiciones deseadas. Todo gracias a cámaras dedicadas que capten las articulaciones del brazo y el movimiento de los dedos.
En una primera aproximación, mediante la cámara *Kinect* de *Microsoft* y utilizando el software *Openni Tracker* de ROS que se encarga de obtener el *TF* de un cuerpo humano basándose en sus movimientos, se pretenden obtener datos de posiciones para ser enviados al manipulador.
En la Ilustración 39 se observa como la cámara *Kinect* junto con el software *Openni Tracker* detecta e interpreta el cuerpo humano.

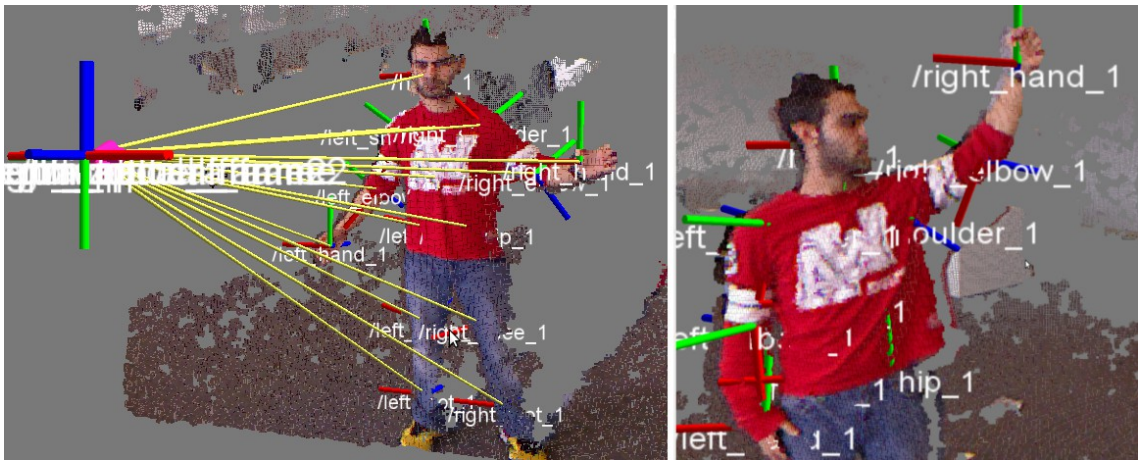


Ilustración 39: Percepción del ser humano. Openni Tracker y Microsoft Kinect

Como se ha comentado en capítulos anteriores, el manipulador *Lwr* se utiliza en otros proyectos de investigación, entre los que destaca un proyecto de colaboración hombre-robot que se basa, entre otras cosas, en el que el robot sea capaz de percibir lo que sucede dentro de su espacio de trabajo y, por consiguiente, tenga la capacidad de esquivar y evadir objetos o personas que se sitúen dentro de dicho espacio. Esto lo

realiza gracias a la cámara *Kinect* de *Microsoft* y a un software dedicado que es capaz de generar nubes de puntos alrededor de objetos en una distancia acotada. El robot sigue trayectorias teniendo en cuenta esas nubes de puntos y evadiendo autónomamente objetos que se sitúen en su trayectoria generando para ello nuevas trayectorias.
Se ha planteado que es interesante unificar ambos proyectos ya que al robot se le pueden enseñar diferentes tareas mediante la teleoperación que después el robot puede reproducir de manera segura, ya que es capaz de detectar cualquier movimiento dentro de su espacio de trabajo.

8 Bibliografía

Conocimiento General:
Libro: Bruno Siciliano, Oussama Khatib , “Handbook of Robotics” - Springer
Libro: Anibal Ollero Baturone, “Robótica Manipuladores y Robots móviles”-Marcombo

ROS:
http://www.bipedolandia.es/t1751-que-es-ros#11663
www.ros.org
Paper: Morgan Quigley “ROS: an open-source Robot Operating System ”

Matrices de Rotación:
http://la-mecanica-cuantica.blogspot.com.es/2009/08/la-matriz-generadora-de-rotacion.html
http://answers.ros.org/question/10124/relative-rotation-between-two-quaternions/

Interfaz Gráfica:
http://zetcode.com/tutorials/pyqt4/introduction/
http://www.slideshare.net/jpadillaa/primeros-pasos-en-pyqt4#btnNext

Cinemáticas:
http://www.disclab.ua.es/robolab/EJS2/RRR_Intro_3.html
http://geus.wordpress.com/2010/09/28/robotica-inverse-kinematics-ik-en-c-con-orocos/
http://geus.wordpress.com/2010/03/31/robotica-forward-kinematics-en-python/
Geometría y Cinemática , CIN_ROB.pdf
Oscar Altuzarra , “Theoretical Kinematics ”, UPV-EHU

Reflexxes:
http://www.reflexxes.com/
Paper: Torsten Kroeger “The Reflexxes Motion Libraries ”

Métodos de aprendizaje:
Paper: Yusuke MAEDA, Tatsuya USHIODA y Satoshi MAKITA , Easy Robot Programming for Industrial Manipulators by Manual Volume Sweeping

KUKA Lightweight :
Libro: KUKA Roboter GmbH , Operating and Programming Instructions for System Integrators
Paper: Rainer Bischoff , Alin Albu-Schäffer , "The KUKA-DLR Lightweight Robot arm – a new reference platform for robotics research and manufacturing "

Orocos:
http://www.orocos.org/
Herman Bruyninckx, Peter Soetens , "The OROCOS Project "

Hápticos:
Paper: Marcos García , "Técnicas y dispositivos de realidad virtual "
Paper: César Peña, Rafael Aracil , "Sistema háptico de asistencia para la teleoperación de robots"

9 Anexo A

9.1 Denavit-Hartenberg

Denavit y Hartenberg propusieron un método sistemático para descubrir y representar la geometría espacial de los elementos de una cadena cinemática, y en particular de un robot, con respecto a un sistema de referencia fijo. Este método utiliza una matriz de transformación homogénea para descubrir la relación espacial entre dos elementos rígidos adyacentes.

9.1.1 Parámetros

Definen el paso de un sistema de referencia asociado a una articulación al siguiente.
Sólo dependen de las características geométricas de cada eslabón y de las articulaciones que le unen con el anterior y siguiente (no dependen de la posición del robot)
Definen las matrices A que permiten el paso de un sistema de referencia asociado a una articulación al siguiente y, por tanto, definen las matrices T .
Son cuatro:

– Dos ángulos (θ_i, α_i)
– Dos distancias (d_i, a_i)

θ_i: Es el ángulo que forman los ejes x_{i-1} y x_i medido en un plano perpendicular al eje z_{i-1}, utilizando la regla de la mano derecha. Se trata de un parámetro variable en articulaciones giratorias.

d_i: Es la distancia a lo largo del eje z_{i-1} desde el origen del sistema de coordenadas (i-1)-ésimo hasta la intersección del eje z_{i-1} con el eje x_i. Se trata de un parámetro variable en articulaciones prismáticas.

a_i: Es la distancia a lo largo del eje x_i que va desde la intersección del eje z_{i-1} con el eje x_i hasta el origen del sistema i-ésimo, en el caso de articulaciones giratorias.

En el caso de articulaciones prismáticas, se calcula como la distancia más corta entre los ejes z_{i-1} y z_i.

α_i: Es el ángulo de separación del eje z_{i-1} y el eje z_i, medido en un plano perpendicular al eje x_i, utilizando la regla de la mano derecha .

Transformaciones básicas de paso de eslabón:

- Rotación alrededor del eje z_{i-1} un ángulo θ_i
- Traslación a lo largo de z_{i-1} una distancia d_i ; vector d_i $(0,0,d_i)$
- Traslación a lo largo de x_i una distancia a_i ; vector a_i $(0,0,a_i)$
- Rotación alrededor del eje x_i un ángulo α_i

$$ {}^{i-1}\mathbf{A}_i = \mathbf{T}(\mathbf{z},\theta_i)\,\mathbf{T}(0,0,d_i)\,\mathbf{T}(a_i,0,0)\,\mathbf{T}(\mathbf{x},\alpha_i) $$

$$ {}^{i-1}\mathbf{A}_i = \begin{bmatrix} C\theta_i & -S\theta_i & 0 & 0 \\ S\theta_i & C\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\alpha_i & -S\alpha_i & 0 \\ 0 & S\alpha_i & C\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = $$

$$ = \begin{bmatrix} C\theta_i & -C\alpha_i S\theta_i & S\alpha_i S\theta_i & a_i C\theta_i \\ S\theta_i & C\alpha_i C\theta_i & -S\alpha_i C\theta_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

9.1.2 Algoritmo

DH **1**.- Numerar los eslabones comenzando con 1 (primer eslabón móvil de la cadena) y acabando con n (último eslabón móvil). Se numerará como eslabón 0 a la base fija del robot.

DH **2**.- Numerar cada articulación comenzando por 1 (la correspondiente al primer grado de libertad) y acabando en n .

DH **3**.- Localizar el eje de cada articulación. Si ésta es rotativa, el eje será su propio eje de giro. Si es prismática, será el eje a lo largo del cual se produce el desplazamiento.

DH **4**.- Para $i \in [0,\ldots,n-1]$, situar el eje z_i sobre el eje de la articulación (i+1).

DH **5**.- Situar el origen del sistema de la base $\{S_0\}$ en cualquier punto del eje z_0. Los ejes x_0 e y_0 se situarán de modo que formen un sistema dextrógiro con z_0.

DH **6**.- Para $i \in [1, \ldots, n-1]$, situar el sistema $\{S_i\}$ (solidario al eslabón i) en la intersección del eje z_i con la línea normal común a z_{i-1} y z_i. Si ambos ejes se cortasen se situaría $\{S_i\}$ en el punto de corte. Si fuesen paralelos $\{S_i\}$ se situaría en la articulación i+1 .

DH **7**.- Situar x_i en la línea normal común a z_{i-1} y z_i.

DH **8**.- Situar y_i de modo que forme un sistema dextrógiro con x_i y z_i.
DH **9**.- Situar el sistema $\{S_n\}$ en el extremo del robot de modo que z_n coincida con la dirección de z_{n-1} y x_n sea normal a z_{n-1} y z_n.

DH **10**.- Obtener θ_i como el ángulo que hay que girar en torno a z_{i-1} para que x_{i-1} y x_i sean paralelos.

DH **11**.- Obtener d_i como la distancia, medida a lo largo de z_{i-1}, que habría que desplazar $\{S_{i-1}\}$ para que x_i y x_{i-1} quedasen alineados.

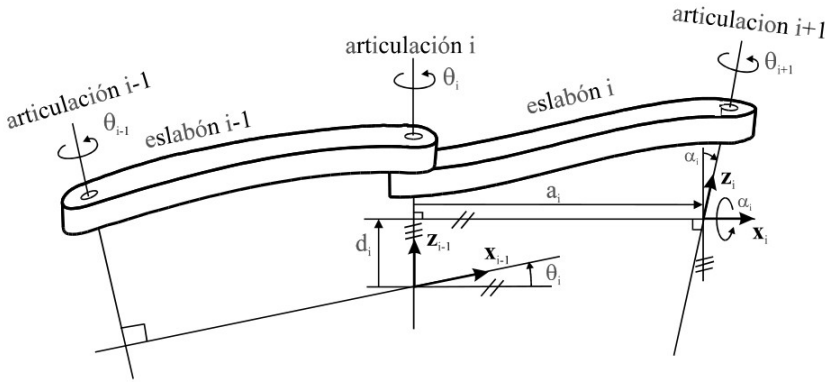
DH **12**.- Obtener a_i como la distancia medida a lo largo de x_i (que ahora coincidiría con x_{i-1}) que habría que desplazar el nuevo $\{S_{i-1}\}$ para que su origen coincidiese con $\{S_i\}$.

DH **13**.- Obtener α_i como el ángulo que habría que girar entorno a x_i (que ahora coincidiría con x_{i-1}) para que el nuevo $\{S_{i-1}\}$ coincidiese totalmente con $\{S_i\}$.

DH **14**.- Obtener las matrices de transformación ${}^{i-1}A_i$.

DH **15**.- Obtener la matriz de transformación entre la base y el extremo del robot
$T = {}^0A_1 \, {}^1A_2 \ldots {}^{n-1}A_n$.

DH **16**.- La matriz T define la orientación (submatriz de rotación) y posición (submatriz de traslación) del extremo referido a la base en función de las n coordenadas articulares .



10 Anexo B

10.1 Tipos de articulaciones

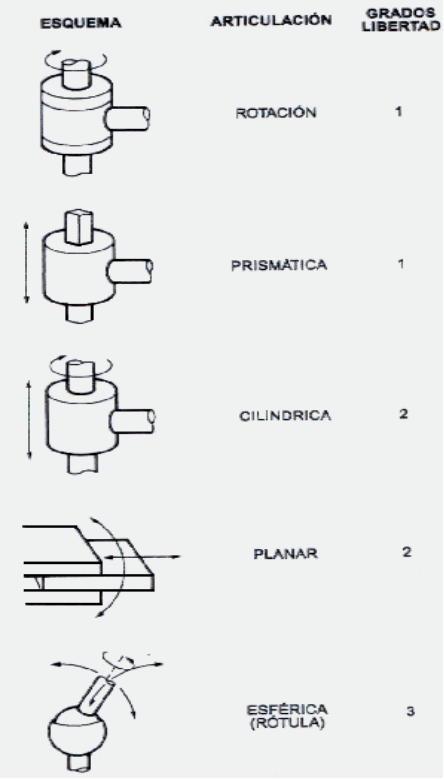


Ilustración 40: Tipos de articulaciones

La articulación de rotación suministra un grado de libertad consistente en una rotación alrededor del eje de la articulación. Esta articulación es la más utilizada.

En la articulación prismática el grado de libertad consiste en una traslación a lo largo del eje de la articulación.

En la articulación cilíndrica existen dos grados de libertad: una rotación y una traslación.

La articulación planar está caracterizada por el movimiento de desplazamiento en un plano, existiendo por lo tanto, dos grados de libertad.

La articulación esférica combina tres giros en tres direcciones perpendiculares en el espacio.
Los grados de libertad son el número de parámetros independientes que fijan la situación del órgano terminal. El número de grados de libertad suele coincidir con el número de eslabones de la cadena cinemática.

11 Anexo C

11.1 ROS

Robot Operating System es un Framework de Desarrollo en Robótica (RSF) que ofrece herramientas y librerías para el desarrollo de sistemas robóticos. Incluye también abstracción de *hardware*, controladores de dispositivos, bibliotecas, visualizadores, paso de mensajes, gestión de paquetes, etc.

Ilustración 41: Logotipo de ROS

A pesar de su nombre, ROS no es un sistema operativo propiamente dicho, de hecho, está orientado para el sistema *UNIX*, principalmente Ubuntu. Actualmente se está adaptando a otros sistemas operativos como *Mac OS X, Fedora, Debian, Gentoo, OpenSUSE,* o *Microsoft Windows* aunque están consideradas como versiones experimentales. ROS se encuentra bajo la licencia *BSD*.
Fue desarrollado en 2007 por el Laboratorio de Inteligencia Artificial de *Stanford* bajo el nombre de *switchyard*. Desde 2008, el desarrollo continua en el instituto de investigación robótico *Willow Garage*. Se han desarrollado las siguientes versiones:

- 22/01/2010 - ROS 1.0
- 01/03/2010 - Box Turtle
- 03/08/2010 - C Turtle
- 02/03/2011 - Diamondback
- 30/08/2011 - Electric Emys
- 23/04/2012 - Fuerte
- 31/12/2012 - **Groovy**

En el momento de finalización de esta memoria se está desarrollando una versión de ROS que se empleará para robots industriales, realizando un mayor esfuerzo en la eficiencia, repetitividad y tiempo real. El nombre de esta versión es *ROS Industrial*.
Existen tutoriales en la página oficial de ROS, www.ros.org, que facilitan mucho su comprensión y puesta en práctica con diferentes robots. También incluye ejemplos de programación con 3 robots reales, *PR2, Husky y TurtleBot*, siendo el primero el favorito y el más elaborado.

ROS aporta soluciones al problema de desarrollo de software para robots en distintas áreas:

Abstracción de Hardware (HAL)[1] encapsulando los robots y dispositivos tras interfaces estables y manteniendo las diferencias de estos en ficheros de configuración.

Cada desarrollador de software tiene como preferente un lenguaje de programación, debido a la sintaxis, eficiencia, tiempos de ejecución/compilación. Por ello *ROS* es multilenguaje, soporta lenguajes como *C++, Python, Octave y LISP*.
En apoyo del desarrollo en diferentes lenguajes, ROS utiliza el llamado lenguaje de definición de interfaz (*IDL*) para describir los mensajes que se envían entre los módulos. El *IDL* usa pequeños ficheros de texto para describir los campos de cada mensaje. Como se puede observar en el siguiente ejemplo como se describen los campos de un mensaje de nube de puntos.

```
Header header
 Point32[] pts
 ChannelFloat32[] chan
```

Algoritmos de robótica con bajo acoplamiento: desde algoritmos de bajo nivel de control, cinemática, *SLAM*[2], etc. Hasta algoritmos de alto nivel como planificación o aprendizaje.

Estructura y jerarquía del sistema de ficheros organizada. Se basa en directorios especiales denominados paquetes, los cuales son el nivel más bajo en la organización de software de ROS, además contienen una estructura determinada. Constan de manifiestos que describen dichos paquetes y que son muy necesarios para indicar las dependencias entre ellos. Un conjunto de paquetes conforman una estructura llamada *stack* o pila.

Para las comunicaciones ROS utiliza un mecanismo **(middleware)** distribuido entre diferentes nodos. Un nodo es un programa software del sistema (desde un algoritmo cinemático hasta un *driver* de visión por computador). Un sistema robótico se compone de diferentes nodos. El objetivo de este sistema es doble:

1. Encapsulado, abstracción y reutilización del software.

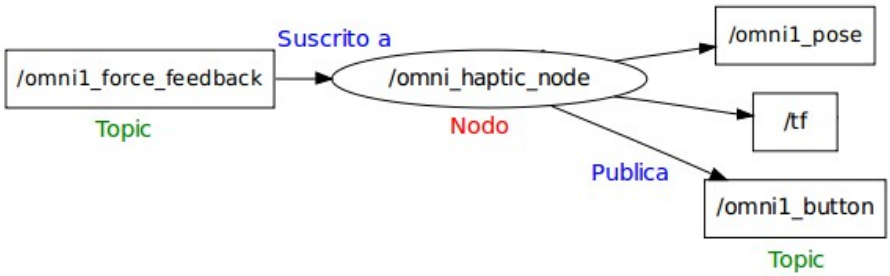
2. Independencia de la localización de los nodos ya que un sistema robótico puede tener más de un procesador.

1 HAL: capa de abstracción de *hardware*. Permite a las aplicaciones de escritorio detectar y usar el *hardware* a través de una API simple.

2 SALM: Es una técnica usada por robots y vehículos autónomos para construir un mapa de un entorno desconocido en el que se encuentra, a la vez que estima su trayectoria al desplazarse dentro de este entorno.

Los nodos se comunican entre ellos mediante diferentes mecanismos:

- **Paso de mensajes**: Un nodo puede publicar o suscribirse a un *topic*. En este *topic* se escriben/leen mensajes que han sido depositados por otros nodos. Puede haber varios publicadores y subscriptores sobre el mismo *topic* concurrentemente.
  Un único nodo puede publicar y subscribirse a múltiples *topics*.
  Los mensajes son estructuras de datos que soportan tipos primitivos como *integer, float, pount, boolean*... al igual que *arrays* de primitivas y constantes. Adicionalmente los mensajes pueden estar compuestos por otros mensajes y por *arrays* de mensajes, dando la profundidad que se requiera.



  Se puede observar cómo el nodo llamado */omni_haptic_node* está suscrito al *topic /omni1_force_feedback* y a su vez publica mensajes en tres *topics* diferentes llamados */omni1_pose, /tf y /omni1_button.*
  Éste es un ejemplo muy sencillo ya que sólo hay un nodo en ejecución, pero este esquema puede ampliar en gran medida su tamaño.
  A pesar de que el modelo basado en la publicación-suscripción de *topics* es un paradigma de comunicación flexible, su sistema de difusión no es apropiado para transacciones síncronas en sistemas distribuidos.

  Por ello existe la posibilidad de comunicación mediante la llamada a servicios.

- **Llamada a servicios**: Estos se definen mediante un par de mensajes, unos para la solicitud y otro para la respuesta. Un nodo ofrece un servicio bajo un nombre *string* y un cliente llama al servicio enviando un mensaje de solicitud y esperando a la respuesta.
  Los servicios son definidos mediante los archivos *SRV*, que se compilan en código fuente por una biblioteca de cliente *ROS*.
  Al igual que los *topics*, los servicios tienen un tipo de servicio asociado que es el nombre de paquete del archivo `.srv`. Como con otros sistemas de ficheros basados en tipos, el tipo de servicio es el nombre del paquete más el nombre del archivo `.srv`.

En el siguiente ejemplo se puede observar la llamada a un servicio.

```
ServiceProxy('kuka_lwr_kinematics/get_ik', GetPositionIk)
```

kuka_lwr Lwr_kinematics se refiere al nodo que implementa el servicio
get_ik es el nombre público del servicio
GetPositionIk es el tipo de servicio.

La siguiente tabla muestra la descripción del tipo de servicio. Cabe recordar que los tipos se encapsulan desde tipos básicos o primitivas como *integer*, *float, string*... hasta llegar a tipos elaborados como por ejemplo *PositionIKRequest*.

kinematics_msgs/GetPositionIK.srv

```
  # A service call to carry out an inverse kinematics computation
  # The inverse kinematics request kinematics_msgs/
PositionIKRequest ik_request
  # Maximum allowed time for IK calculation
duration timeout
```

```
  # The returned solution
  # (in the same order as the list of joints specified in the IKRequest message)
arm_navigation_msgs/RobotState solution
arm_navigation_msgs/ArmNavigationErrorCodes error_code
```

ROS incorpora una gran cantidad de comandos que facilitan la labor del programador. Algunos de ellos semejantes a los propios de *Unix* pero mejorados y dedicados al sistema de ficheros de ROS.
Para poder trabajar con ellos es necesario incluir el paquete *rosbashrc* en el *bashrc* del computador.

En la siguiente tabla se muestran los comandos que más se han utilizado:

roscd	Permite cambiar de paquete, directorio o *stack*. Sin necesidad que sean colindantes.
rosls	Muestra información acerca de un paquete, directorio o *stack*
rosrun	Permite ejecutar un ejecutable / nodo de cualquier paquete
roslaunch	Permite lanzar varios nodos y parámetros simultáneamente
rxgraph	Muestra un gráfico con información de la ejecución actual
rxplot	Genera gráficos a partir de los datos dados
roscreate-pkg	Crea un nuevo paquete ROS, se pueden añadir dependencias
rosmake	Compila el código fuente

rosparam	{list} Muestra la lista de parámetros en ejecución {set, get, load} Modifica, obtiene, carga parámetros
rosservice	{list} Muestra la lista de servicios en ejecución {call} Llama a un servicio con los argumentos deseados {find, info} Busca, informa sobre un servicio {type} Muestra el tipo de servicio
rosnode	{list} Muestra la lista de nodos en ejecución {info} Muestra información sobre el nodo {ping} Comprueba la conexión con el nodo {kill} Mata el nodo indicado
rostopic	{list} Muestra la lista de *topics* en ejecución {echo} Muestra información sobre el {type} Muestra el tipo del *topic* {pub} Publica un mensaje en un *topic* dado

12 Anexo D

12.1 Instalaciones

En el siguiente apartado se va a explicar detalladamente como instalar y poner en marcha todo el software utilizado a lo largo de todo el proyecto.
Las instalaciones se realizarán sobre el sistema operativo Linux Ubuntu 10.04 LTS, versión estable y con soporte para cuatro años por parte de la comunidad de desarrolladores de Ubuntu.

12.1.1 ROS

ROS es la parte software fundamental del proyecto ya que es el *framework* utilizado en en todo su desarrollo.
La página oficial de ROS, www.ros.org/wiki/ muestra, entre sus numerosos tutoriales, uno dedicado a dicha instalación. Para cada versión de ROS se aplica un tipo de instalación diferente. Para esta guía rápida se va a documentar como instalar ROS Fuerte, ya que gran parte de este proyecto se ha realizado con *Electric*, pero la versión final y funcional es Fuerte.

Para que los repositorios sean aceptados por la distribución de Ubuntu se deben configurar dichos repositorios parta que permita entradas de tipo “restricted”, “universe” y “multiverse”.

Es necesario configurar la lista de fuentes “source.list” para que acepte paquetes de ros.org.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu lucid
main" > /etc/apt/sources.list.d/ros-latest.list'
```

Configurar las llaves.

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

Asegurarse que se esta indexado con ros.org, para ello actualizar el sistema.

```
sudo apt-get update
```

Existen diferentes modos de instalación de ROS, no obstante nos centraremos en la instalación completa ya que a lo largo del proyecto se van a necesitar todas o gran parte de las funcionalidades de ROS

```
sudo apt-get install ros-fuerte-desktop-full
```

La instalación puede llevar unos instantes ya que se instala alrededor de 1GB en el disco tras la previa descarga.

Es muy importante que se realice una correcta instalación y declaración de las variables de entorno de ROS, para que estas se agreguen automáticamente al sistema. De ese modo cada vez que se lance una terminal podremos utilizar los comandos que ROS ofrece. No obstante,a lo largo del proyecto se retocará el fichero *.bashrc* para el correcto funcionamiento de las diferentes librerías utilizadas.

```
echo "source /opt/ros/fuerte/setup.bash" >> ~/.bashrc
. ~/.bashrc
```

Se trata de dos comandos diferentes.
El primero añade la fuente *source /opt/ros/fuerte/setup.bash* al fichero *.bashrc.*
El segundo actualiza el propio *bash* sin necesidad de volver a abrir una terminal.

El sistema es capaz de gestionar más de una versión ROS instaladas en el mismo PC. Para poder trabajar correctamente se debe indicar en el fichero *.bashrc* que versión se desea utilizar en cada momento.

Existen otras herramientas como *rosinstall* y *rosdep* que se utilizan con frecuencia en línea de comandos. *rosinstall* permite descargar fácilmente árboles de código fuente para los paquetes de ROS y pilas. *rosdep* permite instalar fácilmente las dependencias del sistema de fuente que va a compilar.

Para instalar estas herramientas, se debe ejecutar:

```
sudo apt-get install python-rosinstall python-rosdep
```

Siguiendo fielmente estos pasos se logra disponer de la versión *ROS Fuerte* en un computador.

12.1.2 Arm navigation

Arm navigation contiene herramientas útiles para la auto-generación de aplicaciones de navegación de brazos para los nuevos brazos robóticos y herramientas interactivas que son capaces de visualizar los distintos componentes proporcionados por el paquete.

Se basa en un lista de pilas que realizan todas las tareas en el movimiento y control de los brazos robóticos. Las pilas son las siguientes:

motion_planning_common - Esta pila contiene un conjunto común de componentes y paquetes que son útiles para la planificación de movimientos. A su vez contiene un conjunto de mensajes y servicios utilizados para especificar consultas a los planificadores de movimiento.

collision_environment - Esta pila contiene herramientas que crean una representación del entorno para la comprobación de colisiones.

motion_planning_environment - Esta pila contiene herramientas que se utilizan para comprobar las colisiones para un estado del robot dado, a su vez comprueba colisiones en las trayectorias y suma o resta objetos del entorno.

motion planners - Esta pila contiene 3 planificadores de movimiento utilizados para la creación de planes de movimiento de un brazo robótico.

kinematics - Esta pila contiene un conjunto de servicios y mensajes de planificación de movimientos que permiten llevar a cabo cálculos cinemáticos.

trajectory_filters - Esta pila contiene un conjunto de suavizadores que se utilizan para suavizar trayectorias.

arm_navigation - Esta pila contiene una implementación de un nodo de acción que se comunica con los planificadores de movimiento, el entorno y los controladores para crear un completo planificador de movimiento. También implementa el seguimiento y monitorización de trayectorias para anticiparse en caso de colisión con el entorno.

motion_planning_visualization - contiene *plugins* para el visualizador *rviz* que le permiten visualizar planes de movimiento y mapas de colisión.

Para instalarlo es necesario escribir el siguiente comando en una terminal.

```
sudo apt-get install ros-fuerte-arm-navigation
```

12.1.3 Phantom Omni

Para poder utilizar el háptico *Phantom Omni* en *ROS* es necesario instalar los controladores necesarios y el paquete Phantom_Omni que contiene los nodos y lanzadores necesarios para la ejecución.
En primer lugar se debe instalar el código fuente ad hoc cedido por el laboratorio de investigación *Georgia Tech*. que incluye diversos nodos y funcionalidades para diferentes dispositivos y robots, como es el caso del Robot PR2.

El repositorio se llama *gt-ros-pkg* y se puede encontrar en la página oficial de ROS. Para descargarlo es necesario disponer del software de control de versiones GIT instalado en el ordenador. Mediante el siguiente comando iniciaremos la descarga e instalación en nuestro sistema.

```
git clone https://code.google.com/p/gt-ros-pkg/
```

Tras la instalación es necesario añadir el SDK de OpenHaptics que nos ayudará a desarrollar nuevas aplicaciones hápticas. En la página del fabricante del háptico *Omni*, http://dsc.sensable.com/ existen manuales, ejemplos y foros de discusión sobre dicho componente.

Para poder instalar el SDK es necesario adquirir una licencia, en nuestro caso basta con la licencia académica para desarrolladores, la cual es gratuita. Una vez que se ha accedido se debe descargar el siguiente SDK.

```
OpenHapticsAE_Linux_v3_0.zip
```

El proceso de instalación se basa en mover el *SDK* descargado dentro del paquete de phantom_omni, para ello.

```
mv OpenHapticsAE_Linux_v3_0.zip `rospack find phantom_omni`/
```

Seguido lanzar el *script* de instalación proporcionado.

```
roscd phantom_omni
./install_open_haptics.sh
```

Es necesario compilar nuevamente el paquete phantom_omni. Tras ello se puede ejecutar el nuevo nodo ROS compilado, cuyo nombre es *Omni.*

```
rosmake phantom_omni
rosrun phantom_omni omni
```

Solución de problemas

Instalación en 64 bits: se necesita editar el fichero *install_open_haptics.sh* y reemplazar las instancias de "32-bit" con "64 bits" y "i386" con "amd64". Se podría actualizar esta secuencia de comandos para que la arquitectura de la plataforma se detectara automáticamente.
Es posible obtener un error por perder bibliotecas *libHD* al intentar ejecutar el nodo *Omni*. Esto se debe a que *OpenHaptics* instala curiosamente bibliotecas /usr/lib64. La mejor manera de resolver esto es crear un archivo /etc/ld.so.conf.d/openhaptics.conf con la línea *"/usr/lib64"*.

Versiones de Ubuntu 11.04 y Ubuntu posteriores: ya no se utiliza raw1394, por lo que no se puede utilizar los controladores *OpenHaptics v3.0*. Hasta que la versión v3.1 no se desarrolle se pueden realizar dos alternativas:

Reemplazar los controladores con los *drivers* inestables "juju" que están disponibles en los foros de DSC. (recomendado)

Crear un módulo raw1394. Una consecuencia de esto es que el *script set_permissions_1394.sh* fallará. En lugar de utilizar este *script*, debe seguir las instrucciones de http://superuser.com/a/239692 para establecer los permisos de forma permanente.

Al final de este proyecto se ha intentado realizar los pasos anteriores para actualizar a una versión más reciente de Ubuntu, pero no ha sido posible, por lo que se mantiene la versión inicial, Ubuntu 10.04 LTS.

Con el paso del tiempo puede que la posición del *Omni* no sea la correcta,que existan **desfases** en la posición, por ello es necesario recalibrar el sistema para corregirlo. Se debe de asegurar de reiniciar el nodo ROS con el *stylus* fuera de la base de calibración y volverlo a insertar cuando se le sea indicado.

Dispositivo no encontrado: si este error se da al ejecutar el nodo ROS puede ser debido a que no se ha emparejado correctamente el ordenador con el háptico, para corregir este error se debe lanzar nuevamente */usr/sbin/PHANToMConfiguration.*

Puede que no tenga acceso a */dev/raw1394* para arreglar esto debe ejecutar el *script set_permissions_1394.sh*

12.1.4 Orocos KDL

Para instalar *Orocos* en la distribución de *ROS Fuerte* se deben seguir los siguientes pasos:

Instalar el paquete *orocos_toolchain*

```
sudo aptitude install ros-fuerte-orocos-toolchain
```

Añadir al fichero *.bashrc*

```
source `rosstack find orocos_toolchain`/env.sh
```

Mediante la herramienta git se clona el repositorios

```
git clone --recursive http://git.gitorious.org/orocos-
toolchain/orocos_toolchain.git
```

Resolver manualmente las dependencias recursivas

```
apt-get install rubygems1.8 libxslt1-dev
```

Instalar las dependencias perdidas

```
rosdep install orocos_toolchain
```

Indicar la fuente del *script* `env.sh` y compilar

```
source orocos_toolchain/env.sh
rosmake orocos_toolchain
```

Instalar *KDL*

```
git clone
http://git.mech.kuleuven.be/robotics/orocos_kinematics_dynamics.git
```

12.1.5 Gedit & plugins

La herramienta *gedit* viene instalada por defecto en las versiones de *Ubuntu* como editor de texto. A pesar de ser una buena herramienta donde poder programar *scripts* y otros códigos ad hoc, es necesario instalar ciertos *plugins* para dotar a la herramienta de una mayor versatilidad.
Se destacan los siguientes por su mayor aporte:

- Cambiar mayúsculas y minúsculas: Cambia el tipo del texto seleccionado.
- Estadísticas del documento: análisis del documento e informes del número de palabras, líneas, caracteres y carcteres no espaciados.
- Herramientas externas: Ejecutar comandos externos y scripts de shell.
- Explorador de archivos: Acceder fácilmente a su sistema de archivos. Incluye soportes remotos, creación de nuevos archivos / directorios, directorios de monitor de cambios, etc.
- Inserta la fecha y la hora actual en la posición del cursor.
- Consola Python: la consola interactiva de Python que se coloca en el panel inferior.
- Buscar / Reemplazar Avanzado

En la siguiente dirección se pueden encontrar todos los *plugin* existentes para cada versión de *gedit*.

https://live.gnome.org/Gedit/Plugins

Tras descargar el *plugin* deseado debe realizar lo siguiente:

Cierre todas las ventanas de gedit.
Extraiga el archivo descargado.
Copie los archivos a la carpeta especificada
Instalar para el usuario actual
~ / .local / share / gedit / plugins para Gnome3
~ / .gnome2/gedit/plugins para Gnome2
Instalar para todos los usuarios
/ usr / lib / gedit / plugins para Gnome3
/ usr/lib/gedit-2/plugins para Gnome2
Resolver el problema de privilegios si es necesario.
Reiniciar gedit
En el menú principal gedit, vaya a Editar-> Preferencias
Ir a la pestaña Plugins
Buscar Avanzada Buscar / Reemplazar en la lista y comprobar que
En el menú principal, Buscar-> Advanced Find / Replace o usando Shift + Ctrl + F

13 Anexo E

13.1 Reuniones y Demos

Videoconferencia (20 Febrero 2013)

Participantes: Urko Esnaola, Jose Luis Outón
Hora estimada: 09.30 a.m. UTC±00:00, Z // UTC+09:00, I (Japón)
Duración: 1 Hora y 30 minutos
Medio: Google +

Orden del día:

En la reunión se hablará acerca del estado actual del proyecto.
Se tratará:

- Adquisición de conocimiento
- Nuevos hitos, modo absoluto y relativo de teleoperación

Temas de la reunión:

1. Poner en práctica los conocimientos de ROS adquiridos y generar un modelo del manipulador e incluir los *TFs* del háptico *Omni* para que se puedan visualizar en *rviz*.

2. Entender las matrices de rotación.

3. Modos de teleoperación, absoluto y relativo

Conclusiones:

Se debe generar un modelo *URDF* del manipulador y del entorno incluyendo el Háptico *Omni* para poder visualizarlo en *rviz* y a tras ello enviar posiciones al robot.

Matrices de rotación necesarias para el modelo relativo de teleoperación.
Dos modos necesarios para acceder a mas localizaciones del espacio de trabajo del manipulador. Utilizar los botones del *Stylus* para indicar en que momento se teleopera y como función de hombre-muerto.

Videoconferencia (4 Marzo 2013)

Participantes: Urko Esnaola, Jose Luis Outón
Hora estimada: 10.30 a.m. UTC±00:00, Z // UTC+09:00, I (Japón)
Duración: 1 Hora
Medio: Google +

Orden del día:

En la reunión se hablará acerca del estado actual del proyecto.
Se comentarán los cambios realizados en los proyectos desde el 19 de Marzo hasta la fecha actual.
Se tratarán también:

- Lineas de mejora
- Viabilidad de los cambios realizados
- Estado de la memoria
- Implantación de tecnología desarrollada en robot *Fanuc*
- Próximo hito en el proyecto
- De que manera se va a gestionar el proyecto.
- Planteamiento y solución de problemas de software

Temas de la reunión:

1. Se ha empezado a trabajar en la implantación de la teleoperación en el robot industrial *Fanuc* mediante *modbus*.
2. Se procedió a investigar en la utilización de *Wiimote* como medio de teleoperación del robot *Kuka Lwr*.
3. Cambio de ubicación en los *TF* del háptico *Omni* con respecto al robot. La ubicación actual se encuentra centrada en el medio del espacio de trabajo del robot.
4. Se ha realizado una limpieza de archivos inútiles y se ha implantado el modelo mejorado y simplificado.
5. Me encuentro trabajando en *arm kinematics contraint aware (Planing scene, constraint)*
6. Los nuevos cables de la mano de *robotiq* ya están pasados por dentro del robot *Kuka*.

Conclusiones:

Puesta al día del estado del proyecto. Aceptación de los cambios producidos en el modelo del robot. Selección de próximos hitos.
Cesión de conocimiento adquirido sobre Wiimote, para otro proyecto relacionado.

Videoconferencia (18 Marzo 2013)

Participantes: Urko Esnaola, Jose Luis Outón
Hora estimada: 09.30 a.m. UTC±00:00, Z // UTC+09:00, I (Japón)
Duración: 1 Hora
Medio: Google +

Orden del día:

En la reunión se hablará acerca del estado actual del proyecto.
Se tratará:

- Lineas de mejora
- Teleoperación con detección de colisiones (Planning scene, Enviroment server)
- Estado de la memoria
- Implantación de toda la tecnología desarrollada en robot Fanuc
- Cambiar paquete de *divers* del háptico *Omni*. Problema de dependencias.
- Planteamiento de problemas de software. Método get_Fk()

Temas de la reunión:

1. Se ha empezado a trabajar en la implantación de la teleoperación en el robot industrial Fanuc mediante *modbus*.

2. Es necesario cambiar el paquete de instalación y *drivers* del háptico *Omni*, para solucionar problema con dependencias y poder teleoperar le robot *kuka* con cualquier otro dispositivo.

3. Problema con el método *get_fk* debido al cambio de las referencias del modelo.

4. Me encuentro trabajando en *arm kinematics contraint aware* (Planing scene, constraint).
   Se ha conseguido detectar las colisiones con el entorno de trabajo del robot. (colisiones con la mesa a la que esta anclado el brazo, colisiones entre el brazo y la garra añadida...)

Conclusiones:

Cambiar de directorio al paquete del háptico *Omni*. Nueva ubicación: paquete externo al robot para evitar dependencias.
Investigar acerca de las cinemáticas del paquete *arm_navigation*. Implementar código en C++.
Cinemáticas de *tecnalia-overlay* funcionan correctamente.

Videoconferencia (25 Marzo 2013)

Participantes: Urko Esnaola, Jose Luis Outón
Hora estimada: 09.30 a.m. UTC±00:00, Z // UTC+09:00, I (Japón)
Duración: 1 Hora
Medio: Google +

Orden del día:

En la reunión se hablará acerca del estado actual del proyecto.
Se tratará:

- Lineas de mejora
- Creación de Interfaz Gráfica
- Estado de la memoria
- Lograr máxima telepresencia
- Futuros hitos. Feedback de fuerzas

Temas de la reunión:

1. Buscar lineas de mejora sobre el trabajo realizado. Buscar valor añadido para lograr mayor telepresencia por parte del operador.

2. Nuevo Objetivo: Se va proceder a crear una interfaz gráfica para facilitar la teleoperación del usuario, aportando datos importantes del estado del robot,

3. Solucionado el problema con el método *get_fk* debido al cambio de las referencias del modelo.

4. Finalizado el trabajo de *arm kinematics contraint aware* (Planing scene, constraint). Se ha conseguido detectar colisiones con el entorno dento del espacio de trabajo del robot.

Conclusiones:

Dotar a la interfaz gráfica de accesos directos por teclado para realizar tareas de teleoperación, como cambiar el factor de escala, cambiar el modo de ejecución, etc.

Videoconferencia (5 Abril 2013)

Participantes: Urko Esnaola, Jose Luis Outón
Hora estimada: 09.30 a.m. UTC±00:00, Z // UTC+09:00, I (Japón)
Duración: 1 Hora
Medio: Google +

Orden del día:

En la reunión se hablará acerca del estado actual del proyecto.
Se tratará:

- Lineas de mejora
- Estado de la memoria
- Feedback de fuerzas y control de posición
- Futuro Hito: Aprendizaje del Robot

Temas de la reunión:

1. Se ha finalizado la interfaz gráfica consiguiendo muy bueno resultados. Subir el código a *subversion* para el control de versiones.

2. El estado de la memoria es correcto, ya se han escrito 3/5 partes del total.

3. Estudio de las opciones del manipulador para obtener el par que detecta en cada articulación

4. Enviar fuerzas al *Omni* que indiquen la diferencia de posición entre manipulador y háptico

Conclusiones:

Se subirán los nuevos ficheros de código *python* a *subversion* a lo largo de la mañana para que Urko pueda tener acceso a ellos.
El manipulador posee un *topic* que indica el par que esta sufriendo proyectado sobre el ultimo efector.
Modificar el código del fichero `Omni.cpp` para recibir los vectores de fuerzas.

Videoconferencia (22 Abril 2013)

Participantes: Urko Esnaola, Jose Luis Outón
Hora estimada: 10.30 a.m. UTC±00:00, Z // UTC+09:00, I (Japón)
Duración: 1 Hora y 30 minutos
Medio: Google +

Orden del día:

En la reunión se hablará acerca del estado actual del proyecto.
Se tratará:

- Estado general de la memoria
- Estado general del proyecto
- *Feedback* de fuerzas y control de posición
- Aprendizaje del Robot

Temas de la reunión:

1. Se ha finalizado el hito del *feedback* de fuerzas y el control de posición con bueno resultados visibles en simulación y en el robot real.

2. El estado de la memoria es correcto, ya se han escrito 3/5 partes del total.

3. Trabajando en el aprendizaje del robot

Conclusiones:

Se subirán los nuevos ficheros de código *python* a *subversion* a lo largo de la mañana para que Urko pueda tener acceso a ellos.
Se deben crear ficheros de aprendizaje donde se guarden las posiciones articulares por las que ha pasado *Lwr* durante la ejecución de una tarea teleoperada.
Estos ficheros deben ser accesibles para su posterior reproducción con fin de enviar las posiciones guardadas al manipulador de nuevo.

Las **demos** han sido numerosas y contantes a lo largo de todo el desarrollo del proyecto debido a las diferentes visitas a Tecnalia por parte de clientes, estudiantes e interesados en el proyecto.

14 Anexo F

14.1 Código Fuente

En esta sección se muestra el código fuente de cada uno de los ficheros que componen el sistema.

14.1.1 change_mode.py

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('lwr_teleop')
import rospy
import subprocess

from std_msgs.msg import String


class ChangeMode:
    ''' This node change the mode of execution. Between Ralative and Absolute mode,
        according to the key that was pressed '''

    def __init__(self):
        # Subscription to key_publish
        rospy.Subscriber('keyboard_key', String, self.processButtonEvent)


    def processButtonEvent(self, data):
#       If you press the key "s" in your keyboard, you change the mode to relative
        if data.data == "s":
            subprocess.call(['rosrun', 'lwr_teleop', 'lwr_get_ik_AX.py'])
#       If you press the key "w" in your keyboard, you change the mode to absolute
        elif data.data == "w":
            subprocess.call(['rosrun', 'lwr_teleop', 'lwr_get_ik.py'])
```

```
if __name__ == '__main__':
    rospy.init_node('ChangeMode_getIK', anonymous=True)
    change_mode = ChangeMode() # Create object ChangeMode
    rospy.spin()
```

14.1.2 force_feedback.py

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('lwr_teleop')
import rospy

from geometry_msgs.msg import Vector3
from geometry_msgs.msg import Pose
from geometry_msgs.msg import Wrench
from phantom_omni.msg import OmniFeedback

class ForceFeedback:

   '''Get position and force of the real robot arm end effector  '''
   def __init__(self):
      rospy.Subscriber("/lwr/cartesian_position", Pose, self.processGetFK)

      self.omni_feedbak_pub = rospy.Publisher('/omni2_force_feedback',
OmniFeedback)
      self.max_force = 100.0
      self.min_force = -100.0

#   The force's range of the arm is [-100 , 100] and the Omni's range to keep the
stability is [-2 , 2]
#   For this reason we change the value form one range to the other. The simplest way
for this is the follow rule:
#    100 ____ 2
# force ____ X

   def processGetFK(self, data):
      pose_vec = Vector3()
      pose_vec = data.position

      force_vec = Vector3()
      force_sub = rospy.wait_for_message("/lwr/cartesian_wrench", Wrench)
      force_vec = force_sub.force

      if force_vec.x > self.max_force:
         force_vec.x = self.max_force
      elif force_vec.x < self.min_force:
         force_vec.x = self.min_force

      if force_vec.y > self.max_force:
         force_vec.y = self.max_force
      elif force_vec.y < self.min_force:
```

```
            force_vec.y = self.min_force

        if force_vec.z > self.max_force:
            force_vec.z = self.max_force
        elif force_vec.z < self.min_force:
            force_vec.z = self.min_force

#       Rule
        force_vec.x = force_vec.x * 2.0 / 100.0
        force_vec.y = force_vec.y * 2.0 / 100.0
        force_vec.z = force_vec.z * 2.0 / 100.0

        extra = force_vec.x

        force_vec.x = -force_vec.y
        force_vec.y = -extra
        force_vec.z = -force_vec.z


        omni_pub = OmniFeedback()
        omni_pub.force = force_vec
        omni_pub.position = pose_vec

        print "force_vec.x", force_vec.x
        print "force_vec.y", force_vec.y
        print "force_vec.z", force_vec.z

        self.omni_feedbak_pub.publish(omni_pub)



if __name__ == '__main__':
    rospy.init_node('force_feedback_omni_lwr')

    # Create object HapticPosePublisher
    force_feedback = ForceFeedback()
    rospy.spin()
```

14.1.3 get_FK.py

```
#!/usr/bin/python
import roslib; roslib.load_manifest('lwr_teleop')
import rospy

from kinematics_msgs.srv import GetKinematicSolverInfo
from kinematics_msgs.srv import GetPositionFK
from kinematics_msgs.srv import GetPositionFKRequest
from sensor_msgs.msg import JointState
from geometry_msgs.msg import PoseStamped



class GetFK:
    '''This class gets the cartesian position of the end effector. For this, Uses the getFk
service.'''
    def __init__(self):

        #Subscription to JointState for know the current state of joints
        self.joint_state_current = None
        rospy.Subscriber("joint_states", JointState, self.processJointStateCurrent)

        # Publish forward Kinematics
        self.forward = rospy.Publisher('get_forward_kinematic',PoseStamped)


    def processJointStateCurrent(self,joints_state_current):  # Function in which
joint_state_current is recived
        # Check if the joints state are from the arm, because there are two kind of
jointstate, one for arm an another for hand (gripper)
        if joints_state_current.header.frame_id != "RobotiqHand":
            self.joint_state_current = joints_state_current


    def getFk(self):    # Get the pose (fk_res) of the last robot's link using the get_fk
service
#       rospy.loginfo("Wait for service kuka_lwr_kinematics/get_fk ...")
        if self.joint_state_current is not None:
            rospy.wait_for_service("kuka_lwr_kinematics/get_fk")
            position_fk_request = self.createFKRequest()
            try:

                get_fk = rospy.ServiceProxy('kuka_lwr_kinematics/get_fk', GetPositionFK)
                fk_res = get_fk(position_fk_request)
                return fk_res.pose_stamped[0]   # Return only the pose stamped
```

```
            except rospy.ServiceException, e:
                print "Service call failed: %s"%e
                return -1
        else:
            return -1



    def createFKRequest(self):    # Create a FK Request
        # Get kinematic solver information
#       rospy.loginfo("Wait for service kuka_lwr_kinematics/get_fk_solver_info ...")
        rospy.wait_for_service("kuka_lwr_kinematics/get_fk_solver_info")
        ks_info = None
        try:
            get_kinematic_solver_info =
rospy.ServiceProxy('kuka_lwr_kinematics/get_fk_solver_info',
GetKinematicSolverInfo)
            ks_info = get_kinematic_solver_info()

        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
        # Create request
        fk_request = GetPositionFKRequest()
        fk_request.header.frame_id = "/calib_lwr_arm_base_link"
        fk_request.fk_link_names = "lwr_arm_7_link"
        fk_request.robot_state.joint_state = self.joint_state_current
        fk_request.robot_state.multi_dof_joint_state.joint_names =
ks_info.kinematic_solver_info.joint_names
        fk_request.robot_state.joint_state.position = self.joint_state_current.position
        return fk_request



if __name__ == '__main__':

    Get_Fk = GetFK()
    rospy.init_node('Get_FK', anonymous=True)

    rate = rospy.Rate(25.0)
    while not rospy.is_shutdown():
        tmp = Get_Fk.getFk()
        if tmp != -1:
            Get_Fk.forward.publish(tmp)
        rate.sleep()
```

14.1.4 gui_teleop.py

```
#!/usr/bin/python
import roslib
roslib.load_manifest('lwr_teleop')
roslib.load_manifest('soem_robotiq_drivers')
import rospy
import sys
import os
import math
import subprocess

from geometry_msgs.msg   import PoseStamped
from std_msgs.msg import String
from std_msgs.msg import Float32
from PyQt4 import QtGui, QtCore
from PyQt4.QtGui import QColor, QFont, QWidget, QLabel, QPalette
from soem_robotiq_drivers.msg import RobotiqSModelFeedback


class GUITeleop(QtGui.QWidget):
    ''' This class is the GUI of teleoperation. This uses PyQt4 '''

    def __init__(self):
        super(GUITeleop, self).__init__()

        self.key_publish = rospy.Publisher('keyboard_key', String)
        self.key_pressed = None
        self.selected = False


        self.reflexxes_on = rospy.Publisher('lwr/rfsm_events', String)

        self.initUI()

        self.poseCurrent = PoseStamped()

        # Subscription to Robotiq/feedback
        rospy.Subscriber("/robotiq/feedback", RobotiqSModelFeedback, self.processHand)

        # Subscription to key_publish
        rospy.Subscriber('keyboard_key', String, self.processButtonEvent)

        # Subscription to forward Kinematics
        rospy.Subscriber('/get_forward_kinematic', PoseStamped, self.processReadPose)
```

```
        rospy.Subscriber('/teleop_pose',PoseStamped, self.processTargetPose)

        rospy.Subscriber('/scale_publish',Float32, self.processScale)

        rospy.Subscriber('/lwr/status',String, self.processError_msg)

        #Publish the name of file where it will record the trajectory
        self.file_publisher = rospy.Publisher('rec_file_name', String)

        self.openEvent()


    def processReadPose(self, data):
#       For each returned data, refresh the robot position and orientation in the text box
of GUI
        self.poseCurrent = data

self.lineedits_pose['button_pose_x'].setText(str("{0:.3f}".format(self.poseCurrent.pose.p
osition.x)))

self.lineedits_pose['button_pose_y'].setText(str("{0:.3f}".format(self.poseCurrent.pose.p
osition.y)))

self.lineedits_pose['button_pose_z'].setText(str("{0:.3f}".format(self.poseCurrent.pose.p
osition.z)))

self.lineedits_orient['button_orient_x'].setText(str("{0:.3f}".format(self.poseCurrent.pos
e.orientation.x)))

self.lineedits_orient['button_orient_y'].setText(str("{0:.3f}".format(self.poseCurrent.pos
e.orientation.y)))

self.lineedits_orient['button_orient_z'].setText(str("{0:.3f}".format(self.poseCurrent.pos
e.orientation.z)))

self.lineedits_orient['button_orient_w'].setText(str("{0:.3f}".format(self.poseCurrent.po
se.orientation.w)))

    def processTargetPose(self,data):
#   For each returned data, we calculate the difference between the target position
Vector and current position vector and write in the error text box
        x_ = math.pow((self.poseCurrent.pose.position.x - data.pose.position.x),2)
        y_ = math.pow((self.poseCurrent.pose.position.y - data.pose.position.y),2)
        z_ = math.pow((self.poseCurrent.pose.position.z - data.pose.position.z),2)
```

```
        self.qleError.setText(str("{0:.3f}".format(math.sqrt(x_ + y_ + z_)-0.850)))


    def processScale(self, data):
        self.qleScale.setText(str("{0:.1f}".format(data.data)))

    def processError_msg(self, data): # Error message
        self.qleError_msg.setText(data.data)

    def processHand(self, data): # Only is necessary the data of one finger because the
three have the same behavior
        self.qle_gripper.setText (str(data.finger_a_position))


    def processButtonEvent(self,data):
#       Change the colour of the different box, to indicate where you are
        palette_white = QPalette(QLabel().palette())
        palette_white.setColor(QPalette.WindowText, QColor(255, 255, 255)) # Text
White
        palette_white.setColor(QPalette.Window, QColor(0, 15, 255)) # Back Blue

        palette_black = QPalette(QLabel().palette())
        palette_black.setColor(QPalette.WindowText, QColor(0, 0, 0)) # Text Black
        palette_black.setColor(QPalette.Window, QColor(255, 255, 255)) # Back White

        palette_red = QPalette(QLabel().palette())
        palette_red.setColor(QPalette.WindowText, QColor(255, 255, 255)) # Text White
        palette_red.setColor(QPalette.Window, QColor(255, 0, 0)) # Back Red

        if data.data == "s":
            self.square_rel.setPalette(palette_white)
            self.square_abs.setPalette(palette_black)

        elif data.data == "w":
            self.square_rel.setPalette(palette_black)
            self.square_abs.setPalette(palette_white)

        elif data.data == "r":
            self.btn_dialog.click()
            self.square_rec.setPalette(palette_red)

        elif data.data == "f":
            self.square_rec.setPalette(palette_black)
            self.square_rel.setPalette(palette_black)
            self.square_abs.setPalette(palette_black)
```

```
    def initUI(self):
        fn = os.path.join(os.path.dirname(__file__))  # Get the relative path we are using.
This path is where this node are.
        path = fn.replace("nodes","images")  # Change the relative path for the correct
path of the images

        self.resize(1050, 650)
        self.center()
        QtGui.QToolTip.setFont(QtGui.QFont('SansSerif', 10))

        self.setWindowIcon(QtGui.QIcon(path+"/Tecnalia.jpg"))
        self.setWindowTitle('LWR Teleop')

#    ---------------------------------------  The code below puts different images in the
main screen

        self.setFocusPolicy(QtCore.Qt.StrongFocus) # put the focus in the main screen

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-q-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to <b>increase</b> the scale')
        lbl.move(90,70)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-a-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to <b>decrease</b> the scale')
        lbl.move(90,200)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-w-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to change to the <b>absolute
execution mode</b>')
        lbl.move(260,70)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-s-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to change to the <b>relative
execution mode</b>')
        lbl.move(260,160)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-e-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to <b>close the gripper</b>')
        lbl.move(585,70)
```

```
        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-d-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to <b>open the gripper</b>')
        lbl.move(585,200)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-r-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to <b>record</b> the trajectory')
        lbl.move(800,70)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/letra-f-tecla.gif"))
        lbl.setToolTip('Press this button in your keyboard to <b>finish the record</b>')
        lbl.move(800,160)

        lbl = QtGui.QLabel(self)
        lbl.setScaledContents(True)
        lbl.setPixmap(QtGui.QPixmap(path+"/arrow_up.png"))
        lbl.setToolTip('Press this button in your keyboard to <b>increase</b> the scale')
        lbl.move(65,75)
        lbl.setMaximumSize(QtCore.QSize(17, 30))

        lbl = QtGui.QLabel(self)
        lbl.setScaledContents(True)
        lbl.setPixmap(QtGui.QPixmap(path+"/arrow_down.png"))
        lbl.setToolTip('Press this button in your keyboard to <b>decrease</b> the scale')
        lbl.move(65,210)
        lbl.setMaximumSize(QtCore.QSize(17, 30))

#    ----------------------------------------- Scale label and text box

        font = QtGui.QFont('Trebuchet MS, Bold Italic',15)

        lbl = QtGui.QLabel(self)
        lbl.setText(" - Scale - ")
        lbl.move(73, 30)
        lbl.setFont(QtGui.QFont(font))
        lbl.setToolTip('This is the <b>scale</b> of the relative mode. With this you can
have more or less precision in the robot')

        self.qleScale = QtGui.QLineEdit(self)
        self.qleScale.setGeometry(56, 140, 120,40)
        self.qleScale.setReadOnly(True)
        self.qleScale.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',17))
        self.qleScale.setAlignment(QtCore.Qt.AlignCenter)
```

```
        self.qleScale.setToolTip('This is the <b>scale</b> of the relative mode. With this
you can have more or less precision in the robot')
        self.qleScale.setFocusPolicy(QtCore.Qt.NoFocus)

#   ------------------------------------------ Mode label and text box

        palette_white = QPalette(QLabel().palette())
        palette_white.setColor(QPalette.Window, QColor(255, 255, 255)) # Back Blue

        self.square_abs = QtGui.QLabel(self)
        self.square_abs.setAutoFillBackground(True)
        self.square_abs.setAlignment(QtCore.Qt.AlignCenter)
        self.square_abs.setText("Absolute")
        self.square_abs.setToolTip('This is the <b>absolute mode</b> of teleoperation')
        self.square_abs.setFont(QtGui.QFont(font))
        self.square_abs.setGeometry(330, 70, 130, 40)
        self.square_abs.setPalette(palette_white)
        self.square_abs.setFocusPolicy(QtCore.Qt.NoFocus)

        self.square_rel = QtGui.QLabel(self)
        self.square_rel.setAutoFillBackground(True)
        self.square_rel.setAlignment(QtCore.Qt.AlignCenter)
        self.square_rel.setText("Relative")
        self.square_rel.setToolTip('This is the <b>relative mode</b> of teleoperation')
        self.square_rel.setFont(QtGui.QFont(font))
        self.square_rel.setGeometry(330, 160, 130, 40)
        self.square_rel.setPalette(palette_white)
        self.square_rel.setFocusPolicy(QtCore.Qt.NoFocus)

        lbl = QtGui.QLabel(self)
        lbl.setText(" - Mode - ")
        lbl.move(320, 30)
        lbl.setFont(QtGui.QFont(font))
        lbl.setToolTip('This is the execution <b>mode</b> of teleoperation')

#   ------------------------------------------ Gripper label and text box

        lbl = QtGui.QLabel(self)
        lbl.setText(" - Gripper Pose - ")
        lbl.move(535, 30)
        lbl.setFont(QtGui.QFont(font))
        lbl.setToolTip('This is the execution <b>mode</b> of teleoperation')

        lbl = QtGui.QLabel(self)
        lbl.setText("Close")
        lbl.move(640, 86)
```

```
        lbl.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',14))

        lbl = QtGui.QLabel(self)
        lbl.setText("Open")
        lbl.move(640, 219)
        lbl.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',14))

        self.qle_gripper = QtGui.QLineEdit(self)
        self.qle_gripper.setGeometry(540, 140, 140,40)
        self.qle_gripper.setReadOnly(True)
        self.qle_gripper.setAlignment(QtCore.Qt.AlignCenter)
        self.qle_gripper.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',17))
        self.qle_gripper.setToolTip('This is the <b>gripper pose</b> of robot')
        self.qle_gripper.setFocusPolicy(QtCore.Qt.NoFocus)

    #   ------------------------------------------ Recorder label and text box

        lbl = QtGui.QLabel(self)
        lbl.setText(" - Recorder - ")
        lbl.move(835, 30)
        lbl.setFont(QtGui.QFont(font))
        lbl.setToolTip('Yoy can <b>record</b> the trajectory of the robot')

        lbl = QtGui.QLabel(self)
        lbl.setText("Start")
        lbl.move(855,75)
        lbl.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',13))

        lbl = QtGui.QLabel(self)
        lbl.setText("Stop")
        lbl.move(855, 170)
        lbl.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',13))


        self.square_rec = QtGui.QLabel(self)
        self.square_rec.setAutoFillBackground(True)
        self.square_rec.setAlignment(QtCore.Qt.AlignCenter)
        self.square_rec.setText("REC")
        self.square_rec.setToolTip('When <b>lit</b> means you are <b>recording</b>')
        self.square_rec.setFont(QtGui.QFont(font))
        self.square_rec.setGeometry(900, 115, 100, 40)
        self.square_rec.setPalette(palette_white)
        self.square_rec.setFocusPolicy(QtCore.Qt.NoFocus)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/rec.png"))
```

```
        lbl.setScaledContents(True)     # To put the image in stretching form
        lbl.setMaximumSize(QtCore.QSize(30, 30))
        lbl.move(1000,119)

#   ---------------------------------------- File select and refresh button

        self.combo_file = QtGui.QComboBox(self)
        self.combo_file.setFocusPolicy(QtCore.Qt.NoFocus)
        self.combo_file.addItem("Select File")
        self.combo_file.setGeometry(800, 270, 180, 30)
        self.combo_file.activated[str].connect(self.onActivated)
        self.combo_file.setToolTip('Select a <b>test</b> file')
        self.combo_file.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',12))

        rMyIcon = QtGui.QPixmap(path+"/refresh.png")

        btn_refresh = QtGui.QPushButton(self)
        btn_refresh.setToolTip('<b>Refresh</b> the file list')
        btn_refresh.setGeometry(990, 270, 30, 30)
        btn_refresh.setFocusPolicy(QtCore.Qt.NoFocus)
        btn_refresh.setIcon(QtGui.QIcon(rMyIcon))

        btn_refresh.clicked.connect(self.refresh)
        btn_refresh.resize(btn_refresh.sizeHint())

#   ---------------------------------------- Play button

        rMyIcon = QtGui.QPixmap(path+"/play.png")
        btn_play = QtGui.QPushButton("Play", self)
        btn_play.setToolTip('<b>Play</b> the test selected')
        btn_play.setGeometry(880, 350, 75, 45)
        btn_play.setFocusPolicy(QtCore.Qt.NoFocus)
        btn_play.setIcon(QtGui.QIcon(rMyIcon))
        btn_play.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',13))

        btn_play.clicked.connect(self.play)
        btn_play.resize(btn_play.sizeHint())

        self.btn_dialog = QtGui.QPushButton(self)
        self.btn_dialog.setFocusPolicy(QtCore.Qt.NoFocus)
        self.btn_dialog.clicked.connect(self.showDialog)
        self.btn_dialog.hide()

#   ---------------------------------------- Only Logos in the middle of the main screen

        lbl = QtGui.QLabel(self)
```

```
        lbl.setPixmap(QtGui.QPixmap(path+"/Phantom_Omni2.png"))
        lbl.setScaledContents(True)     # To put the image in stretching form
        lbl.setToolTip('Haptic Phantom Omni')
        lbl.setMaximumSize(QtCore.QSize(190, 100))
        lbl.move(self.size().width()/2 - 10, self.size().height()/2)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/kuka.png"))
        lbl.setScaledContents(True)
        lbl.setToolTip('Kuka Lwr Robot')
        lbl.setMaximumSize(QtCore.QSize(200, 230))
        lbl.move(self.size().width()/2 - 260, self.size().height()/2-80)

        lbl = QtGui.QLabel(self)
        lbl.setPixmap(QtGui.QPixmap(path+"/Tecnalia_logo.png"))
        lbl.setScaledContents(True)
        lbl.setToolTip('Inspiring Business')
        lbl.setMaximumSize(QtCore.QSize(250, 50))
        lbl.move(self.size().width()/2 + 255, self.size().height()/2 + 265)

#   ------------------------------------------ Position and Orientation labels and text box

        lbl = QtGui.QLabel(self)
        lbl.setText("Robot Position:")
        lbl.move(63, 500)
        lbl.setFont(QtGui.QFont(font))

        lbl = QtGui.QLabel(self)
        lbl.setText("Orientation:")
        lbl.move(63, 560)
        lbl.setFont(QtGui.QFont(font))

        lbl_prop = []   # This is a list of dicionaries
        lbl_prop.append( {'x': 200, 'y': 500, 'text': 'This is the <b>X axis</b> of the robot',
'txt2': 'x', 'id': 'button_pose_x'} )
        lbl_prop.append( {'x': 320, 'y': 500, 'text': 'This is the <b>Y axis</b> of the robot',
'txt2': 'y', 'id': 'button_pose_y'} )
        lbl_prop.append( {'x': 440, 'y': 500, 'text': 'This is the <b>Z axis</b> of the robot',
'txt2': 'z', 'id': 'button_pose_z'} )

        self.lineedits_pose = {}  # This is a dictionary for save the declaration of the
QlineEdit. Then you can access to the instance with the 'id' key
        for button in lbl_prop:
            self.lineedits_pose[ button['id'] ] = QtGui.QLineEdit(self)
            self.lineedits_pose[ button['id'] ].setGeometry(button['x'], button['y'], 90,30)
            self.lineedits_pose[ button['id'] ].setReadOnly(True)
```

```
            self.lineedits_pose[ button['id'] ].setFont(QtGui.QFont('Trebuchet MS, Bold
Italic',15))
            self.lineedits_pose[ button['id'] ].setFocusPolicy(QtCore.Qt.NoFocus)
            self.lineedits_pose[ button['id'] ].setAlignment(QtCore.Qt.AlignCenter)
            self.lineedits_pose[ button['id'] ].setToolTip(button['text'])

            lbl = QtGui.QLabel(self)
            lbl.setText(button['txt2'])
            lbl.move(button['x']+40, button['y']+26 )
            lbl.setFont(QtGui.QFont(font))

#       self.lineedits_pose['button_pose_y'].setText("5555")

        lbl_prop = []
        lbl_prop.append( {'x': 200, 'y': 560, 'text': 'This is the <b>X axis</b> of the robot',
'txt2': 'x', 'id': 'button_orient_x'} )
        lbl_prop.append( {'x': 320, 'y': 560, 'text': 'This is the <b>Y axis</b> of the robot',
'txt2': 'y', 'id': 'button_orient_y'} )
        lbl_prop.append( {'x': 440, 'y': 560, 'text': 'This is the <b>Z axis</b> of the robot',
'txt2': 'z', 'id': 'button_orient_z'} )
        lbl_prop.append( {'x': 560, 'y': 560, 'text': 'This is the <b>W axis</b> of the robot',
'txt2': 'w', 'id': 'button_orient_w'} )

        self.lineedits_orient = {}
        for button in lbl_prop:

            self.lineedits_orient[ button['id'] ] = QtGui.QLineEdit(self)
            self.lineedits_orient[ button['id'] ].setGeometry(button['x'], button['y'], 90,30)
            self.lineedits_orient[ button['id'] ].setReadOnly(True)
            self.lineedits_orient[ button['id'] ].setFont(QtGui.QFont('Trebuchet MS, Bold
Italic',15))
            self.lineedits_orient[ button['id'] ].setFocusPolicy(QtCore.Qt.NoFocus)
            self.lineedits_orient[ button['id'] ].setAlignment(QtCore.Qt.AlignCenter)
            self.lineedits_orient[ button['id'] ].setToolTip(button['text'])

            lbl = QtGui.QLabel(self)
            lbl.setText(button['txt2'])
            lbl.move(button['x']+40, button['y']+26 )
            lbl.setFont(QtGui.QFont(font))

#   ---------------------------------------- Error message label and text box

        lbl = QtGui.QLabel(self)
        lbl.setText('Error msg:')
        lbl.move(760, 480)
        lbl.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',17))
```

```
        palette = QPalette(QtGui.QLineEdit().palette())   # Text Color (RED)
        palette.setColor(QPalette.Text, QColor(255, 0, 0))

        self.qleError_msg = QtGui.QLineEdit(self)
        self.qleError_msg.setGeometry(880, 473, 120,40)
        self.qleError_msg.setAlignment(QtCore.Qt.AlignCenter)
        self.qleError_msg.setFocusPolicy(QtCore.Qt.NoFocus)
        self.qleError_msg.setReadOnly(True)
        self.qleError_msg.setToolTip('These are the possible <b>errors</b> that can
occur')
        self.qleError_msg.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',16))
        self.qleError_msg.setPalette(palette)  # To change the color in the text of
QLineEdit

#    ------------------------------------------ Pose error label and text box

        lbl = QtGui.QLabel(self)
        lbl.setText('Pose error:')
        lbl.move(760, 530)
        lbl.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',17))

        palette = QPalette(QtGui.QLineEdit().palette())   # Text Color (RED)
        palette.setColor(QPalette.Text, QColor(255, 0, 0))

        self.qleError = QtGui.QLineEdit(self)
        self.qleError.setGeometry(880, 523, 120,40)
        self.qleError.setAlignment(QtCore.Qt.AlignCenter)
        self.qleError.setFocusPolicy(QtCore.Qt.NoFocus)
        self.qleError.setReadOnly(True)
        self.qleError.setToolTip('This is the module vector <b>error</b> of the current
position and the target position of the robot')
        self.qleError.setFont(QtGui.QFont('Trebuchet MS, Bold Italic',18))
        self.qleError.setPalette(palette)  # To change the color in the text of QLineEdit

#    ----------------------------------------
        self.show()

#****************************************************** INIT_UI
**********************************************************************


    def showDialog(self):
#   This new dialog appear when you want to record a new trajectory. When you press
"done" or "exit", it begins to record, and no before
        text, ok = QtGui.QInputDialog.getText(self, 'File Name',
```

```
            'Write the name what you want:')

        if ok:
            text.replace(" ", "") # Removes all white spaces
            if str(text) != "":    # if is no empty, publish the name of file to
lwr_learning_recorder.py

self.file_publisher.publish("/home/across/rosstacks/lwr/lwr_robot/lwr_teleop/recording/
"+str(text)+".npz")
                self.key_publish.publish("l")
            else:
                QtGui.QMessageBox.warning(self, "warning", " The <b>name</b> of file
will be <b>\'default\'</b> ")
                self.key_publish.publish("l")
        else:
            QtGui.QMessageBox.warning(self, "warning", " The <b>name</b> of file will
be <b>\'default\'</b> ")
            self.key_publish.publish("l")


    def onActivated(self, text):
#   Indicates if a record file has selected or not
        if str(text) != "Select File":
            self.file_name = str(text)
            print self.file_name
            self.selected = True


    def refresh(self, text):
#   Gets all the files in the recording directory
        self.combo_file.clear()
        output,error = subprocess.Popen(['ls',
'/home/across/rosstacks/lwr/lwr_robot/lwr_teleop/recording'],stdout = subprocess.PIPE,
stderr= subprocess.PIPE).communicate()
        files = output.split("\n") # split the text
        for word in files: # for each word in the line:
            if word != "":
                self.combo_file.addItem(word) # add each file on a combo


    def play(self, text):
#   Executes the recorded trajectory
        if self.selected == True:
            self.qleError.setText("Playing...")
            subprocess.call(["python",
"/home/across/rosstacks/lwr/lwr_robot/lwr_teleop/nodes/lwr_learning_reproducer_with
```

```
_hand.py", "/home/across/rosstacks/lwr/lwr_robot/lwr_teleop/recording/"+
self.file_name])

        else:
            QtGui.QMessageBox.warning(self, "warning", " Select a <b>file</b> to it will
reproduce")


    def center(self): # To center the text in boxes
        qr = self.frameGeometry()
        cp = QtGui.QDesktopWidget().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())


    def keyPressEvent(self, event):
#   Makes the corresponding action for each case. Concretly publishes the key in the
topic
        key = event.key()
        if key == QtCore.Qt.Key_Q:
            self.key_pressed = "q"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_A:
            self.key_pressed = "a"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_W and self.key_pressed != "w":
            self.key_pressed = "w"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_S and self.key_pressed != "s":
            self.key_pressed = "s"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_E:
            self.key_pressed = "e"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_D:
            self.key_pressed = "d"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_R:
            self.key_pressed = "r"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_F:
            self.key_pressed = "f"
            self.key_publish.publish(String(self.key_pressed))
        elif key == QtCore.Qt.Key_Space:
            self.reflexxes_on.publish('e_start')
```

```
    def closeEvent(self, event): # Open a dialog if you close the main screen

        reply = QtGui.QMessageBox.question(self, 'Message',
            "Are you sure to quit?", QtGui.QMessageBox.Yes |
            QtGui.QMessageBox.No, QtGui.QMessageBox.No)

        if reply == QtGui.QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()



    def openEvent(self):
#   Asks if you want to start reflexxes or not. If is like this, starts reflexxes
        reply = QtGui.QMessageBox.question(self, '- Reflexxes -',
            "Do you want Start Reflexxes?", QtGui.QMessageBox.Yes |
            QtGui.QMessageBox.No, QtGui.QMessageBox.Yes)

        if reply == QtGui.QMessageBox.Yes:
            self.reflexxes_on.publish('e_start')
        else:
            QtGui.QMessageBox.question(self, '- Reflexxes -',
            "You can start reflexxes later if you press the 'space key'")



def main():
    rospy.init_node('gui_teleop')
    app = QtGui.QApplication(sys.argv)
    ex = GUITeleop()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

14.1.5 hand_fake.py

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('hand_publisher_tf')
roslib.load_manifest('soem_robotiq_drivers')
import rospy
import math
```

```
import tf
from sensor_msgs.msg import JointState
from soem_robotiq_drivers.msg import RobotiqSModelCommand


class HandStateFake:

    handOpen = 0
    #handClose = 255
    handCurrent = 0
    handCurrentScissor = 0
    handCurrentA = 0
    handCurrentB = 0
    handCurrentC = 0
    handTarget = RobotiqSModelCommand()

    def __init__(self):

        # Subscription to robotiq/command
        rospy.Subscriber("/robotiq/command", RobotiqSModelCommand,
self.processHand)
        self.pubJointStates = rospy.Publisher("/joint_states", JointState)

        self.handCurrent = self.handOpen
        self.handCurrentScissor = self.handOpen
        self.handCurrentA = self.handOpen
        self.handCurrentB = self.handOpen
        self.handCurrentC = self.handOpen

        # Default handTarget
        self.handTarget.gripper_position_req = self.handOpen
        self.handTarget.gripper_speed = 100
        self.handTarget.scissor_position_req = self.handOpen
        self.handTarget.scissor_speed = 100
        self.handTarget.finger_a_position_req = self.handOpen
        self.handTarget.finger_a_speed = 100
        self.handTarget.finger_b_position_req = self.handOpen
        self.handTarget.finger_b_speed = 100
        self.handTarget.finger_c_position_req = self.handOpen
        self.handTarget.finger_c_speed = 100
        self.handTarget.individual_finger_control = False


    def processHand(self, data): #CallBack

        self.handTarget = data
```

```
    def moveAll(self):

        positionGripper = self.handTarget.gripper_position_req
        speedGripper = self.handTarget.gripper_speed

        if positionGripper < self.handCurrent:     # When press the button white the hand
open
            while (positionGripper < self.handCurrent):
                self.handCurrent = self.handCurrent - ((speedGripper+1.0)*5.0)/255.0
                self.handCurrentA = self.handCurrent
                self.handCurrentB = self.handCurrent
                self.handCurrentC = self.handCurrent
                self.fingerTF()

            if positionGripper != self.handCurrent:
                sesgo = self.handCurrent - positionGripper
                self.handCurrent = self.handCurrent - sesgo
                self.handCurrentA = self.handCurrent
                self.handCurrentB = self.handCurrent
                self.handCurrentC = self.handCurrent
                self.fingerTF()

        if positionGripper > self.handCurrent:     # When press the button black the hand
close
            while (positionGripper > self.handCurrent):
                self.handCurrent = self.handCurrent + ((speedGripper+1.0)*5.0)/255.0
                self.handCurrentA = self.handCurrent
                self.handCurrentB = self.handCurrent
                self.handCurrentC = self.handCurrent
                self.fingerTF()

            if positionGripper != self.handCurrent:
                sesgo = positionGripper - self.handCurrent
                self.handCurrent = self.handCurrent + sesgo
                self.handCurrentA = self.handCurrent
                self.handCurrentB = self.handCurrent
                self.handCurrentC = self.handCurrent
                self.fingerTF()

        if positionGripper == self.handCurrent:
            self.handCurrentA = self.handCurrent
            self.handCurrentB = self.handCurrent
            self.handCurrentC = self.handCurrent
```

```
            self.fingerTF()



    def moveScissor(self):

        positionScissor = self.handTarget.scissor_position_req
        speedScissor = self.handTarget.scissor_speed

        if positionScissor < self.handCurrentScissor:
            while (positionScissor < self.handCurrentScissor):
                self.handCurrentScissor = self.handCurrentScissor -
((speedScissor+1.0)*5.0)/255.0
                self.fingerTF()

            if positionScissor != self.handCurrentScissor:
                sesgo = self.handCurrentScissor - positionScissor
                self.handCurrentScissor = self.handCurrentScissor - sesgo
                self.fingerTF()

        if positionScissor > self.handCurrentScissor:
            while (positionScissor > self.handCurrentScissor):
                self.handCurrentScissor = self.handCurrentScissor +
((speedScissor+1.0)*5.0)/255.0
                self.fingerTF()

            if positionScissor != self.handCurrentScissor:
                sesgo = positionScissor - self.handCurrentScissor
                self.handCurrentScissor = self.handCurrentScissor + sesgo
                self.fingerTF()

        if positionScissor == self.handCurrentScissor:
            self.fingerTF()



    def moveEach(self):

        positionA = self.handTarget.finger_a_position_req
        speedA = self.handTarget.finger_a_speed

        if positionA < self.handCurrentA:     # When press the button white the hand open
```

```
    while (positionA < self.handCurrentA):
        self.handCurrentA = self.handCurrentA - ((speedA+1.0)*5.0)/255.0
        self.fingerTF()

    if positionA != self.handCurrentA:
        sesgo = self.handCurrentA - positionA
        self.handCurrentA = self.handCurrentA - sesgo
        self.fingerTF()

if positionA > self.handCurrentA:    # When press the button black the hand close

    while (positionA > self.handCurrentA):
        self.handCurrentA = self.handCurrentA + ((speedA+1.0)*5.0)/255.0
        self.fingerTF()

    if positionA != self.handCurrentA:
        sesgo = positionA - self.handCurrentA
        self.handCurrentA = self.handCurrentA + sesgo
        self.fingerTF()

if positionA == self.handCurrentA:
    self.fingerTF()



positionB = self.handTarget.finger_b_position_req
speedB = self.handTarget.finger_b_speed

if positionB < self.handCurrentB:    # When press the button white the hand open

    while (positionB < self.handCurrentB):
        self.handCurrentB = self.handCurrentB - ((speedB+1.0)*5.0)/255.0
        self.fingerTF()

    if positionB != self.handCurrentB:
        sesgo = self.handCurrentB - positionB
        self.handCurrentB = self.handCurrentB - sesgo
        self.fingerTF()

if positionB > self.handCurrentB:    # When press the button black the hand close

    while (positionB > self.handCurrentB):
        self.handCurrentB = self.handCurrentB + ((speedB+1.0)*5.0)/255.0
        self.fingerTF()

    if positionB != self.handCurrentB:
```

```
            sesgo = positionB - self.handCurrentB
            self.handCurrentB = self.handCurrentB + sesgo
            self.fingerTF()

    if positionB == self.handCurrentB:
        self.fingerTF()



    positionC = self.handTarget.finger_c_position_req
    speedC = self.handTarget.finger_c_speed

    if positionC < self.handCurrentC:     # When press the button white the hand open
        while (positionC < self.handCurrentC):
            self.handCurrentC = self.handCurrentC - ((speedC+1.0)*5.0)/255.0
            self.fingerTF()

        if positionC != self.handCurrentC:
            sesgo = self.handCurrentC - positionC
            self.handCurrentC = self.handCurrentC - sesgo
            self.fingerTF()

    if positionC > self.handCurrentC:     # When press the button black the hand close
        while (positionC > self.handCurrentC):
            self.handCurrentC = self.handCurrentC + ((speedC+1.0)*5.0)/255.0
            self.fingerTF()

        if positionC != self.handCurrentC:
            sesgo = positionC - self.handCurrentC
            self.handCurrentC = self.handCurrentC + sesgo
            self.fingerTF()

    if positionC == self.handCurrentC:
        self.fingerTF()



def fingerTF(self):

    msgJointStates = JointState()
    msgJointStates.header.stamp = rospy.Time.now()
    msgJointStates.header.frame_id = "RobotiqHand"
    msgJointStates.name.append("finger_1_joint_1")
    msgJointStates.name.append("finger_1_joint_2")
    msgJointStates.name.append("finger_1_joint_3")
    msgJointStates.name.append("finger_2_joint_1")
```

```
        msgJointStates.name.append("finger_2_joint_2")
        msgJointStates.name.append("finger_2_joint_3")
        msgJointStates.name.append("finger_middle_joint_1")
        msgJointStates.name.append("finger_middle_joint_2")
        msgJointStates.name.append("finger_middle_joint_3")
        msgJointStates.name.append("palm_finger_1_joint")
        msgJointStates.name.append("palm_finger_2_joint")
        msgJointStates.position.append((self.handCurrentC*math.pi)/360.0)
        msgJointStates.position.append(0.0)
        msgJointStates.position.append(0.0)
        msgJointStates.position.append((self.handCurrentB*math.pi)/360.0)
        msgJointStates.position.append(0.0)
        msgJointStates.position.append(0.0)
        msgJointStates.position.append((self.handCurrentA*math.pi)/360.0)
        msgJointStates.position.append(0.0)
        msgJointStates.position.append(0.0)
        msgJointStates.position.append(-1 * (self.handCurrentScissor*math.pi)/4320.0)
        msgJointStates.position.append((self.handCurrentScissor*math.pi)/4320.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.velocity.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        msgJointStates.effort.append(0.0)
        self.pubJointStates.publish(msgJointStates)

if __name__ == '__main__':
    handstatefake = HandStateFake()
    rospy.init_node('hand_state_fake')
```

```
    rate = rospy.Rate(50)
    while not rospy.is_shutdown():

        if handstatefake.handTarget.individual_finger_control == False and
handstatefake.handTarget.individual_scissor_control == False:    # When the
individual_finger_control is False. All of the fingers move at the same time
            handstatefake.moveAll()

        if handstatefake.handTarget.individual_finger_control == True:    # When the
individual_finger_control is True. Each finger moves separately
            handstatefake.moveEach()

        if handstatefake.handTarget.individual_scissor_control == True:    # When the
individual_sccisor_control is True. The fingers move like a scissor
            handstatefake.moveScissor()


        rate.sleep()
```

14.1.6 hand_state_publisher.py

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('hand_publisher_tf')
roslib.load_manifest('soem_robotiq_drivers')
import rospy
import math
from sensor_msgs.msg import JointState
from soem_robotiq_drivers.msg import RobotiqSModelFeedback

handState = RobotiqSModelFeedback()


class HandStatePublisher:
    ''' Gets the hand position and publishes it in joint_state topic'''
    def __init__(self):
        # Subscription to Robotiq/feedback
        rospy.Subscriber("/robotiq/feedback", RobotiqSModelFeedback, self.processHand)
        self.pubJointStates = rospy.Publisher("/joint_states", JointState)

    def processHand(self, data): # CallBack
        global handState
        handState = data

    def finger(self):
```

```
msgJointStates = JointState()
msgJointStates.header.stamp = rospy.Time.now()
msgJointStates.header.frame_id = "RobotiqHand"
msgJointStates.name.append("finger_1_joint_1")
msgJointStates.name.append("finger_1_joint_2")
msgJointStates.name.append("finger_1_joint_3")
msgJointStates.name.append("finger_2_joint_1")
msgJointStates.name.append("finger_2_joint_2")
msgJointStates.name.append("finger_2_joint_3")
msgJointStates.name.append("finger_middle_joint_1")
msgJointStates.name.append("finger_middle_joint_2")
msgJointStates.name.append("finger_middle_joint_3")
msgJointStates.name.append("palm_finger_1_joint")
msgJointStates.name.append("palm_finger_2_joint")
msgJointStates.position.append((handState.finger_c_position*math.pi)/360)
msgJointStates.position.append(0.0)
msgJointStates.position.append(0.0)
msgJointStates.position.append((handState.finger_b_position*math.pi)/360)
msgJointStates.position.append(0.0)
msgJointStates.position.append(0.0)
msgJointStates.position.append((handState.finger_a_position*math.pi)/360)
msgJointStates.position.append(0.0)
msgJointStates.position.append(0.0)
msgJointStates.position.append(0.0)
msgJointStates.position.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.velocity.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
msgJointStates.effort.append(0.0)
```

```
        msgJointStates.effort.append(0.0)
        self.pubJointStates.publish(msgJointStates)

if __name__ == '__main__':
    handstatepublisher = HandStatePublisher()
    rospy.init_node('hand_state_publisher')

    rate = rospy.Rate(50.0)
    while not rospy.is_shutdown():
            handstatepublisher.finger()
            rate.sleep()
```

14.1.7 haptic_pose_publisher.py

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('lwr_teleop')
import rospy
import tf

from geometry_msgs.msg import PoseStamped

class HapticPosePublisher:
    '''Get position of omni haptic and transform to the reference system that we are
interested '''
    def __init__(self):
        # Subscription to omni2_pose
        rospy.Subscriber("omni2_pose", PoseStamped, self.processOmni2Pose)

        # Publish haptic position in /middle_link base
        self.haptic_publisher = rospy.Publisher('teleop_pose', PoseStamped)
        self.listener = tf.TransformListener()


    def processOmni2Pose(self, pose_link6): # Callback
#   We change the frame_id name to see the pose_link6 frame (in this case omni2_tip)
from middle_link frame
#   In this way we create a relative center in the center of the haptic workspace.
#   Now cover many more positions and configurations.

        pose_link6.header.stamp = rospy.Time()
        pose_link6.header.frame_id = '/omni2_tip'

        base_link = "middle_link"
        self.listener.waitForTransform("/"+base_link, "/omni2_tip", rospy.Time(),
```

```
rospy.Duration(4.0))
        pose_link0 = self.listener.transformPose("/"+base_link, pose_link6)  # middle_link
--> omni2_tip

        pose_link0.header.stamp = rospy.Time.now()
        self.haptic_publisher.publish(pose_link0)



if __name__ == '__main__':
    rospy.init_node('haptic_pose_publisher')

    # Create object HapticPosePublisher
    hapticPosePublisher = HapticPosePublisher()
    rospy.spin()
```

14.1.8 haptic_pose_publisher_with_hand.py

```
#!/usr/bin/env python
import roslib
roslib.load_manifest('lwr_teleop')
roslib.load_manifest('soem_robotiq_drivers')
import rospy
import tf

from geometry_msgs.msg import PoseStamped
from std_msgs.msg import String
from soem_robotiq_drivers.msg import RobotiqSModelCommand



class HapticPosePublisher:
    ''' Get position of omni haptic and transform to the reference system that we are
interested '''
    ''' Publish hand commands to open or close the gripper  '''
    def __init__(self):
        # Subscription to omni2_pose
        rospy.Subscriber("omni2_pose", PoseStamped, self.processOmni2Pose)

        # Subscription to key_publish
        rospy.Subscriber('keyboard_key', String, self.processButtonEvent)

        # Publish haptic position in /middle_link base
        self.haptic_publisher = rospy.Publisher('teleop_pose', PoseStamped)
        self.listener = tf.TransformListener()
```

```
        # Publish in the topic of the HAND
        self.command_pub = rospy.Publisher('/robotiq/command',
RobotiqSModelCommand)



    def processOmni2Pose(self, pose_link6):
#   We change the frame_id name to see the pose_link6 frame (in this case omni2_tip)
from middle_link frame
#   In this way we create a relative center in the center of the haptic workspace.
#   Now cover many more positions and configurations.

        pose_link6.header.stamp = rospy.Time()
        pose_link6.header.frame_id = '/omni2_tip'

        base_link = "middle_link"
        self.listener.waitForTransform("/"+base_link, "/omni2_tip", rospy.Time(),
rospy.Duration(4.0))
        pose_link0 = self.listener.transformPose("/"+base_link, pose_link6)  # middle_link
--> omni2_tip

        pose_link0.header.stamp = rospy.Time.now()
        self.haptic_publisher.publish(pose_link0)


    # Function in which omni2_button is received
    def processButtonEvent(self, data):

        # If a button has pressed
        if data.data == "d":
            print "The hand will open"
            cmd = RobotiqSModelCommand() # Hand command
            cmd.individual_scissor_control = False
            cmd.activate =  True
            cmd.go_to_position = True
            cmd.gripper_position_req = 0
            cmd.gripper_speed = 120
            cmd.gripper_force = 100
            self.command_pub.publish(cmd)

        elif data.data == "e":
            print "The hand will close"
            cmd = RobotiqSModelCommand()
            cmd.individual_scissor_control = False
            cmd.activate =  True
            cmd.go_to_position = True
```

```
        cmd.gripper_position_req = 250
        cmd.gripper_speed = 120
        cmd.gripper_force = 100
        self.command_pub.publish(cmd)


if __name__ == '__main__':
    rospy.init_node('haptic_pose_publisher_with_hand')

    # Create object HapticPosePublisher
    hapticPosePublisher = HapticPosePublisher()
    rospy.spin()
```

14.1.9 lwr_get_ik.py

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('lwr_teleop')
import rospy
import os
from std_msgs.msg import String, Float32
from kinematics_msgs.srv import GetPositionIK, GetPositionIKRequest,
GetKinematicSolverInfo
from arm_navigation_msgs.srv import GetStateValidity, GetStateValidityRequest,
GetPlanningScene, GetPlanningSceneRequest
from geometry_msgs.msg   import PoseStamped
from sensor_msgs.msg import JointState

class LWRGetIK:
    ''' Returns the inverse Kinematic of robot arm. Absolute mode'''

    def __init__(self):
        rospy.init_node('LWR_GetIK', anonymous=True)

        self.alfa = 1.0  # This is the scale, to change the robot precision
        self.message = ""

        #Subscription to teleop_pose
        self.teleop_pose = PoseStamped()
        rospy.Subscriber("teleop_pose", PoseStamped, self.processPoseTeleop)

        #Subscription to JointState for know the current state of joints
        self.joint_state_current = JointState()
        rospy.Subscriber("joint_states", JointState, self.processJointStateCurrent)

        # Subscription to key_publish
        rospy.Subscriber('keyboard_key', String, self.processKeyEvent)
```

```
        #Subscription to the sclae publisher
        rospy.Subscriber('scale_publish', Float32, self.processScale)

        #Publisher for target_joint_state
        self.tjs_publisher_ = rospy.Publisher ('/lwr/target_joint_state',JointState)

        #Publisher for state messages
        self.status_publisher = rospy.Publisher ('/lwr/status',String)

#       ---------------------------------- Get parameter ----------------------------------------
        # my_list is a list object. Each element of the list is a dictionary
        my_list = rospy.get_param('/robot_description_planning/groups')

        if len(my_list) != 0:

            if my_list[0].has_key("name"):
                self.group_name = str(my_list[0]["name"])
            else:
                print "parameter group from
mast_mounted_lwr_robotiq_hand_planning_description.yalm need to contain a name
field"
                self.group_name = ""

#           # we get back the base link from
mast_mounted_lwr_robotiq_hand_planning_description.yalm to create the message for
the kinematic resolver
#           if my_list[0].has_key("base_link"):
#               self.base_link = str(my_list[0]["base_link"])
#           else:
#               print "parameter group from
mast_mounted_lwr_robotiq_hand_planning_description.yalm need to contain a
base_link field"
#               self.base_link = ""
#
#           # we get back the tip link from
mast_mounted_lwr_robotiq_hand_planning_description.yalm to create the message for
the kinematic resolver
#           if my_list[0].has_key("tip_link"):
#               self.tip_link = str(my_list[0]["tip_link"])
#           else:
#               print "parameter group from
mast_mounted_lwr_robotiq_hand_planning_description.yalm need to contain a tip_link
field"
#               self.tip_link = ""
        else:
```

```
            print "There is no parameter from
mast_mounted_lwr_robotiq_hand_planning_description.yalm"
#       ------------------------------------------------------------------------------------

        # Use the service get_ik
    def getIK(self, kin_req1):
        '''Returns the inverse_kinematic solve.
        arg: kin_req: PositionIKRequest
        return: ik_res: RobotState'''
        rospy.loginfo("Wait for service kuka_lwr_kinematics/get_ik ...")
        rospy.wait_for_service("kuka_lwr_kinematics/get_ik")
        try:
            get_ik = rospy.ServiceProxy('kuka_lwr_kinematics/get_ik', GetPositionIK)
            ik_res = get_ik(kin_req1)
            return ik_res
        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
            return ik_res




    def createIKRequest(self, pose_stamped, j_s_current):
        '''Creates the inverse_kinematic request.
        arg: pose_stamped: PoseStamped, j_s_current: JointState
        return: ik_request: PositionIKRequest'''
        rospy.loginfo("Wait for service kuka_lwr_kinematics/get_ik_solver_info ...")
        rospy.wait_for_service("kuka_lwr_kinematics/get_ik_solver_info")
        self.ks_info = None
        try:
            get_kinematic_solver_info =
rospy.ServiceProxy('kuka_lwr_kinematics/get_ik_solver_info',
GetKinematicSolverInfo)
            self.ks_info = get_kinematic_solver_info()

        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
        # Create request
        kin_req = GetPositionIKRequest()
        kin_req.timeout = rospy.Duration(5.0)
        kin_req.ik_request.ik_link_name = "lwr_arm_7_link"  # This line don't affect the
result of IK. Can remove it
        kin_req.ik_request.pose_stamped = pose_stamped   # Set the pose_stamped passed
as argument to the request message

        kin_req.ik_request.ik_seed_state.joint_state = j_s_current
```

```
        kin_req.ik_request.ik_seed_state.joint_state.name =
self.ks_info.kinematic_solver_info.joint_names # joint state names for seed information

        #  Robot State robot_state has in, a JointState and a MultiDOFJointState
        kin_req.ik_request.robot_state.joint_state = j_s_current
        kin_req.ik_request.robot_state.multi_dof_joint_state.joint_names =
self.ks_info.kinematic_solver_info.joint_names  # It isn't necessary to implement

        return kin_req



    def createPlanning(self):
        '''Creates the planning using the enviroment server and the planing service '''
        self.set_planning_scene_diff_name =
'/environment_server/set_planning_scene_diff'
        rospy.loginfo("Wait for service environment_server/set_planning_scene_diff ...")
        rospy.wait_for_service(self.set_planning_scene_diff_name)
        try:
            get_planning_request = GetPlanningSceneRequest()
            self.get_planning_service =
rospy.ServiceProxy(self.set_planning_scene_diff_name, GetPlanningScene)
            self.get_planning_service.call(get_planning_request)
        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
            return -1


    def createSceneValidity(self):
        ''' Creates a scene using the state_validity service'''
        rospy.loginfo("Wait for service
/planning_scene_validity_server/get_state_validity ...")
        rospy.wait_for_service('/planning_scene_validity_server/get_state_validity')
        try:
            self.state_validity =
rospy.ServiceProxy('/planning_scene_validity_server/get_state_validity',
GetStateValidity)

        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
            return -1



    # Function in which teleop_pose is received
```

```
    def processPoseTeleop(self, pose_stamped):
        self.teleop_pose = pose_stamped
        self.teleop_pose.pose.position.x = (self.teleop_pose.pose.position.x * self.alfa)
        self.teleop_pose.pose.position.y = (self.teleop_pose.pose.position.y * self.alfa)
        self.teleop_pose.pose.position.z = (self.teleop_pose.pose.position.z * self.alfa)


      # Function in which joint_state_current is recived
    def processJointStateCurrent(self,joints_state_current):
        if joints_state_current.header.frame_id != "RobotiqHand":   # Check if the joints
state are from the Robot arm
            self.joint_state_current = joints_state_current

    def processKeyEvent(self, data):
        if data.data == "s" or data.data == "f":  # if changes the mode (get_ik // get_ik_AX)
or finish the recorder, the node is shutdown
            os._exit(1)


    def processScale (self, data):
        self.alfa = data.data


if __name__ == '__main__':

    lwrGetIK = LWRGetIK()

    #wait until start reciving data
    tmp_pose = rospy.wait_for_message('teleop_pose', PoseStamped)
    tmp_joints = rospy.wait_for_message('joint_states', JointState)

    lwrGetIK.createPlanning()

    rate = rospy.Rate(50.0) #Before was 10
    while not rospy.is_shutdown():

        lwrGetIK.createSceneValidity()


        #Calculate the Inverse Kinematics
        ik_request = lwrGetIK.createIKRequest(lwrGetIK.teleop_pose,
lwrGetIK.joint_state_current)
        ik_result  = lwrGetIK.getIK(ik_request)


        if ik_result.error_code.val == -1:
```

```
            lwrGetIK.message = "PLANNING_FAILED"

        elif ik_result.error_code.val == 1:
            print "IK acquired"
            lwrGetIK.message = ""
            state_req = GetStateValidityRequest()
            state_req.group_name = lwrGetIK.group_name #'kuka_arm'
            state_req.robot_state.joint_state.name = ['lwr_arm_joint_0', 'lwr_arm_joint_1',
'lwr_arm_joint_2', 'lwr_arm_joint_3', 'lwr_arm_joint_4', 'lwr_arm_joint_5',
'lwr_arm_joint_6']
            state_req.check_collisions = True
            state_req.robot_state.joint_state.position = ik_result.solution.joint_state.position

            state_val_res = lwrGetIK.state_validity.call(state_req)   # Check if the status is
valid

            if state_val_res.error_code.val == state_val_res.error_code.SUCCESS:
                ik_result.solution.joint_state.header.stamp = rospy.Time.now()
                lwrGetIK.tjs_publisher_.publish(ik_result.solution.joint_state)

            else:
                lwrGetIK.message = "Collision"

        else:
            print "some other error: " , ik_result.error_code.val
            lwrGetIK.message = "No Ik result"
        lwrGetIK.status_publisher.publish(lwrGetIK.message)
        rate.sleep()
```

14.1.10 lwr_get_ik_AX.py

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('lwr_teleop')
import rospy
import os
import PyKDL

from std_msgs.msg import String, Float32
from kinematics_msgs.srv import GetPositionIK, GetPositionFK,
GetPositionFKRequest, GetKinematicSolverInfo, GetPositionIKResponse
from kinematics_msgs.msg import PositionIKRequest
from geometry_msgs.msg   import PoseStamped
from sensor_msgs.msg import JointState
from phantom_omni.msg import PhantomButtonEvent
from arm_navigation_msgs.srv import GetStateValidity, GetStateValidityRequest,
GetPlanningScene, GetPlanningSceneRequest
```

```
class LWRGetIKAX:
    ''' Returns the inverse Kinematic of robot arm. Relative mode '''

    def __init__(self):
        rospy.init_node('LWR_GetIK_AX', anonymous=True)
        #Subscription to teleop_pose
        self.x_omni = PoseStamped()
        rospy.Subscriber("teleop_pose", PoseStamped, self.processPoseTeleop)

        #Subscription to JointState for know the current state of joints
        self.joint_state_current = JointState()
        rospy.Subscriber("joint_states", JointState, self.processJointStateCurrent)

        #Subscription to omni2_button for determinate if the button has been pressed
        self.buttonPressed = PhantomButtonEvent()
        rospy.Subscriber("omni2_button", PhantomButtonEvent, self.processButton)

        # Subscription to key_publish
        rospy.Subscriber('keyboard_key', String, self.processKeyEvent)

        #Subscription to the sclae publisher
        rospy.Subscriber('scale_publish', Float32, self.processScale)

        #Publisher for target_joint_state
        self.tjs_publisher_ = rospy.Publisher ('/lwr/target_joint_state',JointState)

        #Publisher for state messages
        self.status_publisher = rospy.Publisher ('/lwr/status',String)

        # This is the factor of precision
        self.alfa = 1.0
        # This is the X for calculate IK
        self.x_to_ik = PoseStamped()

        self.flag = False
        self.message = ""

        self.x_omni_last = PoseStamped()
        self.x_robot_last = PoseStamped()

#       ----------------------------------Get parameter----------------------------------------
        # my_list is a list object. Each element of the list is a dictionary
        my_list = rospy.get_param('/robot_description_planning/groups')
```

```
        if len(my_list) != 0:

            if my_list[0].has_key("name"):
                self.group_name = str(my_list[0]["name"])
            else:
                print "parameter group from
mast_mounted_lwr_robotiq_hand_planning_description.yalm need to contain a name
field"
                self.group_name = ""

            # we get back the base link from
mast_mounted_lwr_robotiq_hand_planning_description.yalm to create the message for
the kinematic resolver
            if my_list[0].has_key("base_link"):
                self.base_link = str(my_list[0]["base_link"])
            else:
                print "parameter group from
mast_mounted_lwr_robotiq_hand_planning_description.yalm need to contain a
base_link field"
                self.base_link = ""

            # we get back the tip link from
mast_mounted_lwr_robotiq_hand_planning_description.yalm to create the message for
the kinematic resolver
#           if my_list[0].has_key("tip_link"):
#               self.tip_link = str(my_list[0]["tip_link"])
#           else:
#               print "parameter group from
mast_mounted_lwr_robotiq_hand_planning_description.yalm need to contain a tip_link
field"
#               self.tip_link = ""
        else:
            print "There is no parameter from
mast_mounted_lwr_robotiq_hand_planning_description.yalm"
#       ------------------------------------------------------------------------------------


    def processScale (self, data):
        self.alfa = data.data

    def processButton(self, data):
        self.buttonPressed = data

     # Function in which teleop_pose is received
    def processPoseTeleop(self, pose_stamped):
```

```
        self.x_omni = pose_stamped


    # Function in which joint_state_current is recived
    def processJointStateCurrent(self,joints_state_current):
        if joints_state_current.header.frame_id != "RobotiqHand":   # Check if the joints
state are from the Robot arm
            self.joint_state_current = joints_state_current

    def processKeyEvent(self, data):
        if data.data == "w" or data.data == "f":  # if i change the mode (get_ik //
get_ik_AX) or finish the recorder, the node is shutdown
            os._exit(1)


    # Get the pose (fk_res) of the last robot's link using the get_fk service
    def getFk(self):
        rospy.loginfo("Wait for service kuka_lwr_kinematics/get_fk ...")
        rospy.wait_for_service("kuka_lwr_kinematics/get_fk")
        position_fk_request = self.createFKRequest()
        try:
            get_fk = rospy.ServiceProxy('kuka_lwr_kinematics/get_fk', GetPositionFK)
            fk_res = get_fk(position_fk_request)
            return fk_res
        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
            return -1



    # Create a FK Request
    def createFKRequest(self):
        # Get kinematic solver information
        rospy.loginfo("Wait for service kuka_lwr_kinematics/get_fk_solver_info ...")
        rospy.wait_for_service("kuka_lwr_kinematics/get_fk_solver_info")
        try:
            get_kinematic_solver_info =
rospy.ServiceProxy('kuka_lwr_kinematics/get_fk_solver_info',
GetKinematicSolverInfo)
            ks_info_1 = get_kinematic_solver_info()

        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
        # Create request
        fk_request = GetPositionFKRequest()
        fk_request.header.frame_id = self.base_link #"/calib_lwr_arm_base_link"
```

```
        fk_request.fk_link_names = "lwr_arm_7_link"
        fk_request.robot_state.joint_state = self.joint_state_current
        fk_request.robot_state.multi_dof_joint_state.joint_names =
ks_info_1.kinematic_solver_info.joint_names;
        fk_request.robot_state.joint_state.position = self.joint_state_current.position
        return fk_request



    def calculateAX(self):  # Subtracting a fixed value

        if self.buttonPressed.white_button == 0 and self.buttonPressed.grey_button == 0:
#There isn't a button pressed
            self.flag = False
            self.x_to_ik = self.getFk().pose_stamped[0] # Here the robot doesn't move

        else:
            if self.flag == False:
                self.x_omni_last = self.x_omni
                self.x_robot_last = self.getFk().pose_stamped[0]
                self.flag = True

            elif self.flag == True:
                # Get the Ax_omni
                Ax_omni = PoseStamped()
                Ax_omni.pose.position.x = self.x_omni.pose.position.x -
self.x_omni_last.pose.position.x
                Ax_omni.pose.position.y = self.x_omni.pose.position.y -
self.x_omni_last.pose.position.y
                Ax_omni.pose.position.z = self.x_omni.pose.position.z -
self.x_omni_last.pose.position.z
                # R1 * R2 = R_result  --> R2 = R1^-1 * R_result
                rotation_omni_1 =
PyKDL.Rotation.Quaternion(self.x_omni_last.pose.orientation.x,
self.x_omni_last.pose.orientation.y, self.x_omni_last.pose.orientation.z,
self.x_omni_last.pose.orientation.w)
                rotation_omni_result =
PyKDL.Rotation.Quaternion(self.x_omni.pose.orientation.x,
self.x_omni.pose.orientation.y, self.x_omni.pose.orientation.z,
self.x_omni.pose.orientation.w)

                rotation_omni_2 = rotation_omni_1.Inverse() * rotation_omni_result
#               rotation_term_2.GetQuaternion()  # get quaternion result
#               rotation_term_2.GetRPY()  # get roll pitch yaw result
#               rotation_term_2.GetEulerZYX()   # get euler angle result in zyx format
```

```
            self.x_to_ik.pose.position.x = (Ax_omni.pose.position.x * self.alfa) +
self.x_robot_last.pose.position.x
            self.x_to_ik.pose.position.y = (Ax_omni.pose.position.y * self.alfa) +
self.x_robot_last.pose.position.y
            self.x_to_ik.pose.position.z = (Ax_omni.pose.position.z * self.alfa) +
self.x_robot_last.pose.position.z
            rotation_robot_1 =
PyKDL.Rotation.Quaternion(self.x_robot_last.pose.orientation.x ,
self.x_robot_last.pose.orientation.y , self.x_robot_last.pose.orientation.z ,
self.x_robot_last.pose.orientation.w)
            rotation_robot_result = rotation_robot_1 * rotation_omni_2
            self.x_to_ik.pose.orientation.x = rotation_robot_result.GetQuaternion()[0] #*
-1.0
            self.x_to_ik.pose.orientation.y = rotation_robot_result.GetQuaternion()[1] #*
-1.0
            self.x_to_ik.pose.orientation.z = rotation_robot_result.GetQuaternion()[2] #*
-1.0
            self.x_to_ik.pose.orientation.w = rotation_robot_result.GetQuaternion()[3]

#             print "x: " , self.x_to_ik.pose.position.x , "    y: " ,
self.x_to_ik.pose.position.y , "   z: " , self.x_to_ik.pose.position.z
#             print "Ox: " , self.x_to_ik.pose.orientation.x , "    Oy: " ,
self.x_to_ik.pose.orientation.y , "   Oz: " , self.x_to_ik.pose.orientation.z , "   Ow: " ,
self.x_to_ik.pose.orientation.w



    #Use the service get_ik
  def getIK(self, position_ik_request, timeout):
    '''Returns the inverse_kinematic solve.
    arg: kin_req: PositionIKRequest
    return: ik_res: RobotState'''
    rospy.loginfo("Wait for service kuka_lwr_kinematics/get_ik ...")
    rospy.wait_for_service("kuka_lwr_kinematics/get_ik")
    try:
      get_ik = rospy.ServiceProxy('kuka_lwr_kinematics/get_ik', GetPositionIK)
      ik_res = get_ik(position_ik_request,rospy.Duration(timeout))
      print type(ik_res)
      return ik_res
    except rospy.ServiceException, e:
      print "Service call failed: %s"%e
      ik_res_error = GetPositionIKResponse()
      ik_res_error.error_code = -1
      return ik_res_error
```

```
    def createIKRequest(self, pose_stamped, j_s_current):
        '''Creates the inverse_kinematic request.
        arg: pose_stamped: PoseStamped, j_s_current: JointState
        return: ik_request: PositionIKRequest'''
        rospy.loginfo("Wait for service kuka_lwr_kinematics/get_ik_solver_info ...")
        rospy.wait_for_service("kuka_lwr_kinematics/get_ik_solver_info")
        ks_info = None
        try:
            get_kinematic_solver_info =
rospy.ServiceProxy('kuka_lwr_kinematics/get_ik_solver_info',
GetKinematicSolverInfo)
            ks_info = get_kinematic_solver_info()

        except rospy.ServiceException, e:
            print "Service call failed: %s"%e

        # Create request
        ik_request = PositionIKRequest()
        ik_request.ik_link_name = "lwr_arm_7_link"
        ik_request.pose_stamped = pose_stamped   # Set the pose_stamped passed as
argument to the request message
        ik_request.ik_seed_state.joint_state.name =
ks_info.kinematic_solver_info.joint_names # joint state names for seed information
        ik_request.ik_seed_state.joint_state = j_s_current
        #  Robot State robot_state has in, a JointState and a MultiDOFJointState
        ik_request.robot_state.joint_state = j_s_current

        return ik_request



    def createPlanning(self):
        '''Creates the planning using the enviroment server and the planing service '''
        self.set_planning_scene_diff_name =
'/environment_server/set_planning_scene_diff'
        rospy.loginfo("Wait for service environment_server/set_planning_scene_diff ...")
        rospy.wait_for_service(self.set_planning_scene_diff_name)
        try:
            get_planning_request = GetPlanningSceneRequest()
            self.get_planning_service =
rospy.ServiceProxy(self.set_planning_scene_diff_name, GetPlanningScene)
            self.get_planning_service.call(get_planning_request)
        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
```

```
            return -1


    def createSceneValidity(self):
        ''' Creates a scene using the state_validity service'''
        rospy.loginfo("Wait for service
/planning_scene_validity_server/get_state_validity ...")
        rospy.wait_for_service('/planning_scene_validity_server/get_state_validity')
        try:
            self.state_validity =
rospy.ServiceProxy('/planning_scene_validity_server/get_state_validity',
GetStateValidity)

        except rospy.ServiceException, e:
#           print "Service call failed: %s"%e
            return -1




if __name__ == '__main__':

    lwrGetIK = LWRGetIKAX()

    #wait until start reciving data
    tmp_pose = rospy.wait_for_message('teleop_pose', PoseStamped)
    tmp_joints = rospy.wait_for_message('joint_states', JointState)

    lwrGetIK.createPlanning()

    for num in range(1,10000):
        lwrGetIK.message = "Wait !!"
        lwrGetIK.status_publisher.publish(lwrGetIK.message)


    rate = rospy.Rate(50.0) #Before was 10
    while not rospy.is_shutdown():

        lwrGetIK.createSceneValidity()

        #Calculate the Forward Kinematics
        lwrGetIK.calculateAX()

        #Calculate the Inverse Kinematics
        ik_request = lwrGetIK.createIKRequest(lwrGetIK.x_to_ik,
lwrGetIK.joint_state_current)
```

```
        ik_result  = lwrGetIK.getIK(ik_request,5.0)


        if ik_result.error_code == -1:
            lwrGetIK.message = "No_Planning"

        elif ik_result.error_code.val == 1:
            print "IK acquired"
            lwrGetIK.message = ""
            state_req = GetStateValidityRequest()
            state_req.group_name = lwrGetIK.group_name
            state_req.robot_state.joint_state.name = ['lwr_arm_joint_0', 'lwr_arm_joint_1',
'lwr_arm_joint_2', 'lwr_arm_joint_3', 'lwr_arm_joint_4', 'lwr_arm_joint_5',
'lwr_arm_joint_6']
            state_req.check_collisions = True
            state_req.robot_state.joint_state.position = ik_result.solution.joint_state.position

            state_val_res = lwrGetIK.state_validity.call(state_req)   # Check if the status is
valid

            if state_val_res.error_code.val == state_val_res.error_code.SUCCESS:
                ik_result.solution.joint_state.header.stamp = rospy.Time.now()
                lwrGetIK.tjs_publisher_.publish(ik_result.solution.joint_state)

            else:
                lwrGetIK.message = "Collision"

        else:
            print "some other error: " , ik_result.error_code.val
            lwrGetIK.message = "No Ik result"
        lwrGetIK.status_publisher.publish(lwrGetIK.message)
        rate.sleep()
```

14.1.11 lwr_learning_recorder.py

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('lwr_teleop')
import rospy
import numpy

from sensor_msgs.msg import JointState
from std_msgs.msg import String

class LwrLearningRecorder:

    def __init__(self):
```

```
        rospy.Subscriber('keyboard_key', String, self.processKeyEvent)
        rospy.Subscriber('rec_file_name', String, self.processFileName)

        self.filename =
"/home/across/rosstacks/lwr/lwr_robot/lwr_teleop/recording/default.npz"

    def processKeyEvent(self, data):

        if data.data == "l": # Start REC
            self.data_msr_jnt_pos = [numpy.empty(0) for i in range(7)]
            self.fri_joint_state_sub = rospy.Subscriber("/joint_states", JointState,
self.processJointStateCurrent) #Subscription to JointState for know the current state of
joints


        if data.data == "f": # Stop REC
            self.fri_joint_state_sub.unregister()
            self.fri_joint_state_sub = None
            self.stop_and_save(self.filename)



    # Start the REC
    def processJointStateCurrent(self, data):
        if(self.data_msr_jnt_pos):
            for i in range(7):
                self.data_msr_jnt_pos[i] = numpy.append(self.data_msr_jnt_pos[i],
data.position[i])



    def processFileName(self, data):
        self.filename = data.data

    # Stop the REC
    def stop_and_save(self, data):
        print "filename: ", data
        self.file = open(data, 'wb')
        numpy.savez(self.file, *self.data_msr_jnt_pos)
        self.file.close()
        self.file = None
        self.data_msr_jnt_pos = None
```

```
if __name__ == '__main__':
    rospy.init_node('lwr_learning_recorder')
    lwr_learning_recorder = LwrLearningRecorder()
    rospy.spin()
```

14.1.12 lwr_learning_recorder_with_hand.py

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('lwr_teleop')
roslib.load_manifest('soem_robotiq_drivers')
import rospy
import numpy

from sensor_msgs.msg import JointState
from std_msgs.msg import String
from soem_robotiq_drivers.msg import RobotiqSModelCommand

class LwrLearningRecorder:

    def __init__(self):

        rospy.Subscriber('keyboard_key', String, self.processKeyEvent)
        rospy.Subscriber('rec_file_name', String, self.processFileName)
        rospy.Subscriber("/robotiq/command", RobotiqSModelCommand,
self.processHand)

        self.filename =
"/home/across/rosstacks/lwr/lwr_robot/lwr_teleop/recording/default.npz"
        self.hand_pose = 0

    def processKeyEvent(self, data):

        if data.data == "l": # Start REC
            self.data_msr_jnt_pos = [numpy.empty(0) for i in range(8)]
            self.fri_joint_state_sub = rospy.Subscriber("/joint_states", JointState,
self.processJointStateCurrent) #Subscription to JointState for know the current state of
joints


        if data.data == "f": # Stop REC
            self.fri_joint_state_sub.unregister()
            self.fri_joint_state_sub = None
            self.stop_and_save(self.filename)


    def processHand(self, data):
```

```
        self.hand_pose = data.gripper_position_req


    # Start the REC
    def processJointStateCurrent(self, data):
        if(self.data_msr_jnt_pos):
            if data.header.frame_id == "":
                for i in range(7):
                    self.data_msr_jnt_pos[i] = numpy.append(self.data_msr_jnt_pos[i],
data.position[i])
                self.data_msr_jnt_pos[7] = numpy.append(self.data_msr_jnt_pos[7],
self.hand_pose)
                print self.hand_pose


    def processFileName(self, data):
        self.filename = data.data

    # Stop the REC
    def stop_and_save(self, data):
        print "filename: ", data
        self.file = open(data, 'wb')
        numpy.savez(self.file, *self.data_msr_jnt_pos)
        self.file.close()
        self.file = None
        self.data_msr_jnt_pos = None



if __name__ == '__main__':
    rospy.init_node('lwr_learning_recorder_with_hand')
    lwr_learning_recorder = LwrLearningRecorder()
    rospy.spin()
```

14.1.13 lwr_learning_reproducer.py

```
#!/usr/bin/env python
""" Write a filename as argument """
import roslib; roslib.load_manifest('lwr_teleop')
import rospy
import numpy
import sys, getopt

from sensor_msgs.msg import JointState
from std_msgs.msg import String
```

```
def main ():

    # Accessing Command Line Arguments
    try:
        opts, args = getopt.getopt(sys.argv[1:], "h", ["help"])
    except getopt.error, msg:
        print msg
        print "for help use --help"
        sys.exit(2)
    # process options
    for o, a in opts:
        if o in ("-h", "--help"):
            print __doc__
            sys.exit(0)
    # process arguments
    for arg in args:
        DATA_FILENAME = arg


#    DATA_FILENAME = '/home/across/data.npz'

    rospy.init_node('lwr_learning_reproducer')

    rospy.loginfo('Loading the data from file: ' + DATA_FILENAME)
    npzfile = numpy.load(DATA_FILENAME)
    joint_data = []
    for i in range(7):
        joint_data.append(npzfile['arr_' + str(i)])


    rospy.loginfo('Preparing joint position command publisher')
    joint_position_command_pub = rospy.Publisher('/lwr/target_joint_state',JointState)
#Publisher for target_joint_state
    status_publisher = rospy.Publisher('/lwr/status',String)


    joint_position_command = JointState()
    joint_position_command.name = ['lwr_arm_joint_0', 'lwr_arm_joint_1',
'lwr_arm_joint_2', 'lwr_arm_joint_3', 'lwr_arm_joint_4', 'lwr_arm_joint_5',
'lwr_arm_joint_6']



    rospy.loginfo('Welcome to the trajectory reproduction test')
#    rospy.loginfo('Follow the instructions and everything should go fine')
```

```
#    rospy.loginfo('Enable power in the robot, move some joint a bit and DON\'T LET
GO THE POWER!')
#    raw_input('Hit any key when ready')

    rospy.loginfo('Initializing joint position command to current position')
    current_position = rospy.wait_for_message('/joint_states', JointState)


    joint_position_command.position = current_position.position
    joint_position_command.header.stamp = rospy.Time.now()
    joint_position_command_pub.publish(joint_position_command)

    rospy.loginfo('Preparing to move the robot to the recorded data initial position.')
#    rospy.loginfo('PLEASE ENTER FRI COMMAND MODE BEFORE
PROCEEDING')
#    raw_input('Hit any key when ready')
#    rospy.loginfo('Homing in 5...')
#    rospy.sleep(1.0)
#    rospy.loginfo('4...')
#    rospy.sleep(1.0)
#    rospy.loginfo('3...')
#    rospy.sleep(1.0)
#    rospy.loginfo('2...')
#    rospy.sleep(1.0)
#    rospy.loginfo('1...')
#    rospy.sleep(1.0)

    initial_position = [joint[0] for joint in joint_data] # get the first data of each file.
(There are seven files. One for each joint)

    rospy.loginfo('Slowly moving the robot to the initial position (in 5 sec.): ' +
str(initial_position))
    current_position = list(current_position.position)
    joint_position_command.position = list(current_position)


    rate = rospy.Rate(50)
    for t in range(5*50):
        for i in range(7):
            joint_position_command.position[i] = current_position[i] + (initial_position[i] -
current_position[i])*(t/(5.*50.))
        joint_position_command.header.stamp = rospy.Time.now()
        joint_position_command_pub.publish(joint_position_command)
        rate.sleep()

    joint_position_command.position = initial_position
```

```
    joint_position_command.header.stamp = rospy.Time.now()
    joint_position_command_pub.publish(joint_position_command)


    rospy.loginfo('Ready to reproduce recorded trajectory')
#   raw_input('Hit any key when ready')
#   rospy.loginfo('Reproducing in 5...')
#   rospy.sleep(1.0)
#   rospy.loginfo('4...')
#   rospy.sleep(1.0)
#   rospy.loginfo('Reproducing in 3...')
#   rospy.sleep(1.0)
#   rospy.loginfo('2...')
#   rospy.sleep(1.0)
#   rospy.loginfo('1...')
#   rospy.sleep(1.0)

    for t in range(len(joint_data[0])):
        for i in range(7):
            joint_position_command.position[i] = joint_data[i][t]
        joint_position_command.header.stamp = rospy.Time.now()
        joint_position_command_pub.publish(joint_position_command)
        rate.sleep()

    status_publisher.publish("")
    rospy.loginfo('Hope everything was nice and smooth!')


if __name__ == "__main__":
    main()
```

14.1.14 lwr_learning_reproducer_with_hand.py

```
#!/usr/bin/env python
""" Write a filename as argument """
import roslib; roslib.load_manifest('lwr_teleop')
roslib.load_manifest('soem_robotiq_drivers')
import rospy
import numpy
import sys, getopt

from sensor_msgs.msg import JointState
from std_msgs.msg import String
from soem_robotiq_drivers.msg import RobotiqSModelCommand

def main ():
```

```
    # Accessing Command Line Arguments
    try:
        opts, args = getopt.getopt(sys.argv[1:], "h", ["help"])
    except getopt.error, msg:
        print msg
        print "for help use --help"
        sys.exit(2)
    # process options
    for o, a in opts:
        if o in ("-h", "--help"):
            print __doc__
            sys.exit(0)
    # process arguments
    for arg in args:
        DATA_FILENAME = arg


#    DATA_FILENAME = '/home/across/data.npz'

    rospy.init_node('lwr_learning_reproducer_with_hand')

    rospy.loginfo('Loading the data from file: ' + DATA_FILENAME)
    npzfile = numpy.load(DATA_FILENAME)
    joint_data = []
    for i in range(8):
        joint_data.append(npzfile['arr_' + str(i)])


    rospy.loginfo('Preparing joint position command publisher')
    joint_position_command_pub = rospy.Publisher('/lwr/target_joint_state',JointState)
#Publisher for target_joint_state
    status_publisher = rospy.Publisher('/lwr/status',String)
    command_pub = rospy.Publisher('/robotiq/command', RobotiqSModelCommand)

#  arm
    joint_position_command = JointState()
    joint_position_command.name = ['lwr_arm_joint_0', 'lwr_arm_joint_1',
'lwr_arm_joint_2', 'lwr_arm_joint_3', 'lwr_arm_joint_4', 'lwr_arm_joint_5',
'lwr_arm_joint_6']


#  hand
    cmd = RobotiqSModelCommand()
    cmd.individual_scissor_control = False
    cmd.activate =  True
```

```
    cmd.go_to_position = True
    cmd.gripper_speed = 100
    cmd.gripper_force = 0




    rospy.loginfo('Welcome to the trajectory reproduction test')
#   rospy.loginfo('Follow the instructions and everything should go fine')
#   rospy.loginfo('Enable power in the robot, move some joint a bit and DON\'T LET
GO THE POWER!')
#   raw_input('Hit any key when ready')

    rospy.loginfo('Initializing joint position command to current position')
    current_position = rospy.wait_for_message('/joint_states', JointState)


    joint_position_command.position = current_position.position
    joint_position_command.header.stamp = rospy.Time.now()
    joint_position_command_pub.publish(joint_position_command)

    rospy.loginfo('Preparing to move the robot to the recorded data initial position.')
#   rospy.loginfo('PLEASE ENTER FRI COMMAND MODE BEFORE
PROCEEDING')
#   raw_input('Hit any key when ready')
#   rospy.loginfo('Homing in 5...')
#   rospy.sleep(1.0)
#   rospy.loginfo('4...')
#   rospy.sleep(1.0)
#   rospy.loginfo('3...')
#   rospy.sleep(1.0)
#   rospy.loginfo('2...')
#   rospy.sleep(1.0)
#   rospy.loginfo('1...')
#   rospy.sleep(1.0)

    initial_position = [joint[0] for joint in joint_data] # get the first data of each file.
(There are seven files. One for each joint)

    rospy.loginfo('Slowly moving the robot to the initial position (in 5 sec.): ' +
str(initial_position))
    current_position = list(current_position.position)
    joint_position_command.position = list(current_position)
```

```
    # publish the first position of the arm slowly. In 5 seconds
    rate = rospy.Rate(50)
    for t in range(5*50):
        for i in range(7):
            joint_position_command.position[i] = current_position[i] + (initial_position[i] -
current_position[i])*(t/(5.*50.))
        joint_position_command.header.stamp = rospy.Time.now()
        joint_position_command_pub.publish(joint_position_command)
        rate.sleep()

    joint_position_command.position = initial_position
    joint_position_command.header.stamp = rospy.Time.now()
    joint_position_command_pub.publish(joint_position_command)


    cmd.gripper_position_req = int(initial_position[7])
    command_pub.publish(cmd)


    rospy.loginfo('Ready to reproduce recorded trajectory')
#    raw_input('Hit any key when ready')
#    rospy.loginfo('Reproducing in 5...')
#    rospy.sleep(1.0)
#    rospy.loginfo('4...')
#    rospy.sleep(1.0)
#    rospy.loginfo('Reproducing in 3...')
#    rospy.sleep(1.0)
#    rospy.loginfo('2...')
#    rospy.sleep(1.0)
#    rospy.loginfo('1...')
#    rospy.sleep(1.0)

    for t in range(len(joint_data[0])):
        for i in range(7):
            joint_position_command.position[i] = joint_data[i][t]
            cmd.gripper_position_req = int(joint_data[7][t])
        joint_position_command.header.stamp = rospy.Time.now()
        joint_position_command_pub.publish(joint_position_command)
        command_pub.publish(cmd)
        rate.sleep()

    status_publisher.publish("")
    rospy.loginfo('Hope everything was nice and smooth!')


if __name__ == "__main__":
```

```
main()
```

14.1.15 Quaternion_tranform.py

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('lwr_teleop')
import rospy
import PyKDL


class Convert:
    ''' Convertions about rotation'''

    def __init__(self):
        rospy.init_node('Translate', anonymous=True)


    def calculate(self):  # Subtracting a fixed value

            # R1 * R2 = R_result  --> R2 = R1^-1 * R_result

        rotation_omni_1 = PyKDL.Rotation.Quaternion(0.4095, 0.2800, 0.8605, 0.1145)
#       rotation_omni_result = PyKDL.Rotation.Quaternion(-0.699606367281,
0.71379433505, 0.0229513214657, 0.0228432693105)

#       rotation_omni_2 = rotation_omni_1.Inverse() * rotation_omni_result
#       print "Quaternion: ", rotation_omni_2.GetQuaternion()  # get quaternion result
        print "RPY: ", rotation_omni_1.GetRPY()

#       print "RPY: ", rotation_omni_2.GetRPY()  # get roll pitch yaw result
#       print "Euler: ", rotation_omni_2.GetEulerZYX()   # get euler angle result in zyx
format


if __name__ == '__main__':

    convert = Convert()
    convert.calculate()
```

14.1.16 scale_publisher.py

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('lwr_teleop')
import rospy
from std_msgs.msg import String
from std_msgs.msg import Float32

class Scale:


    def __init__(self):
        self.scale_default = 1.0
        self.scale_max = 4.0
        self.scale_min = 0.1
        # Subscription to key_publish
        rospy.Subscriber('keyboard_key', String, self.processButtonEvent)
        self.scale_publish = rospy.Publisher('scale_publish', Float32)


    def processButtonEvent(self, data):

        if data.data == "q":
            if "{0:.2f}".format(self.scale_default) < "{0:.2f}".format(self.scale_max):
                self.scale_default = self.scale_default + 0.1
                print self.scale_default

        elif data.data == "a":
            if "{0:.2f}".format(self.scale_default) > "{0:.2f}".format(self.scale_min):
                self.scale_default = self.scale_default - 0.1
                print self.scale_default



if __name__ == '__main__':
        rospy.init_node('scale_publisher')
        scale = Scale() # Create object Scale

        rate = rospy.Rate(4.0) #Before was 10
        while not rospy.is_shutdown():
                scale.scale_publish.publish(scale.scale_default)
                rate.sleep()
```

14.1.17 lwr_omni.launch

```
<?xml version="1.0"?>
<launch>
      <!-- Launch phantom omni -->
      <node pkg="tf" type="static_transform_publisher" name="omni_teleop_base"
args="0 0 0.85 0 0 0 calib_lwr_arm_base_link middle_link 100" />

      <node pkg="tf" type="static_transform_publisher"
name="omni_teleop_tip_link_joint" args="0 0 0 3.0382387469125667
-1.0676596682912134 0.5293656353976447 omni2_link6 omni2_tip 100" />

      <node pkg="tf" type="static_transform_publisher"
name="omni_teleop_base_middle" args=" -0.261425868874 -0.000489295063002
0.169954509016 0 -1.5707 -3.141592 middle_link omni2 100" />

      <include file="$(find phantom_omni)/omni2.launch"/>

</launch>
```

14.1.18 lwr_omni_teleop.launch

```
<?xml version="1.0"?>
<launch>

   <!-- Launch the lwr_fake on Rviz -->
  <include file="$(find lwr_bringup)/launch/mast_mounted_lwr_arm_fake.launch"/>

   <!-- Launch phantom omni and static_transform_publisher -->
  <include file="$(find lwr_teleop)/launch/lwr_omni.launch"/>

   <!-- Launch arm_navigation -->
  <include file="$(find
mast_mounted_lwr_arm_navigation)/launch/lwr_environment_server.launch"/>
  <include file="$(find
mast_mounted_lwr_arm_navigation)/launch/mast_mounted_lwr_planning_environment
.launch"/>

   <!-- Launch Kinematics Mode Absolut/Relative -->
  <node name="lwr_change_ik" pkg="lwr_teleop" type="change_mode.py" />

   <!-- Launch the haptic pose publisher -->
  <node name="haptic_pose_publisher" pkg="lwr_teleop"
type="haptic_pose_publisher.py" />

   <!-- Launch the scale publisher -->
```

```
  <node name="scale_publisher" pkg="lwr_teleop" type="scale_publisher.py" />

    <!-- Launch get Forward Kinematics -->
  <node name="get_forward_kinematics" pkg="lwr_teleop" type="get_FK.py"/>

    <!-- Launch the GUI -->
  <node name="lwr_GUI" pkg="lwr_teleop" type="gui_teleop.py" required="true"/>
<!-- With required=True all the nodes in the launch file close, when this node is closed
-->

    <!-- Launch the position recorder. (Machine learning) -->
  <node name="lwr_learning_recorder" pkg="lwr_teleop"
type="lwr_learning_recorder.py"/>

</launch>
```

14.1.19 lwr_omni_teleop_with_hand_real.launch

```
<?xml version="1.0"?>
<launch>

  <!-- Launch the lwr_fake on Rviz with HAND Robotiq -->
  <include file="$(find
lwr_bringup)/launch/mast_mounted_lwr_arm_robotiq_hand.launch"/>

  <!-- Launch phantom omni and static_transform_publisher-->
  <include file="$(find lwr_teleop)/launch/lwr_omni.launch"/>

  <!-- Launch arm_navigation -->
  <include file="$(find
mast_mounted_lwr_robotiq_hand_arm_navigation)/launch/mast_mounted_lwr_robotiq
_hand_planning_environment.launch"/>
  <include file="$(find
mast_mounted_lwr_robotiq_hand_arm_navigation)/launch/lwr_environment_server.lau
nch"/>


  <!-- Launch the haptic pose publisher -->
  <node name="haptic_pose_publisher" pkg="lwr_teleop"
type="haptic_pose_publisher_with_hand.py" />

  <!-- Launch Kinematics Mode Absolut/Relative -->
  <node name="lwr_change_ik" pkg="lwr_teleop" type="change_mode.py" />

  <!-- Launch the sclae publisher -->
  <node name="scale_publisher" pkg="lwr_teleop" type="scale_publisher.py" />
```

```
  <!-- Launch the HAND state publisher -->
  <node name="hand_state_publisher" pkg="lwr_teleop"
type="hand_state_publisher.py" />

  <!-- Launch get Forward Kinematics -->
  <node name="get_forward_kinematics" pkg="lwr_teleop" type="get_FK.py"/>

  <!-- Launch the GUI -->
  <node name="lwr_GUI" pkg="lwr_teleop" type="gui_teleop.py" required="true"/>
<!-- With "required=True" all the nodes in the launch file will close, when this node
closes -->

  <!-- Launch the position recorder. (Machine learning) -->
  <node name="lwr_learning_recorder" pkg="lwr_teleop"
type="lwr_learning_recorder_with_hand.py"/>

  <!-- Launch the force_feedback node. Get the force and position of real robot and send
to the Omni -->
<!--  <node name="force_feedback" pkg="lwr_teleop" type="force_feedback.py"/>-->

</launch>
```

14.1.20 omni.cpp

```
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <tf/transform_broadcaster.h>
#include <geometry_msgs/Wrench.h>
#include <geometry_msgs/Vector3.h>
#include <geometry_msgs/WrenchStamped.h>

#include <string.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <sstream>

#include <HL/hl.h>
#include <HD/hd.h>
#include <HDU/hduError.h>
#include <HDU/hduVector.h>
#include <HDU/hduMatrix.h>

#include "phantom_omni/PhantomButtonEvent.h"
#include "phantom_omni/OmniFeedback.h"
```

```
#include <pthread.h>

float prev_time;

struct OmniState
{
   hduVector3Dd position;  //3x1 vector of position
   hduVector3Dd velocity;  //3x1 vector of velocity
   hduVector3Dd inp_vel1;  //3x1 history of velocity used for filtering velocity estimate
   hduVector3Dd inp_vel2;
   hduVector3Dd inp_vel3;
   hduVector3Dd out_vel1;
   hduVector3Dd out_vel2;
   hduVector3Dd out_vel3;
   hduVector3Dd pos_hist1; //3x1 history of position used for 2nd order backward
difference estimate of velocity
   hduVector3Dd pos_hist2;
   hduVector3Dd rot;
   hduVector3Dd joints;
   hduVector3Dd force;     //3 element double vector force[0], force[1], force[2]
   float thetas[7];
   int buttons[2];
   int buttons_prev[2];
   bool lock;
   hduVector3Dd lock_pos;
};


class PhantomROS {

   public:
   ros::NodeHandle n;
   ros::Publisher pose_publisher;
//    ros::Publisher omni_pose_publisher;

   ros::Publisher button_publisher;
   ros::Subscriber haptic_sub;
   ros::Subscriber control_sub;
   std::string omni_name;
   std::string sensable_frame_name;
   std::string link_names[7];

   OmniState *state;
   tf::TransformBroadcaster br;

   void init(OmniState *s)
```

```
    {
        ros::param::param(std::string("~omni_name"), omni_name, std::string("omni1"));

        //Publish on NAME_pose
        std::ostringstream stream00;
        stream00 << omni_name << "_pose";
        std::string pose_topic_name = std::string(stream00.str());
        pose_publisher =
n.advertise<geometry_msgs::PoseStamped>(pose_topic_name.c_str(), 100);
//        omni_pose_publisher =
n.advertise<geometry_msgs::PoseStamped>("omni_pose_internal", 100);

        //Publish button state on NAME_button
        std::ostringstream stream0;
        stream0 << omni_name << "_button";
        std::string button_topic = std::string(stream0.str());
        button_publisher =
n.advertise<phantom_omni::PhantomButtonEvent>(button_topic.c_str(), 100);

        //Subscribe to NAME_force_feedback
        std::ostringstream stream01;
        stream01 << omni_name << "_force_feedback";
        std::string force_feedback_topic = std::string(stream01.str());
        haptic_sub = n.subscribe(force_feedback_topic.c_str(), 100,
&PhantomROS::force_callback, this);


        control_sub = n.subscribe("unilateral_pose_control", 100,
&PhantomROS::pose_control, this);


        //Frame of force feedback (NAME_sensable)
        std::ostringstream stream2;
        stream2 << omni_name << "_sensable";
        sensable_frame_name = std::string(stream2.str());

        for (int i = 0; i < 7; i++)
        {
            std::ostringstream stream1;
            stream1 << omni_name << "_link" << i;
            link_names[i] = std::string(stream1.str());
        }


        state = s;
        state->buttons[0] = 0;
```

```
    state->buttons[1] = 0;
    state->buttons_prev[0] = 0;
    state->buttons_prev[1] = 0;
    hduVector3Dd zeros(0, 0, 0);
    state->velocity = zeros;
    state->inp_vel1 = zeros;  //3x1 history of velocity
    state->inp_vel2 = zeros;  //3x1 history of velocity
    state->inp_vel3 = zeros;  //3x1 history of velocity
    state->out_vel1 = zeros;  //3x1 history of velocity
    state->out_vel2 = zeros;  //3x1 history of velocity
    state->out_vel3 = zeros;  //3x1 history of velocity
    state->pos_hist1 = zeros; //3x1 history of position
    state->pos_hist2 = zeros; //3x1 history of position
    state->lock = false;  // Jose: I have changed this value. Before was "true"
    state->lock_pos = zeros;
}
```

```
/*******************************************************************************
*********
   ROS node callback.

*******************************************************************************
*********/
 //    void force_callback(const geometry_msgs::WrenchConstPtr& wrench)
  void force_callback(const phantom_omni::OmniFeedbackConstPtr& omnifeed)
  {
    ////////////////////Some people might not like this extra damping, but it
    ////////////////////helps to stabilize the overall force feedback. It isn't
    ////////////////////like we are getting direct impedance matching from the
    ////////////////////omni anyway
    state->force[0] = omnifeed->force.x - 0.001*state->velocity[0];
    state->force[1] = omnifeed->force.y - 0.001*state->velocity[1];
    state->force[2] = omnifeed->force.z - 0.001*state->velocity[2];

    state->lock_pos[0] = omnifeed->position.x;
    state->lock_pos[1] = omnifeed->position.y;
    state->lock_pos[2] = omnifeed->position.z;
  //        state->force[2] = wrench->force.z;
  }
```

```
    void pose_control(geometry_msgs::Vector3 force_control)
    {
        state->force[0] = force_control.x;
        state->force[1] = force_control.y;
        state->force[2] = force_control.z;
    }



    void publish_omni_state()
    {
        //Construct transforms
        tf::Transform l0, sensable, l1, l2, l3, l4, l5, l6, l0_6;
        l0.setOrigin(tf::Vector3(0., 0, 0.15));
        l0.setRotation(tf::createQuaternionFromRPY(0, 0, 0));
        br.sendTransform(tf::StampedTransform(l0, ros::Time::now(), omni_name.c_str(),
link_names[0].c_str()));

        sensable.setOrigin(tf::Vector3(0., 0, 0));
        //sensable.setRotation(tf::Quaternion(-M_PI/2, 0, M_PI/2));
        sensable.setRotation(tf::createQuaternionFromRPY(-M_PI/2, 0, M_PI/2));
        br.sendTransform(tf::StampedTransform(sensable, ros::Time::now(),
omni_name.c_str(), sensable_frame_name.c_str()));

        l1.setOrigin(tf::Vector3(0., 0, 0.));
        //l1.setRotation(tf::Quaternion(-state->thetas[1], 0, 0));
        l1.setRotation(tf::createQuaternionFromRPY(0, 0, -state->thetas[1]));

        l2.setOrigin(tf::Vector3(0., 0, 0.));
        //l2.setRotation(tf::Quaternion(0, state->thetas[2], 0));
        l2.setRotation(tf::createQuaternionFromRPY(0, state->thetas[2], 0));

        l3.setOrigin(tf::Vector3(-.131, 0, 0.));
        //l3.setRotation(tf::Quaternion(0, state->thetas[3], 0));
        l3.setRotation(tf::createQuaternionFromRPY(0, state->thetas[3], 0));

        l4.setOrigin(tf::Vector3(0., 0, -.137));
        //l4.setRotation(tf::Quaternion(state->thetas[4]+M_PI, 0, 0));
        l4.setRotation(tf::createQuaternionFromRPY(0, 0, state->thetas[4]+ M_PI +
0.34906585)); // 0.4014257

        l5.setOrigin(tf::Vector3(0., 0., 0.));
        //l5.setRotation(tf::Quaternion(0., -state->thetas[5]+M_PI,0));
        l5.setRotation(tf::createQuaternionFromRPY(0., -state->thetas[5]+M_PI +
1.06465084 ,0));
```

```
        l6.setOrigin(tf::Vector3(0., 0., 0.));
        l6.setRotation(tf::createQuaternionFromRPY(0., 0, state->thetas[6]+M_PI));

        l0_6 = l0 * l1 * l2 * l3 * l4 * l5 * l6;
        br.sendTransform(tf::StampedTransform(l0_6, ros::Time::now(),
link_names[0].c_str(), link_names[6].c_str()));
        //Don't send these as they slow down haptics thread

        // ----- tf::StampedTransform(tf::Pose &T, ros::Time &t, std::string parent,
std::string child) ----- The pose here is pose of child in the frame of parent

//        br.sendTransform(tf::StampedTransform(l1, ros::Time::now(),
link_names[0].c_str(), link_names[1].c_str()));
//        br.sendTransform(tf::StampedTransform(l2, ros::Time::now(),
link_names[1].c_str(), link_names[2].c_str()));
//        br.sendTransform(tf::StampedTransform(l3, ros::Time::now(),
link_names[2].c_str(), link_names[3].c_str()));
//        br.sendTransform(tf::StampedTransform(l4, ros::Time::now(),
link_names[3].c_str(), link_names[4].c_str()));
//        br.sendTransform(tf::StampedTransform(l5, ros::Time::now(),
link_names[4].c_str(), link_names[5].c_str()));
//        br.sendTransform(tf::StampedTransform(l6, ros::Time::now(),
link_names[5].c_str(), link_names[6].c_str()));

        //Sample 'end effector' pose
        geometry_msgs::PoseStamped pose_stamped;
        pose_stamped.header.frame_id = link_names[6].c_str();
        pose_stamped.header.stamp = ros::Time::now();
        pose_stamped.pose.position.x = 0.0;   //was 0.03 to end of phantom
        pose_stamped.pose.orientation.w = 1.;
        pose_publisher.publish(pose_stamped);

//        geometry_msgs::PoseStamped omni_internal_pose;
//        omni_internal_pose.header.frame_id = omni_name.c_str();
//        omni_internal_pose.header.stamp = ros::Time::now();
//        omni_internal_pose.pose.position.x = state->position[0]/1000.0;
//        omni_internal_pose.pose.position.y = state->position[1]/1000.0;
//        omni_internal_pose.pose.position.z = state->position[2]/1000.0;
//        omni_pose_publisher.publish(omni_internal_pose);

        if ((state->buttons[0] != state->buttons_prev[0]) or (state->buttons[1] != state-
>buttons_prev[1]))
        {

          if ((state->buttons[0] == state->buttons[1]) and (state->buttons[0] == 1))
          {
```

```
            state->lock = !(state->lock);
        }
        phantom_omni::PhantomButtonEvent button_event;
        button_event.grey_button = state->buttons[0];
        button_event.white_button = state->buttons[1];
        state->buttons_prev[0] = state->buttons[0];
        state->buttons_prev[1] = state->buttons[1];
        button_publisher.publish(button_event);
      }
    }
};

HDCallbackCode HDCALLBACK omni_state_callback(void *pUserData)
{
    OmniState *omni_state = static_cast<OmniState *>(pUserData);

    /*  Begin grabs state information from the device.
        End sends information such as forces to the device.
        All hdSet and state calls should be done within a begin/end frame. */

    hdBeginFrame(hdGetCurrentDevice()); // Returns the ID of the current device.
    //Get angles, set forces
    hdGetDoublev(HD_CURRENT_GIMBAL_ANGLES, omni_state->rot);
    hdGetDoublev(HD_CURRENT_POSITION,      omni_state->position);
    hdGetDoublev(HD_CURRENT_JOINT_ANGLES,  omni_state->joints);

        hduVector3Dd vel_buff(0, 0, 0);
    vel_buff = (omni_state->position*3 - 4*omni_state->pos_hist1 + omni_state-
>pos_hist2)/0.002;  //mm/s, 2nd order backward dif
        //    omni_state->velocity = 0.0985*(vel_buff+omni_state-
>inp_vel3)+0.2956*(omni_state->inp_vel1+omni_state->inp_vel2)-(-
0.5772*omni_state->out_vel1+0.4218*omni_state->out_vel2 - 0.0563*omni_state-
>out_vel3);    //cutoff freq of 200 Hz
         omni_state->velocity = (.2196*(vel_buff+omni_state-
>inp_vel3)+.6588*(omni_state->inp_vel1+omni_state->inp_vel2))/1000.0-(-
2.7488*omni_state->out_vel1+2.5282*omni_state->out_vel2 - 0.7776*omni_state-
>out_vel3);  //cutoff freq of 20 Hz

    omni_state->pos_hist2 = omni_state->pos_hist1;
    omni_state->pos_hist1 = omni_state->position;
    omni_state->inp_vel3  = omni_state->inp_vel2;
    omni_state->inp_vel2  = omni_state->inp_vel1;
    omni_state->inp_vel1  = vel_buff;
    omni_state->out_vel3  = omni_state->out_vel2;
    omni_state->out_vel2  = omni_state->out_vel1;
    omni_state->out_vel1  = omni_state->velocity;
```

```
    //    printf("position x, y, z: %f %f %f \n", omni_state->position[0], omni_state->position[1], omni_state->position[2]);
    //    printf("velocity x, y, z, time: %f %f %f \n", omni_state->velocity[0], omni_state->velocity[1],omni_state->velocity[2]);
    if (omni_state->lock == true)
    {
        omni_state->force[0] = 0;
        omni_state->force[1] = 0;
        omni_state->force[2] = 0;


//      omni_state->force = 0.05*(omni_state->lock_pos-omni_state->position) - 0.001*omni_state->velocity;        // Jose: I have commented this line

//*        printf("position x, y, z: %f %f %f \n", omni_state->position[0], omni_state->position[1], omni_state->position[2]);   // TEST RELE
//*                if(omni_state->position[0] > 4.0){
//*                        omni_state->force[0] = -2.0 * (omni_state->position[0]-4.0);

//                }else if (omni_state->position[0] < -4.0){
//                        omni_state->force[0] = 0.6;
//*                }

    }

    hdSetDoublev(HD_CURRENT_FORCE,        omni_state->force);

    //Get buttons
    int nButtons = 0;
    hdGetIntegerv(HD_CURRENT_BUTTONS, &nButtons);
    omni_state->buttons[0] = (nButtons & HD_DEVICE_BUTTON_1) ? 1 : 0;
    omni_state->buttons[1] = (nButtons & HD_DEVICE_BUTTON_2) ? 1 : 0;

    hdEndFrame(hdGetCurrentDevice());

    HDErrorInfo error;
    if (HD_DEVICE_ERROR(error = hdGetError()))
    {
        hduPrintError(stderr, &error, "Error during main scheduler callback\n");
        if (hduIsSchedulerError(&error))
            return HD_CALLBACK_DONE;
    }

    float t[7] = {0., omni_state->joints[0], omni_state->joints[1], omni_state->joints[2]-
```

```
omni_state->joints[1],
                  omni_state->rot[0],    omni_state->rot[1],    omni_state->rot[2]};
   for (int i = 0; i < 7; i++)
      omni_state->thetas[i] = t[i];
   return HD_CALLBACK_CONTINUE;
}

/
*****************************************************************************
*********
Automatic Calibration of Phantom Device - No character inputs
*****************************************************************************
*********/
void HHD_Auto_Calibration()
{
  int calibrationStyle;
  int supportedCalibrationStyles;
  HDErrorInfo error;

  hdGetIntegerv(HD_CALIBRATION_STYLE, &supportedCalibrationStyles);
  if (supportedCalibrationStyles & HD_CALIBRATION_ENCODER_RESET)
  {
      calibrationStyle = HD_CALIBRATION_ENCODER_RESET;
      ROS_INFO("HD_CALIBRATION_ENCODER_RESE..\n\n");
  }
  if (supportedCalibrationStyles & HD_CALIBRATION_INKWELL)
  {
      calibrationStyle = HD_CALIBRATION_INKWELL;
      ROS_INFO("HD_CALIBRATION_INKWELL..\n\n");
  }
  if (supportedCalibrationStyles & HD_CALIBRATION_AUTO)
  {
     calibrationStyle = HD_CALIBRATION_AUTO;
     ROS_INFO("HD_CALIBRATION_AUTO..\n\n");
  }

  do
  {
     hdUpdateCalibration(calibrationStyle);
     ROS_INFO("Calibrating.. (put stylus in well)\n");
     if (HD_DEVICE_ERROR(error = hdGetError()))
     {
        hduPrintError(stderr, &error, "Reset encoders reset failed.");
        break;
        }
  }   while (hdCheckCalibration() != HD_CALIBRATION_OK);
```

```
  ROS_INFO("\n\nCalibration complete.\n");
}

void *ros_publish(void *ptr)
{
  PhantomROS *omni_ros = (PhantomROS *) ptr;
  int publish_rate;
  omni_ros->n.param(std::string("publish_rate"), publish_rate, 100);
  ros::Rate loop_rate(publish_rate);
  ros::AsyncSpinner spinner(2);
  spinner.start();

  while(ros::ok())
  {
    omni_ros->publish_omni_state();
    loop_rate.sleep();
  }
  return NULL;
}

int main(int argc, char** argv)
{
  //////////////////////////////////////////////////////////////
  // Init Phantom
  //////////////////////////////////////////////////////////////
  HDErrorInfo error;
  HHD hHD;
  hHD = hdInitDevice(HD_DEFAULT_DEVICE);
  if (HD_DEVICE_ERROR(error = hdGetError()))
  {
    hduPrintError(stderr, &error, "Failed to initialize haptic device");
    ROS_ERROR("Failed to initialize haptic device"); //: %s", &error);
    return -1;
  }

  ROS_INFO("Found %s.\n\n", hdGetString(HD_DEVICE_MODEL_TYPE));
  hdEnable(HD_FORCE_OUTPUT);
  hdStartScheduler();
  if (HD_DEVICE_ERROR(error = hdGetError()))
  {
    ROS_ERROR("Failed to start the scheduler");//, &error);
    return -1;
  }
  HHD_Auto_Calibration();
```

```
  ////////////////////////////////////////////////////////////////
  // Init ROS
  ////////////////////////////////////////////////////////////////
  ros::init(argc, argv, "omni_haptic_node");
  OmniState state;
  PhantomROS omni_ros;

  omni_ros.init(&state);
  hdScheduleAsynchronous(omni_state_callback, &state,
HD_MAX_SCHEDULER_PRIORITY);

  ////////////////////////////////////////////////////////////////
  // Loop and publish
  ////////////////////////////////////////////////////////////////
  pthread_t publish_thread;
  pthread_create(&publish_thread, NULL, ros_publish, (void*) &omni_ros);
  pthread_join(publish_thread, NULL);

  ROS_INFO("Ending Session....\n");
  hdStopScheduler();
  hdDisableDevice(hHD);

  return 0;
}
```