

Sistema de monitorización inercial del movimiento de las extremidades superiores

Daniel Fernández Villanueva

22 de julio de 2013

Índice general

I Memoria	6
1. ANTECEDENTES	7
2. OBJETIVO DEL PROYECTO	9
3. ESPECIFICACIONES DE DISEÑO	12
3.1. Especificaciones del hardware	12
3.2. Especificaciones del software	12
4. ESTUDIO DE SOLUCIONES	14
4.1. Obtención del estado del brazo	14
4.1.1. Encoders	14
4.1.2. Sensores ópticos	14
4.1.3. Sensores inerciales	15
4.1.4. Solución escogida: red de sensores inerciales xsens MTx	15
4.2. Comunicación entre los sensores y el PC	15
4.2.1. Driver para la comunicación xbus Master - PC	16
4.3. Desarrollo, comunicación entre programas y distribución en varias unidades de procesamiento	16
4.3.1. La plataforma de desarrollo de software para robots ROS	17
4.3.2. Herramientas proporcionadas por ROS	17
4.3.3. ROS como sistema distribuido	21
4.4. Sistema operativo	22
4.4.1. El sistema operativo Ubuntu	22
5. CÁLCULO DE LAS POSICIONES Y ORIENTACIONES DE LOS SENsoRES	23
5.1. Formas de representar orientaciones espaciales	23
5.1.1. Ángulos de Euler	24
5.1.2. Matriz de rotación	24
5.1.3. Cuaternión	26
5.1.4. Solución elegida: cuaterniones	27
5.2. Utilización de cuaterniones para la representación de rotaciones de un sólido rígido	27
5.2.1. Rotación de un vector alrededor de un eje y un ángulo dados	29
5.2.2. Composición de rotaciones en coordenadas extrínsecas	31
5.2.3. Composición de rotaciones en coordenadas intrínsecas	32
5.2.4. Relación entre rotaciones intrínsecas y extrínsecas	32
5.2.5. Orientación relativa entre dos sólidos	35

5.3. Cálculo de la posición del brazo	36
5.4. Obtención de los ángulos de Euler a partir del cuaternión de orientación	37
5.4.1. Ampliación del intervalo de los ángulos obtenidos	40
Generación de una segunda solución a partir de la primera	40
Giro de los ejes locales	41
6. IMPLEMENTACIÓN DEL SOFTWARE	43
6.1. Librería para manejo de cuaterniones, vectores y matrices	44
6.1.1. La clase dfv::Quaternion	44
6.1.2. La clase dfv::Vector3	46
6.1.3. La clase dfv::Matrix	48
6.1.4. Cómo usar la librería dfv en otro paquete de ROS	49
6.2. Driver para la comunicación xbus Master/xsens - PC	50
6.2.1. Funcionamiento de la clase xsens::Driver	52
6.2.2. Programa principal: xsens_node	55
6.2.3. Cómo tomar datos del sensor desde otro programa	56
6.3. Controlador del brazo robótico del robot Youbot real	57
6.3.1. El driver youbot_oodl	58
6.3.2. La clase Youbot	58
6.4. Visualizador de la posición del brazo	60
6.4.1. La clase gazebo::CModelList	60
6.5. Controlador de un simulador del brazo robótico del robot Youbot	63
7. PRUEBAS Y RESULTADOS OBTENIDOS	64
7.1. Captura de datos de un sensor Xsens MTi-G	64
7.2. Captura de datos de una red de sensores Xsens MTx conectados a un máster Xbus	66
7.3. Utilización de los datos obtenidos para el control del brazo robótico de un robot Youbot	66
7.4. Distribución del sistema anterior en varias unidades de procesamiento	67
7.5. Conclusiones	68
8. BIBLIOGRAFÍA	69
II Anexos	70
A. INSTALACIÓN DEL SOFTWARE	71
A.1. Instalación de ROS Fuerte	71
A.1.1. Configuración de los repositorios de Ubuntu	71
A.1.2. Configuración del archivo sources.list	71
A.1.3. Configuración de la keys	72
A.1.4. Descarga e instalación	72
A.1.5. Configuración del entorno	73
A.2. Instalación del simulador Gazebo	73
A.3. Instalación del simulador de brazo robótico del YouBot	73
A.4. Instalación del stack youbot-xsens-controller	74

B. PUESTA EN MARCHA DEL SOFTWARE	75
B.1. Iniciación del driver del sensor Xsens	75
B.2. Iniciación del driver del robot Youbot	75
B.3. Ejecución del programa youbot_controller	75
B.4. Ejecución del simulador del brazo robótico del robot Youbot	76
B.5. Ejecución de la visualización del modelo del brazo humano	76
C. SOLUCIÓN DE PROBLEMAS	79
C.1. Error al iniciar Gazebo	79
C.2. Al ejecutar el simulador del brazo robótico no se abre la interfaz gráfica de Gazebo	80
C.3. El driver del Xsens no contacta con los sensores	80
III Presupuesto	81

Índice de figuras

1.1.	Sistema de monitorización de la cinemática de la espina dorsal y brazos	7
2.1.	Esquema de un brazo humano	10
2.2.	El robot YouBot	10
4.1.	Uso de encoders en un exoesqueleto para la monitorización del brazo.	14
4.2.	El sensor Kinect	14
4.3.	IMU Xsens MTi-G	15
4.4.	Red de sensores xsens conectados a un máster xbus	16
4.5.	Logotipo de ROS	17
4.6.	Ejemplo de una red de nodos de ROS	19
4.7.	Comando rostopic mostrando datos de un sensor	20
4.8.	Programa rxplot mostrando datos de un sensor	20
4.9.	El simulador Gazebo	21
4.10.	Programa Gazebo con la simulación del brazo del Youbot	21
4.11.	El sistema operativo Ubuntu	22
5.1.	Rotación de un vector mediante la operación $q_1 p q_1^*$	30
5.2.	Orientación relativa entre dos sólidos rígidos	35
5.3.	Posiciones del brazo inicial y genérica	36
5.4.	Sistema de coordenadas local sobre la esfera	38
5.5.	Tercera rotación en el sistema de coordenadas local sobre el punto P de la esfera	38
5.6.	Dos soluciones para θ (<i>pitch</i>) y ψ (<i>yaw</i>)	41
5.7.	Giro de los ejes locales para evitar el gimbal lock	42
6.1.	Esquema simplificado de comunicaciones	44
6.2.	Articulaciones del brazo robótico del YouBot y su identificación	58
6.3.	Programa Gazebo con un modelo de un brazo humano	60
7.1.	Representación gráfica de los datos de los acelerómetros	65
A.1.	Ventana Software Sources	72
B.1.	Ventana de Gazebo abierta con el archivo empty.world.launch	77
B.2.	Eliminación del plano en la simulación de Gazebo	77
B.3.	Puesta en pausa de la simulación de Gazebo	77
B.4.	Visualización del brazo en Gazebo	78

Índice de cuadros

5.1.	Propiedades de los cuaterniones	26
5.2.	Algoritmo de cálculo de los ángulos de Euler a partir del cuaternion de orientación.	40
6.1.	Unidades de los datos calibrados devueltos por los sensores	54
6.2.	Topics publicados por el programa xsens_node	55
6.3.	Parámetros publicados por el programa xsens_node	56
6.4.	Topics a los que está suscrito el driver youbot_oodl	59
7.1.	Resultados de la prueba para obtener la frecuencia de lectura de un sensor Xsens MTi-G	65
7.2.	Resultados de la prueba para obtener la frecuencia de lectura de una red de 3 sensores MTx conectados a un máster Xbus	66
7.3.	Valor de las variables de entorno para el sistema distribuido	67

Parte I

Memoria

Capítulo 1

ANTECEDENTES

Los sistemas de monitorización del cuerpo humano llevan existiendo desde la década de los 70 del siglo XX. En esta época se empezaron a usar técnicas fotogramétricas en investigación biomecánica, que consistían en la medición sobre fotografías para determinar las propiedades geométricas y situaciones espaciales de los distintos objetos y partes del cuerpo.

A medida que mejoraba la tecnología electrónica, se fue incorporando el campo de la captura de movimiento a otras especialidades, como el entrenamiento en ciertos deportes o la educación. Más recientemente, con el inmenso avance en el procesamiento gráfico, se ha comenzado a utilizar en las animaciones por ordenador, videojuegos, películas, realidad virtual, y muchos otros sectores relacionados con la industria audiovisual y el entretenimiento.

Existen además algunas aplicaciones médicas de los sistemas de monitorización del movimiento. Uno de los ejemplos más destacados es el llamado *análisis de la marcha*, en el que se registran los movimientos y posiciones del cuerpo humano. El análisis de estos movimientos permite realizar estudios detallados de personas que presentan patologías como parálisis cerebral, enfermedad de Parkinson y trastornos neuromusculares, y permiten un diagnóstico más detallado, un mejor diseño del plan de tratamiento y un control de seguimiento más sofisticado.

Por último, cabe citar el uso de la monitorización del movimiento para teleoperación de sistemas actuadores. En este caso se estaría hablando de interfaces hombre-máquina basados tanto en el movimiento del propio cuerpo humano, como en el movimiento por parte de la persona de un modelo de la máquina. Posibles aplicaciones varían desde el

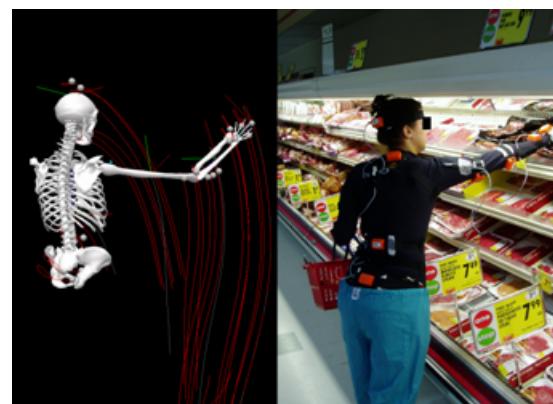


Figura 1.1: Sistema de monitorización de la cinemática de la espina dorsal y de los brazos. Crédito: Escuela de Medicina Alpert, *Brown University*, EEUU.

control de actividades forestales, hasta operaciones en entornos nucleares y de exploración espacial. En todos estos casos es posible además incorporar un *feedback* a modo de fuerzas que se aplican al modelo, lo que mejora la percepción de la persona del medio en que está situado la máquina actuadora.¹

Este proyecto se centrará en la implementación de un sistema de monitorización del movimiento, principalmente del brazo humano, aunque se utilizará una arquitectura que hará posible su uso en otros entornos. Además se creará una serie de aplicaciones que hagan uso de este sistema de monitorización, y se comprobará la eficacia de éste en ellas.

¹Wen-Hong Zhu, S. E. Salcudean. *Teleoperation with Adaptative Motion/Force Control*. Dept. of Electrical and Computer Engineering, University of British Columbia, Canada.

Capítulo 2

OBJETIVO DEL PROYECTO

El presente proyecto tiene como objetivo la implementación de un prototipo de un sistema de monitorización en tiempo real de las extremidades superiores del cuerpo humano. Entiéndase por sistema de monitorización un conjunto de elementos, tanto de hardware como de software, que permiten la determinación en tiempo real del estado de un brazo humano. El estado de un brazo puede definirse como cierto conjunto de magnitudes de las que se pueden derivar otras magnitudes que pueden resultar de mayor utilidad para la aplicación en la que se desee utilizar el sistema de monitorización.

El estado de un brazo humano puede ser determinado mediante tres longitudes y tres orientaciones, que se corresponden con los tres eslabones principales del brazo humano –brazo, antebrazo y mano–. Estos eslabones están conectados mediante las articulaciones hombro, codo y muñeca, cada una con múltiples grados de libertad. Dado que las longitudes de los eslabones permanecen constantes a lo largo del tiempo, y el estado de rotación de una articulación puede ser determinado fácilmente si se conocen los estados de rotación de cada eslabón con referencia a un sistema fijo, el objetivo principal del presente proyecto será determinar dicho estado de rotación respecto al sistema fijo de las tres articulaciones principales del brazo. Se verá cómo únicamente con las tres mencionadas orientaciones es posible calcular el resto de magnitudes deseadas¹.

De forma inherente al objeto de este proyecto, aparecen una serie de problemas a los que es necesario proporcionar una solución:

1. Toma de datos de los sensores:

Se tendrá que implementar un sistema que permita obtener los datos que proporcionan los sensores utilizados para registrar las orientaciones. Este sistema deberá de ser capaz de realizar la lectura en tiempo real, con mínima latencia y a una frecuencia aceptable. Los elementos que componen el sistema serán los sensores, que medirán el estado real del brazo, y un programa informático llamado *driver* que actuará de interfaz entre los sensores y el PC.

2. Tratamiento de los datos: Una vez se tenga disponible un sistema para leer los datos de los sensores, el siguiente paso será procesar dichos datos para obtener las magnitudes necesarias para las distintas aplicaciones (Por ejemplo: orientación relativa entre sensores, ángulos de Euler, etc.). Este procesamiento lo llevarán a cabo

¹Véase el capítulo CÁLCULO DE LAS POSICIONES Y ORIENTACIONES DE LOS SENSORES

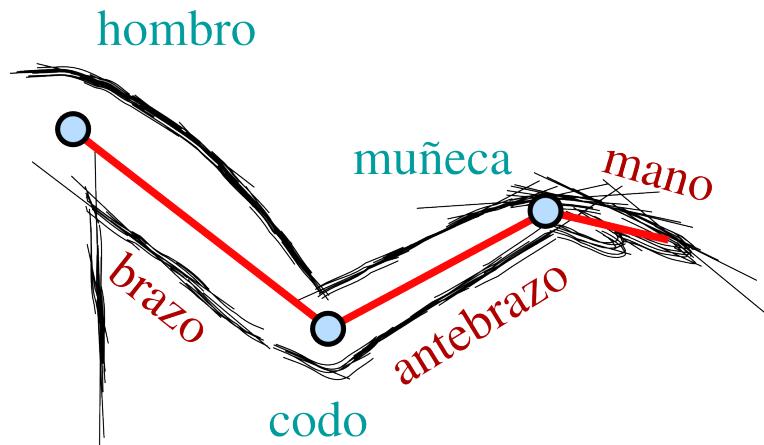


Figura 2.1: Esquema de un brazo humano. Sabiendo de antemano las longitudes de brazo, antebrazo y mano, y las orientaciones de éstos con respecto a un sistema fijo en cada momento, es posible determinar las posiciones, velocidades, aceleraciones, etc. de cada punto del brazo, además de los estados de las articulaciones.

uno o varios programas informáticos que tienen que ser capaces de comunicarse con el *driver*. Dado que se tratará de dotar al sistema de la máxima modularidad posible, estos programas serán independientes del *driver*.

3. Aplicaciones del sistema de monitorización:

En esta última fase se crearán los sistemas necesarios para la utilización de los datos con el objetivo deseado. En este proyecto se crearán en concreto tres aplicaciones en las que se usará el sistema de monitorización:

- a) Visualización de un esquema del brazo humano en un visualizador 3D.
- b) Control de un simulador de un brazo robótico.
- c) Control del movimiento de un robot real (robot YouBot).



Figura 2.2: El robot YouBot.

En este documento se realizará una descripción detallada de las soluciones que se han

adoptado para cada problema, y como se han incorporado dentro del sistema total de monitorización, de forma que cada parte se comunique con las demás de forma eficaz.

Capítulo 3

ESPECIFICACIONES DE DISEÑO

En esta apartado se detallan las características esenciales que se pretenden implementar en el sistema de monitorización.

3.1. Especificaciones del hardware

Las siguientes especificaciones se refieren a las características que ha de reunir el conjunto de sensores con los que se realizarán las medidas.

- **Flexibilidad:** La red de sensores no sólo debe ser capaz de registrar la posición de un brazo humano, sino que además debe de ser aplicable a otro tipo de sistemas articulados, tanto humanos como de máquinas y robots. Por ello dicho sistema ha de ser flexible en cuanto a número de sensores y grados de libertad que se puedan monitorizar.
- **Pequeño y ligero:** El sistema no puede ser demasiado aparatoso, ya que en se pretende colocarlo en extremidades humanas. Además no debería de dificultar la movilidad de la persona en absoluto.
- **Fiabilidad.** Que sea lo más fidedigno posible en las medidas que proporciona. Esto puede tener mucha importancia en aplicaciones donde se requiera precisión, como por ejemplo el control del brazo robótico que se pretende implementar.
- **Económico.** Por razones obvias, se buscará un sistema cuyo coste de creación e implementación sea pequeño.
- **Versatilidad.** El sistema debería dejar la puerta abierta a poder ser incorporado en multitud de aplicaciones de forma sencilla, además de poder ser ampliado con mayor número de sensores o de distinto tipo sin mucho esfuerzo y sin tener que cambiar otras partes del sistema. Por ejemplo, si en un futuro se pretenden incorporar más sensores inerciales, o incluso sensores de distinto tipo, se pretenderá que sólo sea necesario modificar el programa *driver* para la comunicación con los nuevos sensores, y el resto del sistema pueda funcionar como antes.

3.2. Especificaciones del software

En la siguiente lista se muestran las características que se buscarán tanto en el software que se utilice para el desarrollo de los programas para este proyecto, como en los propios

programas en sí.

- **Modularidad:** Se ha mencionado en las especificaciones del hardware que se pretende crear un sistema en el que se puedan incorporar en un futuro otros sensores. De la misma forma, también se busca un sistema que permita que se puedan añadir nuevos programas que utilicen los datos obtenidos de los sensores, pero que realicen otras funciones distintas a los programas originales, sin tener que modificar en absoluto estos últimos. Ésto dejará la puerta abierta por ejemplo a incluir todo tipo de visualizaciones, u otro tipo de actuadores que no sean el robot YouBot.
- **Sistema distribuido:** Se pretende implementar un sistema que pueda ser ejecutado tanto en uno como en varios ordenadores de forma paralela, cada uno encargándose de realizar las tareas de cierta parte del sistema.
- **Interactividad con el usuario:** Dado que el sistema de monitorización tiene carácter de prototipo, no se buscará de momento la creación de una interfaz gráfica a modo de producto final. A lo que se dará especial importancia es a la fácil comunicación entre programas y a la posibilidad de crear de forma sencilla y rápida otros programas que sí permitan realizar visualizaciones en programas ya existentes. Aún así, esta arquitectura deja la puerta abierta a la creación posterior de un programa final que incorpore todos los componentes del sistema y permita interactuar con él mediante una interfaz gráfica de usuario.
- **Economía:** El desarrollo del software ha de ser lo más barato posible, usando en la medida que se pueda software libre y de código abierto.

Capítulo 4

ESTUDIO DE SOLUCIONES

4.1. Obtención del estado del brazo

Para obtener el estado del brazo, será necesario un conjunto de sensores que sean capaces de registrar las orientaciones de brazo, antebrazo y mano. A continuación se presentarán algunas opciones disponibles, junto a sus ventajas y desventajas.

4.1.1. Encoders

El uso de *encoders* suele ser una opción interesante en el caso de que se quisiera detectar el ángulo de una articulación de un grado de libertad, por lo que son el tipo de sensores usados en muchos robots para saber el estado de sus articulaciones. Sin embargo, para el caso de un brazo humano no resultan prácticos; su colocación en las articulaciones del brazo es aparatoso –se tendría que hacer uso de un exoesqueleto en el que colocar los sensores– y serían necesarios varios encoders para registrar los ángulos en articulaciones de varios grados de libertad.

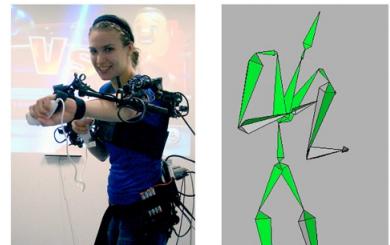


Figura 4.1: Uso de encoders en un exoesqueleto para la monitorización del brazo. Crédito: Mpeg-V

4.1.2. Sensores ópticos



Figura 4.2: El sensor *Kinect*.

Una opción interesante es el uso de sensores ópticos de detección de movimiento –refiriéndose a sensores de tipo *Kinect*–. En estos sensores no es necesario acoplar ningún elemento al cuerpo humano, además de ser bastante económicos y capaces de registrar la posición y orientación de varios elementos a la vez. Sin embargo, la imprecisión de este tipo de sensores no resulta conveniente para el tipo de aplicaciones en las que se pretende aplicar el sistema de monitorización –llegando a tener errores máximos medios del rango de $20^\circ - 40^\circ$ ¹–

¹Choppin, S., Wheat, J., *MARKER-LESS TRACKING OF HUMAN MOVEMENT USING MICROSOFT KINECT*, 2012

4.1.3. Sensores inerciales

Los sensores inerciales –normalmente incluidos dentro de una misma unidad llamada Unidad de Medición Inercial - Inertial Measurement Unit o IMU para abreviar– utilizan un conjunto de acelerómetros, giróscopos y en ocasiones magnetómetros para la obtención de la velocidad, orientación, fuerzas magnéticas entre otros posibles parámetros. Esta opción es realmente interesante ya que muchos IMUs ofrecen la posibilidad de dar como salida la orientación del elemento al que estén adheridos de forma directa. Además son pequeños y sencillos de colocar en el cuerpo humano.

Una de las principales desventajas que presentan los IMUs es la deriva que presentan a lo largo del tiempo. Esto es debido a que para obtener el estado en cierto tiempo, hacen uso de las medidas del estado anterior y los errores presentes se van acumulando. Sin embargo para la orientación es posible minimizar la deriva a un mínimo, incluso eliminarla, mediante un sistema llamado AHRS (*Attitude and Heading Reference System*), en el que se utilizan la medida de la aceleración de la gravedad y el norte magnético para compensar las derivas que se obtienen con los giróscopos.



Figura 4.3: IMU Xsens MTi-G.

4.1.4. Solución escogida: red de sensores inerciales xsens MTx

Los sensores Xsens MTx implementan un filtro de Kalman para la obtención de la orientación, e incorporan la corrección AHRS para el cálculo de ésta, por lo que proporcionan una medida de alta precisión y sin deriva apreciable de la orientación que presenta el sensor. Además es posible utilizar varios de estos sensores a la vez formando una red conectándolos a un máster Xbus, lo que permitirá tomar los datos de los distintos eslabones que forman el brazo al mismo tiempo. En la figura 4.4 se muestra una red de sensores Xsens MTx conectados al máster Xbus.

4.2. Comunicación entre los sensores y el PC

La conexión del máster Xbus al PC se realizará mediante cable USB. Para la toma de datos del sensor será necesario un programa especial llamado *driver* que se encargue de gestionar las comunicaciones a través del puerto serie.



Figura 4.4: Red de sensores xsens conectados a un máster xbus.

4.2.1. Driver para la comunicación xbus Master - PC

Las funcionalidades que se pretenden implementar en el *driver* del Xsens son las siguientes:

- Capacidad de funcionar con varios sensores conectados
- Posibilidad de realizar la configuración de los sensores.
- Lectura a frecuencia aceptable.
- Integración en el sistema ROS. Se verá más adelante la importancia de este punto.

Existen algunos drivers disponibles en Internet preparados para trabajar en ROS con sensores Xsens. Sin embargo, los que se han encontrado o bien sólo sirven para un sólo sensor, o trabajan en versiones bastante desfasadas de ROS y se encuentra disponible poca o nula documentación sobre ellos.

La opción que se ha decidido seguir fue aprovechar que en la documentación de los sensores Xsens se proporciona un conjunto de archivos con código en C++ en los que se implementa el protocolo de comunicación a través del puerto serie con un sensor Xsens o un máster Xbus, y a partir de ellos se construirá el driver para la comunicación con los sensores y la publicación de los datos en ROS. Se realiza una descripción detallada de este programa en la sección *Driver para la comunicación Xbus Master/Xsens - PC* del capítulo *IMPLEMENTACIÓN DEL SOFTWARE*.

4.3. Desarrollo, comunicación entre programas y distribución en varias unidades de procesamiento

Dada la multi-modularidad de la que se pretende dotar al sistema de monitorización, será necesario un sistema en el que distintos programas se puedan comunicar entre sí, tanto dentro del mismo PC como en una red de PCs. Sería posible resolver las comunicaciones partiendo desde cero mediante sockets TCP/IP. Sin embargo es bastante más práctico utilizar algún tipo de plataforma que se haga cargo por ella misma de las comunicaciones.

Hasta hace relativamente poco tiempo, los desarrolladores de software para robots carecían de una plataforma común sobre la que sustentarse en la creación de programas y

la resolución de las comunicaciones, y sobre la que compartir y colaborar entre ellos. Con la aparición de las plataformas *open source* de desarrollo, se ha eliminado la necesidad de crear las aplicaciones y las comunicaciones desde cero. Entre estas plataformas, ha comenzado a destacar una, que en poco tiempo se ha convertido en la plataforma líder para desarrollo de software para robots.

4.3.1. La plataforma de desarrollo de software para robots ROS

ROS (del inglés *Robot Operating System*

- Sistema Operativo Robótico) es una plataforma de desarrollo de software que incluye conjunto de utilidades centradas en ayudar al desarrollador en la creación de programas para el control de robots. Esta herramienta incorpora abstracción del hardware, drivers para dispositivos, librerías, visualizadores, utilidades para el intercambio de mensajes entre programas y administradores de paquetes de software, entre otras muchas cosas. ROS es además software abierto, bajo una licencia BSD, por lo que cualquier persona puede ver su código fuente y modificarlo.



Figura 4.5: Logotipo de ROS.

ROS fue desarrollado originalmente en el año 2007 bajo el nombre *switchyard* por el Laboratorio de Inteligencia Artificial de la Universidad de Stanford. Desde 2008 su desarrollo continúa principalmente en el laboratorio de investigación tecnológica Willow Garage. ROS destaca sobre las demás plataformas en que se centra en construir una comunidad de colaboradores, y desde sus orígenes ha sido diseñado para facilitar compartir el software creado por desarrolladores en todo el mundo. El resultado de esta filosofía es que actualmente hay disponibles miles de paquetes de ROS, que contienen *drivers* para una inmensa cantidad de sensores y robots, librerías, mensajes, y todo tipo herramientas, gratuitos, fáciles de instalar y que es posible modificar si así se desease.

4.3.2. Herramientas proporcionadas por ROS

La plataforma ROS proporciona solución a una gran cantidad de problemas que vienen dados inherentemente al objetivo de este proyecto:

- **Creación y compilación de programas:**

ROS proporciona librerías en los lenguajes C++, python y lisp en las que se implementan las diversas herramientas que pone a disposición, como intercambio de mensajes entre programas, servidor de parámetros, gestión del tiempo, entre muchas otras. En el presente proyecto se utilizará la implementación en C++ –**roscpp**–, que es la librería más ampliamente difundida en la comunidad de ROS, además de estar diseñada para ser la librería más actualizada y con mayor número de herramientas.

Además ROS pone a disposición un gestor de paquetes de software con el fin de facilitar la reutilización de código. Un **paquete** es la forma más básica de organización

del software en ROS. Un paquete puede contener programas –llamados **nodos**–, librerías, o cualquier otra cosa que posea cierta funcionalidad. La herramienta *roscreate-pkg* permite la creación de un paquete de forma automática. También existe otra herramienta que permite realizar la compilación del paquete, llamada *rosmake*.

En un **stack** de ROS se pueden incluir varios paquetes. Los *stacks* tienen por fin hacer más sencillo compartir conjuntos de paquetes entre distintos ordenadores. Todos los paquetes creados en el presente proyecto se incluirán dentro de un mismo *stack*.

- **Comunicación entre programas:** Uno de los puntos más fuertes de ROS son las herramientas que incluye para la comunicación entre programas, las cuales hacen posible crear un sistema totalmente distribuido, incluso entre varios ordenadores, en los que cada programa puede comunicarse con los demás mediante un interfaz sencilla y con mínima latencia. El sistema de intercomunicación de ROS se basa en los siguientes conceptos:
 - **Máster:** El máster de ROS es el programa que se encarga de gestionar las comunicaciones entre los distintos nodos (programas de ROS). Permite que distintos nodos se encuentren entre si, lleva un registro de los *publishers* y *subscribers* de los distintos *topics* y de los *servicios*, además de proporcionar un *servidor de parámetros*.
 - **Topics:** Los *topics* son los canales de comunicación que utilizan los distintos *nodos* para comunicarse entre sí. Esta comunicación es llevada a cabo mediante *mensajes*. Para leer de un *topic* un nodo hace uso de un *subscriber* (subscriptor). Se dice entonces que el nodo está *suscrito* al *topic*. Otro nodo puede estar publicando mensajes en el *topic* mediante un *publisher*. En este caso se diría que el nodo está *publicando* en el *topic*. En la figura 4.6 se muestra el grafo de comunicaciones de un sistema de navegación de un robot. Se puede comprobar la complejidad que puede llegar a tomar una red de nodos. En esta figura, cada flecha representa un *topic*. Cada *topic* tiene un nombre característico, y acepta un tipo de mensajes. Las elipses se corresponden con los nodos, que también poseen un nombre único que los identifica. El inicio de cada flecha representa un *publisher* del nodo del que parte la flecha, y su final se corresponde con un *subscriber* al *topic*, contenido en el nodo donde termina la flecha.
 - **Mensajes:** Un *topic* acepta una determinada clase de mensaje. Los mensajes están compuestos por un conjunto de uno o varios tipos de datos –floats, strings, arrays, etc.–, o de otros mensajes en una estructura anidada. Existen unas definiciones de mensajes estándar, aunque es posible crear nuevas definiciones de estos mensajes. En este proyecto se han tratado de utilizar mensajes estándar en la medida de lo posible.
 - **Servicios:** Los *topics* proporcionan una comunicación entre nodos de tipo publicador-subscriptor muy flexible, pero en ocasiones se requiere un sistema de tipo solicitud-respuesta, en el que un cliente envía un mensaje de solicitud y el servidor contesta con un mensaje de respuesta. Este tipo de comunicación se implementa en los *servicios* de ROS. Los servicios están pensados para utilizarse en la solicitud de realización de ciertas tareas de vez en cuando. Para

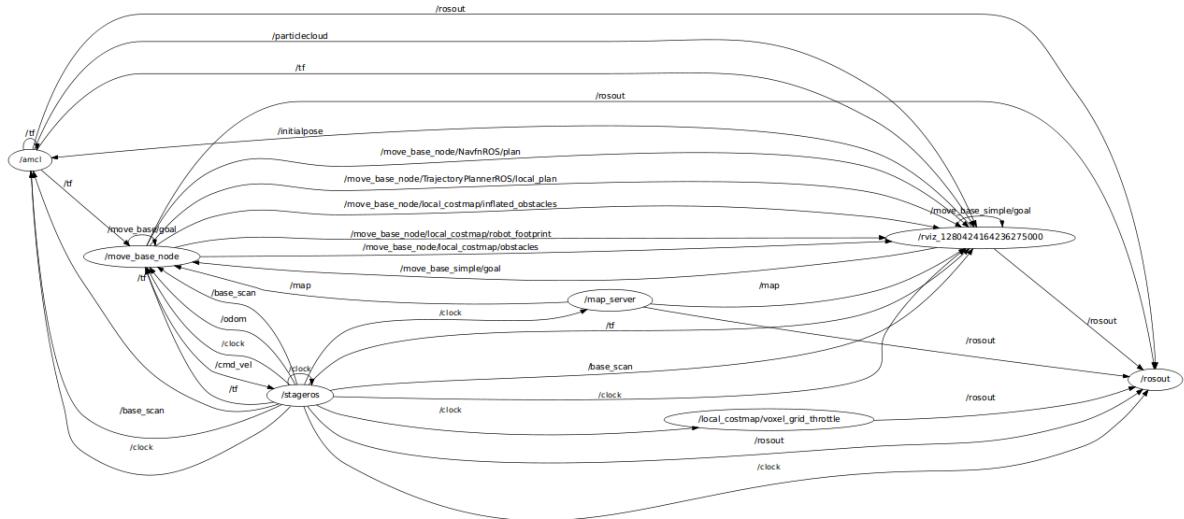


Figura 4.6: Ejemplo de una red de nodos de ROS.

la comunicación de datos en *streaming* –por ejemplo para el envío de los datos de los sensores– es preferible el uso de *topics*.

- **Servidor de parámetros:** El servidor de parámetros de ROS es una especie de diccionario que puede ser accedido por los distintos nodos para crear, leer o modificar los valores de los parámetros almacenados en él. Se usará el servidor de parámetros para almacenar los valores de configuración de los sensores.
 - **Bags:** Los *bags* de ROS son archivos que permiten almacenar los datos de uno o varios mensajes para su posterior reproducción. Permiten, por ejemplo, grabar los datos de los sensores para poder usarlos en otro momento o lugar en los que no estén disponibles los propios sensores.

■ Visualización de datos

- **rostopic**: El comando *rostopic* permite conocer en tiempo real los topics que se encuentran publicados en ROS, y obtener los valores que están siendo publicados en ellos. Para obtener una lista de los topics se ejecutará el siguiente comando:

```
$ rostopic list
```

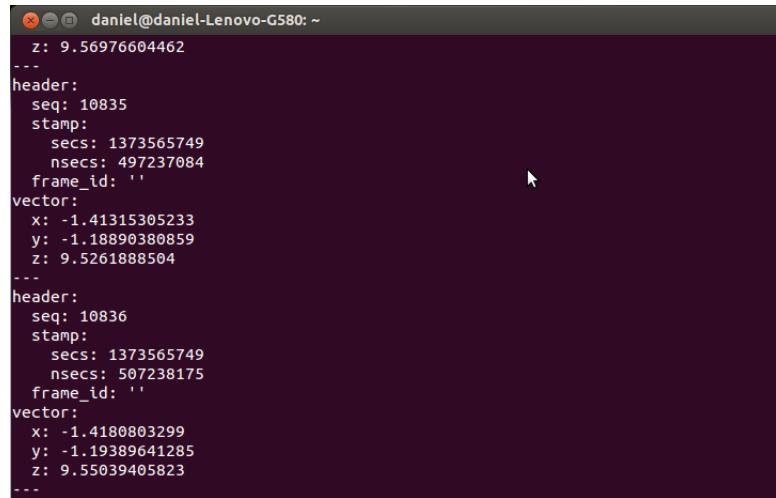
Si se quisiera conocer los valores publicados se puede ejecutar el siguiente comando:

```
$ rostopic echo <nombre_del_mensaje>
```

Por ejemplo, en la figura 4.7 se muestran los datos de los acelerómetros de un sensor xsens conectado al PC. Es el resultado de ejecutar el siguiente comando:

```
$ rostopic echo /xsens_node/sensor0/acc
```

- **rxplot**: La herramienta rxplot permite la representación gráfica de uno o varios valores a la vez, en tiempo real. Un ejemplo se muestra en la figura 4.8, que se ha obtenido con el comando:



```
daniel@daniel-Lenovo-G580: ~
z: 9.56976604462
...
header:
  seq: 10835
  stamp:
    secs: 1373565749
    nsecs: 497237084
  frame_id: ''
vector:
  x: -1.41315305233
  y: -1.18890380859
  z: 9.5261888504
...
header:
  seq: 10836
  stamp:
    secs: 1373565749
    nsecs: 507238175
  frame_id: ''
vector:
  x: -1.4180803299
  y: -1.19389641285
  z: 9.55039405823
...
```

Figura 4.7: Comando rostopic mostrando datos de un sensor en tiempo real.

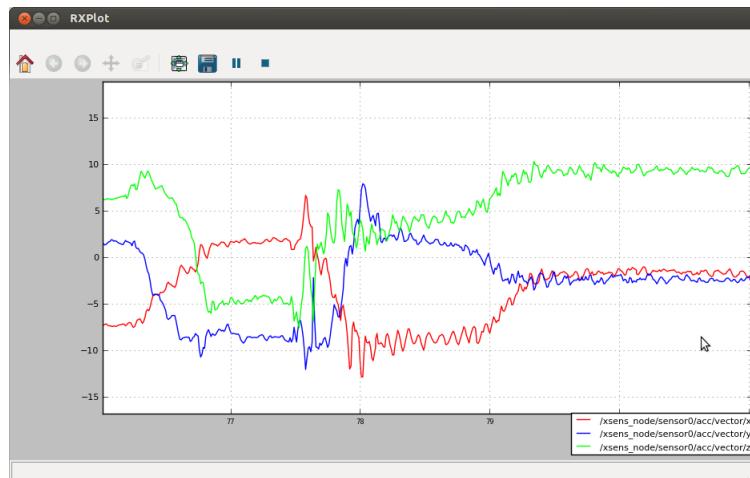


Figura 4.8: Programa rxplot mostrando una gráfica con los datos de los acelerómetros de un sensor en tiempo real

```
$ rxplot -r 10 /xsens_node/sensor0/acc/vector/x:y:z
```

■ Visualización 3D

ROS incluye varios paquetes para la visualización y simulación 3D de objetos. Entre ellos, los más importantes son:

- **rviz:** El paquete rviz incluido en ROS es un paquete que permite la lectura de topics de ROS desde el propio programa, y asignar los valores a parámetros de la visualización. No es un simulador de física de objetos 3D, por lo que no permite la simulación de robots propiamente dicha.
- **gazebo:** El programa Gazebo es un simulador de física de objetos rígidos. Permite la simulación de un robot o de un conjunto de robots, y su interacción entre ellos y con el ambiente. Además es posible obtener una simulación de la respuesta que tendrían ciertos tipos de sensores en el mundo real.

En la documentación del robot YouBot viene incluida una simulación del brazo del robot en Gazebo (figura 4.10) con la que se puede interactuar mediante



Figura 4.9: El simulador *Gazebo*.

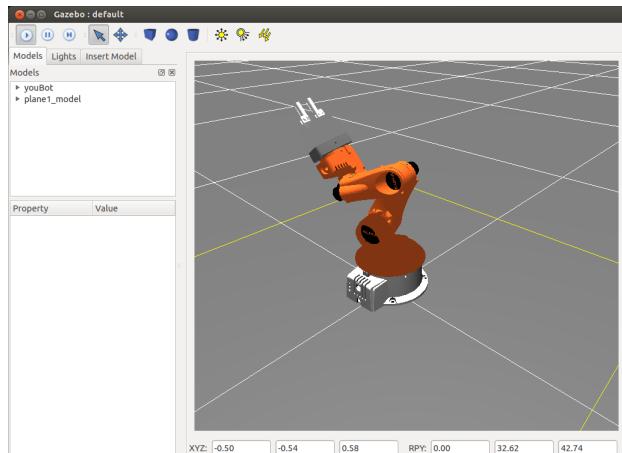


Figura 4.10: Programa Gazebo con la simulación del brazo del Youbot

ciertos topics de ROS.

4.3.3. ROS como sistema distribuido

ROS ha sido diseñado para poder funcionar en una red de ordenadores de forma distribuida. Se puede crear una red de nodos ejecutándose simultáneamente en varios PCs conectados entre sí. Solamente hay que tener en cuenta lo siguiente:

- Sólo es necesario un máster para todos los ordenadores conectados. Se ha de determinar inicialmente en qué ordenador se localizará el máster.
- Todos los nodos serán configurados para usar el mismo máster. La variable de entorno que define la dirección del máster es ROS_MASTER_URI. Esta variable se tiene que definir al iniciar un *shell* mediante el comando:

```
$ export ROS_MASTER_URI=http://direccion_del_master:11311
```

- Tiene que haber conectividad completa y bidireccional entre cada par de máquinas.
- Cada máquina tiene que tener un nombre único –normalmente se utilizará su dirección IP– que el resto de máquinas puedan entender. Este nombre se establece en la variable de entorno ROS_IP. Para definir su valor se ejecutará el siguiente comando al abrir un nuevo *shell*:

```
$ export ROS_IP=mi_direccion_ip
```

Una vez establecidas las variables ROS_IP y ROS_MASTER_URI en cada máquina, todos los nodos podrían ser ejecutados indistintamente en cualquier lugar de la red –a excepción de los drivers, que han de ejecutarse en la misma máquina que el hardware del que realizan el control–, y el conjunto debería comportarse tal y como lo haría si estuviera en un único ordenador.

4.4. Sistema operativo

La plataforma ROS ha sido creada para trabajar principalmente en el sistema operativo Ubuntu. Existe otras versiones capaces de funcionar en otros sistemas operativos como Windows, pero son experimentales y no poseen todas las funcionalidades de la versión principal, además de carecer de soporte y ser algo inestables. Por ello, el sistema operativo elegido para la realización y ejecución del software será Ubuntu.

4.4.1. El sistema operativo Ubuntu

Ubuntu es un sistema operativo con núcleo Linux, distribuido como software libre y de código abierto. Es la distribución más popular de los sistemas GNU/Linux en ordenadores personales. En este proyecto se hará un uso intensivo del intérprete de comandos (*shell* o terminal) de Ubuntu, ya que permite ejecutar los comandos de ROS de forma rápida y cómoda.



Figura 4.11: El sistema operativo Ubuntu.

Capítulo 5

CÁLCULO DE LAS POSICIONES Y ORIENTACIONES DE LOS SENSORES

Uno de los objetivos del proyecto es capturar el estado de orientación de los sensores para la utilizar los datos obtenidos en otros programas. Para ello es necesario utilizar un sistema de representación matemática de las orientaciones que permita la derivación de las magnitudes que se deseen (como ejes y ángulos de rotación) de forma rápida e inequívoca. En este capítulo se pretende dar una visión general de la derivación matemática y justificación de los algoritmos cuya implementación se mostrará en el capítulo *IMPLEMENTACIÓN DEL SOFTWARE*.

5.1. Formas de representar orientaciones espaciales

Existen multitud de representaciones matemáticas de la orientación de un sólido rígido:

- Ángulos de Euler
- Eje y ángulo de Euler
- Matriz de rotación
- Cuaternión
- Parámetros de Rodrigues
- Parámetros de Cayley-Klein
- ...

De todas estas formas de representación, los sensores xsens, al igual que muchos IMUs modernos, proporcionan la posibilidad de obtener los ángulos de Euler, la matriz de rotación y el cuaternión de rotación. En los siguientes apartados se analizarán las ventajas y desventajas de dichos sistemas y se escogerá la más conveniente.

5.1.1. Ángulos de Euler

Los ángulos de Euler son tres ángulos introducidos por Leonhard Euler para describir la orientación de un sólido rígido o de un sistema de referencia respecto a otro. Representan una secuencia de tres rotaciones elementales alrededor de los ejes de un sistema de coordenadas. Cualquier orientación puede describirse como la composición de tres rotaciones elementales, que pueden suceder alrededor de los ejes de un sistema de referencia fijo (rotaciones extrínsecas) o alrededor de los ejes de un sistema solidario al sólido rígido (rotaciones intrínsecas), lo que se denomina un sistema de referencia local.

Existen multitud de maneras distintas de expresar los ángulos de Euler, dependiendo del orden de los ejes sobre los cuales se realizan las rotaciones, y de si el sistema de referencia es local o global. La definición usada por los sensores xsens es la composición de rotaciones alrededor de los ejes XYZ en ese orden y un sistema de referencia global (fijo a la Tierra), en el que la ausencia de rotación equivale al vector Z paralelo a la línea que une la posición del sensor con el centro de la Tierra y sentido ascendente, y el vector X en dirección al norte magnético.

Los ángulos proporcionados por el sensor son:

- $\phi = roll$ = rotación alrededor del eje X $[-\frac{\pi}{2}, \frac{\pi}{2}]$
- $\theta = pitch$ = rotación alrededor del eje Y $[-\frac{\pi}{4}, \frac{\pi}{4}]$
- $\psi = yaw$ = rotación alrededor del eje Z $[-\frac{\pi}{2}, \frac{\pi}{2}]$

El uso de ángulos de Euler presenta un par de problemas:

- Si θ se extiende al intervalo $[-\frac{\pi}{2}, \frac{\pi}{2}]$ en vez de restringirse a $[-\frac{\pi}{4}, \frac{\pi}{4}]$, la descripción de la rotación no es única. En este caso existen dos posibles soluciones para cada orientación, por lo que los datos que proporciona el sensor no son suficientes para diferenciar entre un estado de rotación u otro.
- Cuando θ se acerca al valor $\pm\frac{\pi}{4}$ existe una singularidad matemática causada por la infinitud de valores que pueden adquirir ϕ y ψ en dicho caso. Esta situación es la que se conoce con el nombre de *gimbal lock*.

Estos problemas no están presentes en los demás modos de salida del sensor.

5.1.2. Matriz de rotación

Una matriz de rotación es una matriz usada para expresar una rotación en un espacio euclídeo. Si se supone un sistema generador S con un conjunto de vectores base B del espacio vectorial V :

$$B = \{\vec{i}, \vec{j}, \vec{k}\}, \quad \vec{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \vec{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \vec{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$V = S(\vec{i}, \vec{j}, \vec{k})$$

Se somete al sistema de coordenadas a una rotación R . Los vectores base sufren una transformación que se puede expresar con respecto al sistema de referencia original de la siguiente forma:

$$\vec{i}' = \alpha_{11}\vec{i} + \alpha_{21}\vec{j} + \alpha_{31}\vec{k} = \begin{bmatrix} \alpha_{11} \\ \alpha_{21} \\ \alpha_{31} \end{bmatrix}$$

$$\vec{j}' = \alpha_{12}\vec{i} + \alpha_{22}\vec{j} + \alpha_{32}\vec{k} = \begin{bmatrix} \alpha_{12} \\ \alpha_{22} \\ \alpha_{32} \end{bmatrix}$$

$$\vec{k}' = \alpha_{13}\vec{i} + \alpha_{23}\vec{j} + \alpha_{33}\vec{k} = \begin{bmatrix} \alpha_{13} \\ \alpha_{23} \\ \alpha_{33} \end{bmatrix}$$

donde α_{ij} son las componentes de la base rotada expresadas con respecto a la base original. Se tiene un vector cualquiera en la base original:

$$\vec{v} = a\vec{i} + b\vec{j} + c\vec{k} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

Tras la rotación las componentes del vector expresadas conforme a la nueva base no variarán debido a que se mueve solidario al sistema de referencia. Por lo tanto el vector rotado es el siguiente:

$$\vec{v}' = a\vec{i}' + b\vec{j}' + c\vec{k}'$$

Sustituyendo los vectores $\vec{i}', \vec{j}', \vec{k}'$ por sus expresiones respecto a la base original:

$$\vec{v}' = a \begin{bmatrix} \alpha_{11} \\ \alpha_{21} \\ \alpha_{31} \end{bmatrix} + b \begin{bmatrix} \alpha_{12} \\ \alpha_{22} \\ \alpha_{32} \end{bmatrix} + c \begin{bmatrix} \alpha_{13} \\ \alpha_{23} \\ \alpha_{33} \end{bmatrix} = \begin{bmatrix} a\alpha_{11} + b\alpha_{12} + c\alpha_{13} \\ a\alpha_{21} + b\alpha_{22} + c\alpha_{23} \\ a\alpha_{31} + b\alpha_{32} + c\alpha_{33} \end{bmatrix}$$

Este nuevo vector puede expresarse como el producto de una matriz por un vector:

$$\vec{v}' = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = R\vec{v}$$

Dicha matriz R es la denominada matriz de rotación. Se comprueba que puede interpretarse como una matriz cuyos elementos son las componentes de los vectores base del sistema de coordenadas rotado expresados con respecto al sistema de coordenadas original.

El uso de matrices de rotación evita los problemas que presentan los ángulos de Euler; Proporcionan una descripción biunívoca del estado de rotación del sensor, y además no presentan el problema del *gimbal lock*.

5.1.3. Cuaternión

Los cuaterniones son una extensión de los números complejos ideada por el matemático irlandés William Rowand Hamilton con el objetivo de poder utilizar el análisis complejo en un espacio de 3 dimensiones. Un cuaternión es la combinación lineal de cuatro cantidades $\{1, i, j, k\}$, donde cada una de las cantidades $\{i, j, k\}$ es la raíz cuadrada de -1 , de tal forma que se cumplen las siguientes propiedades:

$$i^2 = j^2 = k^2 = ijk = -1$$

La expresión general de un cuaternión es la siguiente:

$$q = w + xi + yj + zk \quad (5.1)$$

donde w, x, y y z son números reales. Los cuaterniones satisfacen las leyes conmutativa y asociativa de la suma, la ley asociativa de la multiplicación, las leyes distributivas de la multiplicación respecto a la suma y la existencia de los elementos neutros para la suma y la multiplicación. Una propiedad importante de los cuaterniones es que no satisfacen la propiedad conmutativa de la multiplicación.

Propiedades de los cuaterniones	
Conmutativa respecto a la suma	$q_1 + q_2 = q_2 + q_1$
Asociativa respecto a la suma	$q_1 + (q_2 + q_3) = (q_1 + q_2) + q_3$
Asociativa respecto a la multiplicación	$q_1(q_2q_3) = (q_1q_2)q_3$
Distributiva de la multiplicación respecto a la suma	$q_1(q_2 + q_3) = q_1q_2 + q_1q_3$ $(q_1 + q_2)q_3 = q_1q_3 + q_2q_3$
Elemento neutro de la suma	$q + 0 = q$
Elemento neutro de la multiplicación	$q1 = 1q = q$
No conmutatividad de la multiplicación	$q_1q_2 \neq q_2q_1$

Cuadro 5.1: Propiedades de los cuaterniones

Los cuaterniones presentan las mismas ventajas que las matrices de rotación en cuanto a unicidad y ausencia de *gimbal lock*. Pero además son superiores a las matrices en los siguientes aspectos:

- Su representación en memoria es más compacta que la de las matrices (4 números frente a 9 necesarios para la matriz).
- Se puede construir fácilmente un cuaternión a partir de un eje y un ángulo, y viceversa. Estas operaciones son más complejas para matrices de rotación y ángulos de Euler.
- Mayor estabilidad numérica de los cuaterniones. Tras la composición de varias rotaciones en un ordenador necesariamente se van a acumular errores de redondeo. Para que los cuaterniones y matrices representen una rotación deben ciertas propiedades. En concreto, los cuaterniones tienen que ser unitarios y la matriz de rotación tiene que ser ortogonal. Es bastante más fácil normalizar un cuaternión para que vuelva a representar una rotación que recomponer una matriz para que vuelva a ser ortogonal.
- Además con los cuaterniones es sencillo componer una interpolación esférica (llamada *slerp - spherical linear interpolation*) para producir una rotación suave a lo largo del tiempo.

5.1.4. Solución elegida: cuaterniones

Por la gran cantidad de ventajas que presentan los cuaterniones con respecto a las demás formas de representar una rotación, éstos serán los escogidos para registrar el estado de orientación de los sensores y realizar los cálculos matemáticos.¹

5.2. Utilización de cuaterniones para la representación de rotaciones de un sólido rígido

En esta sección se verá cómo describir rotaciones simples y composiciones de rotaciones de un sólido rígido mediante el uso de cuaterniones.

Sea un vector unitario:

$$\vec{e} = e_x i + e_y j + e_z k, \quad \|e\| = \sqrt{e_x^2 + e_y^2 + e_z^2} = 1 \quad (5.2)$$

Se definirá el cuaternion de rotación con ángulo θ sobre el eje definido por el vector \vec{e} :

$$q(\theta, \vec{e}) = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) e_x i + \sin\left(\frac{\theta}{2}\right) e_y j + \sin\left(\frac{\theta}{2}\right) e_z k \quad (5.3)$$

Este cuaternion representa la orientación de un sólido tras sufrir una rotación de ángulo θ alrededor del eje definido por el vector \vec{e} . Se puede descomponer el cuaternion como suma de un número real y un cuaternion imaginario puro unitario multiplicado por otro número real:

$$q(\theta, \vec{e}) = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) e \quad (5.4)$$

Donde e es el cuaternion imaginario puro (parte real nula) cuyas componentes se corresponden a las del vector unitario \vec{e} que define el eje de rotación.

Por cuestión de comodidad definimos las siguientes variables:

$$c_i = \cos\left(\frac{\theta_i}{2}\right) \quad (5.5)$$

$$s_i = \sin\left(\frac{\theta_i}{2}\right) \quad (5.6)$$

De tal forma que ahora el cuaternion se escribirá de la siguiente manera:

$$q(\theta_i, \vec{e}_i) = c_i + s_i e_i \quad (5.7)$$

Si se tienen en cuenta las propiedades del cuadro 5.1, el producto de dos cuaterniones se puede expresar de esta forma:

$$q_1 q_2 = (c_1 + s_1 e_1)(c_2 + s_2 e_2) = c_1 c_2 + c_1 s_2 e_2 + c_2 s_1 e_1 + s_1 s_2 e_1 e_2 \quad (5.8)$$

¹Otra razón de índole no matemática para la elección de los cuaterniones es que el simulador Gazebo representa las orientaciones de los cuerpos mediante cuaterniones. Utilizando cuaterniones no existirá el problema de tener que generar el cuaternion de rotación cada vez que se publique un mensaje dirigido a Gazebo.

En donde:

$$\begin{aligned}
 e_1e_2 &= (e_{1_x}i + e_{1_y}j + e_{1_z}k)(e_{2_x}i + e_{2_y}j + e_{2_z}k) = \\
 &= e_{1_x}i(e_{2_x}i + e_{2_y}j + e_{2_z}k) + e_{1_y}j(e_{2_x}i + e_{2_y}j + e_{2_z}k) + e_{1_z}k(e_{2_x}i + e_{2_y}j + e_{2_z}k) = \\
 &= -(e_{1_x}e_{2_x} + e_{1_y}e_{2_y} + e_{1_z}e_{2_z}) + i(e_{1_y}e_{2_z} - e_{1_z}e_{2_y}) + j(-e_{1_x}e_{2_z} + e_{1_z}e_{2_x}) + k(e_{1_x}e_{2_y} - e_{1_y}e_{2_x})
 \end{aligned} \tag{5.9}$$

Se definirán las siguientes operaciones con cuaterniones imaginarios puros (parte real nula):

Definición 5.2.1. Sean dos cuaterniones imaginarios puros q_1 y q_2 tales que:

$$q_1 = x_1i + y_1j + z_1k$$

$$q_2 = x_2i + y_2j + z_2k$$

Se define el operador producto escalar (\cdot) como:

$$q_1 \cdot q_2 = x_1x_2 + y_1y_2 + z_1z_2$$

Este operador presenta las mismas propiedades que el mismo operador para vectores de 3 dimensiones:

$$q_1 \cdot q_2 = q_2 \cdot q_1$$

$$q_1 \cdot q_1 = \|q_1\|^2$$

$$q_1 \cdot q_2 = 0 \iff q_1 \perp q_2$$

Definición 5.2.2. Sean dos cuaterniones imaginarios puros q_1 y q_2 tales que:

$$q_1 = x_1i + y_1j + z_1k$$

$$q_2 = x_2i + y_2j + z_2k$$

Se define el operador producto vectorial (\times) como:

$$q_1 \times q_2 = \begin{bmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{bmatrix} = (y_1z_2 - z_1y_2)i + (z_1x_2 - x_1z_2)j + (x_1y_2 - y_1x_2)k$$

Este operador presenta las mismas propiedades que el mismo operador para vectores de 3 dimensiones:

$$q_1 \times q_2 = -(q_2 \times q_1)$$

$$q_1 \times q_2 = 0 \iff q_1 \parallel q_2$$

$$q_1 \times q_2 = q_3 : q_3 \perp q_1 \wedge q_3 \perp q_2$$

Utilizando la definición de estos operadores se puede simplificar la ecuación (5.9):

$$e_1e_2 = -e_1 \cdot e_2 + e_1 \times e_2 \tag{5.10}$$

Por lo tanto, sustituyendo (5.10) en la ecuación (5.8):

$$q_1q_2 = c_1c_2 + c_1s_2e_2 + c_2s_1e_1 + s_1s_2(-e_1 \cdot e_2 + e_1 \times e_2)$$

$$q_1q_2 = c_1c_2 - s_1s_2(e_1 \cdot e_2) + c_1s_2e_2 + c_2s_1e_1 + s_1s_2(e_1 \times e_2) \tag{5.11}$$

Definición 5.2.3. El conjugado de un cuaternión $q = c + se$ se define como el cuaternión resultado de negar la parte imaginaria. Se representará de la siguiente manera:

$$\text{conj}(q) = q^* = c - se \quad (5.12)$$

Definición 5.2.4. La norma o magnitud de un cuaternión $q = w + xi + yj + zk$ se define como la siguiente operación:

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2} \quad (5.13)$$

Esta magnitud representaría una hipotética longitud que tendría el cuaternión si el espacio de los cuaterniones \mathbb{H} se identifica con un espacio euclídeo \mathbb{R}^4 . Para un cuaternión de rotación se tiene que:

$$q = c + se, \quad \|q\| = \sqrt{c^2 + \|se\|^2} = \sqrt{c^2 + s^2} = \sqrt{\cos^2 \frac{\theta}{2} + \sin^2 \frac{\theta}{2}} = 1$$

Dado a que se va a trabajar siempre con cuaterniones unitarios (también llamados *versores*), se podrá asumir lo siguiente:

$$\|q\| = 1, \quad q^{-1} = \frac{q^*}{\|q\|^2} = q^* \quad (5.14)$$

De tal forma que:

$$qq^* = q^*q = 1 \quad (5.15)$$

5.2.1. Rotación de un vector alrededor de un eje y un ángulo dados

Se puede realizar la rotación de un vector \vec{p} alrededor de un eje \vec{e} y un ángulo θ mediante la siguiente operación:

$$p' = qpq^*, \quad q = q(\theta, \vec{e}) \quad (5.16)$$

donde p es el cuaternión asociado al vector \vec{p} , que se define como:

$$\vec{p} = p_x \vec{i} + p_y \vec{j} + p_z \vec{k} \Rightarrow p = p_x i + p_y j + p_z k \quad (5.17)$$

Se comprueba que en efecto la ecuación (5.16) es cierta. Desarrollando la expresión para un cuaternión genérico q y un vector expresado como cuaternion Pp , donde P es el módulo del vector y p es el cuaternion asociado al vector normalizado:

$$q(Pp)q^* = P(qpq^*) = P(c+se)p(c-se) = P(cp+sep)(c-se) = P(c^2p - cspe + csep - s^2epe)$$

Se aplican las propiedades de los cuaterniones unitarios vistas anteriormente:

$$q(Pp)q^* = P(c^2p + 2cs(e \times p) + s^2(e \cdot p)e - s^2(e \times p) \times e) \quad (5.18)$$

Se le puede asignar a cada término un significado geométrico:

- $(e \cdot p)e$ es la proyección de p sobre e

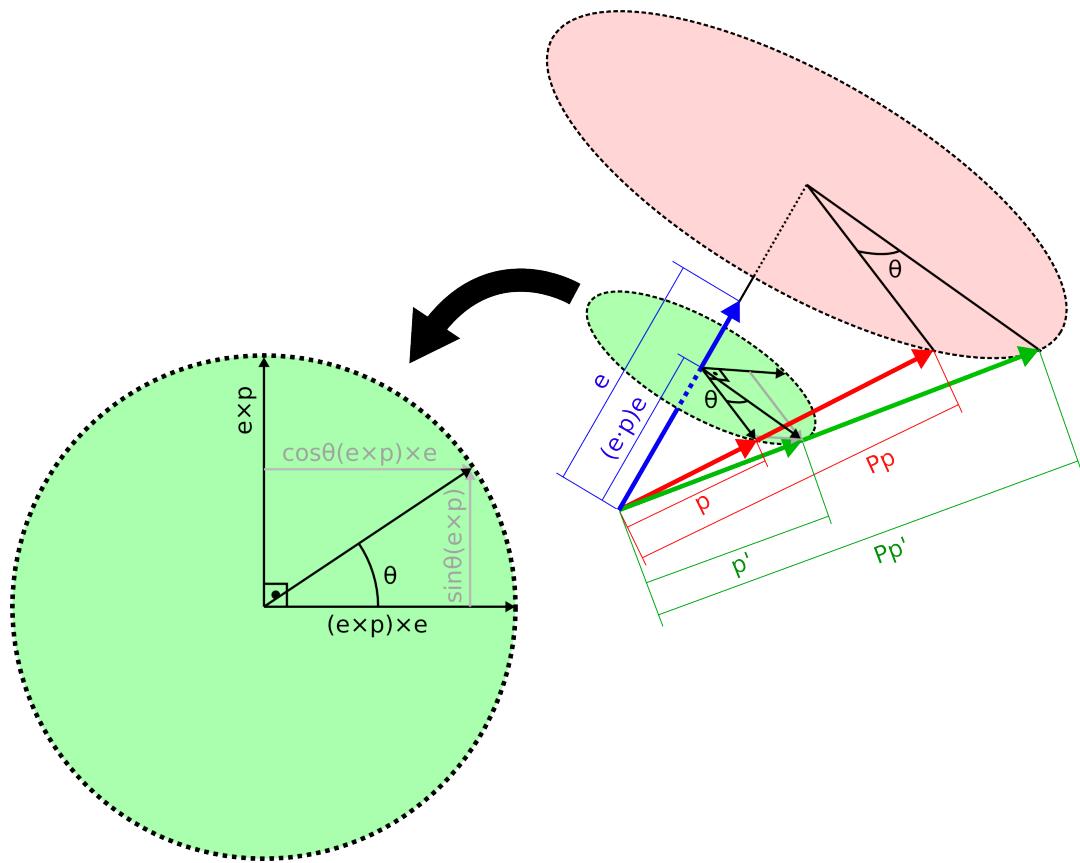


Figura 5.1: Rotación de un vector mediante la operación $q_1 p q_1^*$

- $(e \times p)$ es un vector perpendicular a e y a p . Formaría una supuesta coordenada y en el círculo de giro.
- $(e \times p) \times e$ es un vector perpendicular al anterior, cuyo origen podemos situar al final de $(e \cdot p)e$ y su final, en el mismo punto que p . Sería la coordenada x del círculo de giro.

En la figura 5.1 se muestra más claramente el significado de cada término.

Es posible descomponer así el término $c^2 p$ como suma de dos de estos vectores:

$$c^2 p = c^2(e \cdot p)e + c^2(e \times p) \times e$$

Sustituyendo el resultado en (5.18):

$$q(Pp)q^* = P(c^2(e \cdot p)e + c^2(e \times p) \times e + 2cs(e \times p) + s^2(e \cdot p)e - s^2(e \times p) \times e)$$

Reorganizando términos:

$$q(Pp)q^* = P((c^2 + s^2)(e \cdot p)e + (c^2 - s^2)(e \times p) \times e + 2cs(e \times p))$$

Si se aplican igualdades trigonométricas a esta expresión, teniendo en cuenta que $c = \cos(\frac{\theta}{2})$ y $s = \sin(\frac{\theta}{2})$:

$$q(Pp)q^* = P((e \cdot p)e + \cos\theta(e \times p) \times e + \sin\theta(e \times p))$$

De la figura 5.1 se deduce que $p' = (e \cdot p)e + \cos \theta(e \times p) \times e + \sin \theta(e \times p)$, y por lo tanto se puede concluir que:

$$q(Pp)q^* = Pp'$$

Por lo que para un vector cualquiera \vec{v} se cumple que $qvq^* = v'$, donde v' es el cuaternión asociado al vector rotado.

Si se realiza una multiplicación a la izquierda por q_1^* y por la derecha por q_1 a cada término de la ecuación (5.16) (lo que equivale a realizar una rotación dada por el cuaternión q_1^*):

$$q_1^*p'q_1 = q_1^*(q_1pq_1)q_1^* \quad (5.19)$$

$$q_1^*p'q_1 = p \quad (5.20)$$

se obtiene el punto inicial, de lo que se deduce que el conjugado del cuaternión representa una rotación inversa a la del cuaternión original:

$$q = q(\theta, \vec{e}) \Rightarrow q^* = q(\theta, -\vec{e}) = q(-\theta, \vec{e}) \quad (5.21)$$

5.2.2. Composición de rotaciones en coordenadas extrínsecas

El cuaternión de rotación definido por $q(\theta, \vec{e})$ representa una rotación alrededor de un eje \vec{e} fijo al sistema de referencia global.

Realizando una nueva rotación a un vector que ha sufrido una rotación definida por $q_1 = q(\theta_1, \vec{e}_1)$, se obtendrá una rotación compuesta por una primera rotación seguida de otra rotación caracterizada por $q_2 = q(\theta_2, \vec{e}_2)$:

$$\begin{aligned} \text{Primera rotación: } & p' = q_1pq_1^*, \quad q_1 = q(\theta_1, \vec{e}_1) \\ \text{Segunda rotación: } & p'' = q_2p'q_2^*, \quad q_2 = q(\theta_2, \vec{e}_2) \end{aligned}$$

$$p'' = q_2(q_1pq_1^*)q_2^* = (q_2q_1)p(q_1^*q_2^*) = q_{12}pq_{12}^* \quad (5.22)$$

Se puede expresar la composición de dos rotaciones como un nuevo cuaternión que resulta de la multiplicación en orden inverso de los cuaterniones que definen las dos rotaciones:

$$q_{12} = q_2q_1 \quad (5.23)$$

De aquí se deduce que el conjugado del producto de dos cuaterniones es el producto de los conjugados en orden inverso:

$$q_{12} = (q_2q_1)^* = q_1^*q_2^* \quad (5.24)$$

De forma análoga, para n cuaterniones:

$$q_{12\dots(n-1)n} = q_nq_{n-1}\dots q_2q_1 \quad (5.25)$$

$$(q_{12\dots(n-1)n})^* = (q_nq_{n-1}\dots q_2q_1)^* = q_1^*q_2^*\dots q_{n-1}^*q_n^* \quad (5.26)$$

5.2.3. Composición de rotaciones en coordenadas intrínsecas

Para representar una rotación alrededor de un eje expresado en el sistema de referencia local del sólido, se tendrá que realizar la construcción del cuaternion teniendo en cuenta que dicho eje ha sufrido la misma rotación que el sólido con respecto al sistema de referencia global. Supongamos un sólido que ha sufrido una rotación inicial representada por el cuaternion q_1 . Se quiere realizar una rotación de ángulo θ_2 sobre un eje e_2^L en coordenadas locales. Dicho eje expresado en coordenadas globales será:

$$e_2^G = q_1 e_2^L q_1^*$$

Teniendo en cuenta la ecuación (5.7), el cuaternion de rotación asociado al ángulo θ_2 y el vector local e_2^L después de que el sólido haya sufrido una rotación definida por el cuaternion q_1 será:

$$\begin{aligned} q_{2_{q_1}} &= q(\theta_2, \vec{e}_2^G) = \cos\left(\frac{\theta_2}{2}\right) + \sin\left(\frac{\theta_2}{2}\right) e_2^G \\ q_{2_{q_1}} &= c_2 + s_2(q_1 e_2^L q_1^*) \end{aligned} \quad (5.27)$$

5.2.4. Relación entre rotaciones intrínsecas y extrínsecas

El algoritmo de obtención de los ángulos de Euler que se explicará más adelante proporciona una solución en el caso de ejes solidarios al sensor, esto es, en un sistema de coordenadas intrínsecas. Es posible obtener con este algoritmo soluciones en un sistema extrínseco que pueden ser de utilidad en ciertas aplicaciones. A continuación se demostrará que una rotación compuesta por varias rotaciones en el sistema de coordenadas intrínseco del sólido rígido se corresponde a la composición de rotaciones en el sistema extrínseco realizadas en orden inverso.

Se define un cuaternion de rotación asociado al eje local \vec{e}_2^L después de haber sufrido una rotación definida por el cuaternion $q_1 = q(\theta_1, \vec{e}_1)$ como:

$$q_{2_{q_1}} = q(\theta_2, \vec{e}_2^L), \quad \vec{e}_2^L = (q_1 e_2 q_1^*) \vec{v} \quad (5.28)$$

Se va a suponer sin pérdida de generalidad que la primera rotación se ha realizado desde una posición en la que coinciden los sistemas local y global ², por lo que:

$$q_1^G = q_1^L = q_1 \quad (5.29)$$

donde q_1^G es la rotación alrededor de un eje en coordenadas globales y q_1^L la rotación en coordenadas locales.

Para demostrar la afirmación de partida se tendrá que demostrar la veracidad de la siguiente igualdad:

$$q_2^L q_1 = q_1 q_2^G \quad (5.30)$$

²Por definición el origen del cuaternion de orientación ($q = 1$) del sensor es la orientación donde coinciden el sistema global y local. Por ello, el eje de rotación en los dos sistemas coincidirá y por lo tanto el cuaternion de rotación será el mismo.

Se reordenará la igualdad para que los cálculos sean más sencillos:

$$q_2^L q_1 = q_1 q_2^G \iff q_2^L = q_1 q_2^G q_1^* \quad (5.31)$$

Desarrollo del lado izquierdo de la igualdad:

$$q_2^L = c_2 + s_2(q_1 e_2 q_1^*) \quad (5.32)$$

$$\begin{aligned} q_1 e_2 q_1^* &= (c_1 + s_1 e_1) e_2 (c_1 - s_1 e_1) = (c_1 e_2 + s_1 e_1 e_2) (c_1 - s_1 e_1) = \\ &= c_1^2 e_2 + s_1 c_1 e_1 e_2 - c_1 s_1 e_2 e_1 - s_1^2 e_1 e_2 e_1 = c_1^2 e_2 + s_1 c_1 (e_1 e_2 - e_2 e_1) - s_1^2 e_1 e_2 e_1 \end{aligned} \quad (5.33)$$

De (5.10) se tiene lo siguiente:

$$\begin{aligned} e_1 e_2 &= -e_1 \cdot e_2 + e_1 \times e_2 \\ e_2 e_1 &= -e_2 \cdot e_1 + e_2 \times e_1 = -e_1 \cdot e_2 - e_1 \times e_2 \\ e_1 e_2 - e_2 e_1 &= 2(e_1 \times e_2) \end{aligned} \quad (5.34)$$

$$e_1 e_2 e_1 = (-e_1 \cdot e_2 + e_1 \times e_2) e_1 = (-e_1 \cdot e_2) e_1 + (e_1 \times e_2) e_1$$

De esta ecuación:

$$\begin{aligned} (e_1 \times e_2) e_1 &= -(e_1 \times e_2) \cdot e_1 + (e_1 \times e_2) \times e_1 \\ (e_1 \times e_2) \text{ será un vector perpendicular a } e_1, \text{ por lo que } (e_1 \times e_2) \cdot e_1 &= 0: \end{aligned}$$

$$(e_1 \times e_2) e_1 = (e_1 \times e_2) \times e_1$$

Por lo tanto:

$$e_1 e_2 e_1 = -(e_1 \cdot e_2) e_1 + (e_1 \times e_2) \times e_1 \quad (5.35)$$

Sustituyendo en (5.33):

$$\begin{aligned} q_1 e_2 q_1^* &= c_1^2 e_2 + s_1 c_1 (e_1 e_2 - e_2 e_1) - s_1^2 e_1 e_2 e_1 = \\ &= c_1^2 e_2 + 2s_1 c_1 (e_1 \times e_2) - s_1^2 ((-e_1 \cdot e_2) e_1 + (e_1 \times e_2) \times e_1) \\ q_1 e_2 q_1^* &= c_1^2 e_2 + 2s_1 c_1 (e_1 \times e_2) + s_1^2 (e_1 \cdot e_2) e_1 - s_1^2 (e_1 \times e_2) \times e_1 \end{aligned} \quad (5.36)$$

Finalmente, sustituyendo en (5.32):

$$q_2^L = c_2 + s_2(c_1^2 e_2 + 2s_1 c_1 (e_1 \times e_2) + s_1^2 (e_1 \cdot e_2) e_1 - s_1^2 (e_1 \times e_2) \times e_1)$$

$$q_2^L = c_2 + s_2 c_1^2 e_2 + 2s_1 s_2 c_1 (e_1 \times e_2) + s_1^2 s_2 (e_1 \cdot e_2) e_1 - s_1^2 s_2 (e_1 \times e_2) \times e_1 \quad (5.37)$$

Ahora se procederá a desarrollar el lado derecho de la igualdad (5.31):

$$\begin{aligned}
q_1 q_2^G q_1^* &= (c_1 + s_1 e_1)(c_2 + s_2 e_2)(c_1 - s_1 e_1) = \\
&= (c_1 c_2 + s_2 c_1 e_2 + s_1 c_2 e_1 + s_1 s_2 e_1 e_2)(c_1 - s_1 e_1) = \\
&= c_1^2 c_2 + s_2 c_1^2 e_2 + s_1 c_1 c_2 e_1 + s_1 s_2 c_1 e_1 e_2 - s_1 c_1 c_2 e_1 - s_1 s_2 c_1 e_2 e_1 - s_1^2 c_2 e_1 e_1 - s_1^2 s_2 e_1 e_2 e_1
\end{aligned}$$

Como $\|e_1\| = 1$, se tendrá que:

$$e_1 e_1 = -e_1 \cdot e_1 + e_1 \times e_1 = -1 \quad (5.38)$$

Sustituyendo y reorganizando:

$$q_1 q_2^G q_1^* = c_2 + s_2 c_1^2 e_2 + s_1 s_2 c_1 (e_1 e_2 - e_2 e_1) - s_1^2 s_2 e_1 e_2 e_1$$

De (5.34) y (5.35):

$$q_1 q_2^G q_1^* = c_2 + s_2 c_1^2 e_2 + 2s_1 s_2 c_1 (e_1 \times e_2) - s_1^2 s_2 (-e_1 \cdot e_2) e_1 + (e_1 \times e_2) \times e_1$$

$$q_1 q_2^G q_1^* = c_2 + s_2 c_1^2 e_2 + 2s_1 s_2 c_1 (e_1 \times e_2) + s_1^2 s_2 (e_1 \cdot e_2) e_1 - s_1^2 s_2 (e_1 \times e_2) \times e_1 \quad (5.39)$$

Finalmente se compara (5.37) con (5.39):

$$q_2^L = c_2 + s_2 c_1^2 e_2 + 2s_1 s_2 c_1 (e_1 \times e_2) + s_1^2 s_2 (e_1 \cdot e_2) e_1 - s_1^2 s_2 (e_1 \times e_2) \times e_1$$

$$q_1 q_2^G q_1^* = c_2 + s_2 c_1^2 e_2 + 2s_1 s_2 c_1 (e_1 \times e_2) + s_1^2 s_2 (e_1 \cdot e_2) e_1 - s_1^2 s_2 (e_1 \times e_2) \times e_1$$

Se puede ver que los términos a la derecha de la igualdad son exactamente los mismos, por lo que se tiene que:

$$q_2^L = q_1 q_2^G q_1^* \quad (5.40)$$

Y por lo tanto es verdad la afirmación de partida. Una rotación q_2^L alrededor de un eje local tras una rotación q_1 es la misma que la rotación global q_2^G seguida de la rotación q_1 :

$$q_2^L q_1 = q_1 q_2^G \quad (5.41)$$

Ahora se va a suponer que el cuaternion q_1 está compuesto de otras dos rotaciones, que se van a expresar de forma global y local según la ecuación (5.41):

$$q_1 = q_{1b}^L q_{1a} = q_{1a} q_{1b}^G$$

Sustituyendo en (5.41):

$$q_2^L q_{1b}^L q_{1a} = q_{1a} q_{1b}^G q_2^G$$

Renombrando los términos:

$$q_3^L q_2^L q_1 = q_1 q_2^G q_3^G$$

Aplicando el mismo método de forma sucesiva se puede generalizar el resultado a n rotaciones:

$$q_n^L \cdots q_3^L q_2^L q_1 = q_1 q_2^G q_3^G \cdots q_n^G \quad (5.42)$$

Se concluye que la composición de n rotaciones en coordenadas locales es la misma que la composición de n rotaciones en coordenadas globales realizadas en orden inverso.

5.2.5. Orientación relativa entre dos sólidos

Se tienen dos sólidos con sendos sistemas de coordenadas S_1 y S_2 . Definiremos la rotación relativa del sólido 1 sobre el sólido 2 como la rotación existente entre sus sistemas de coordenadas.

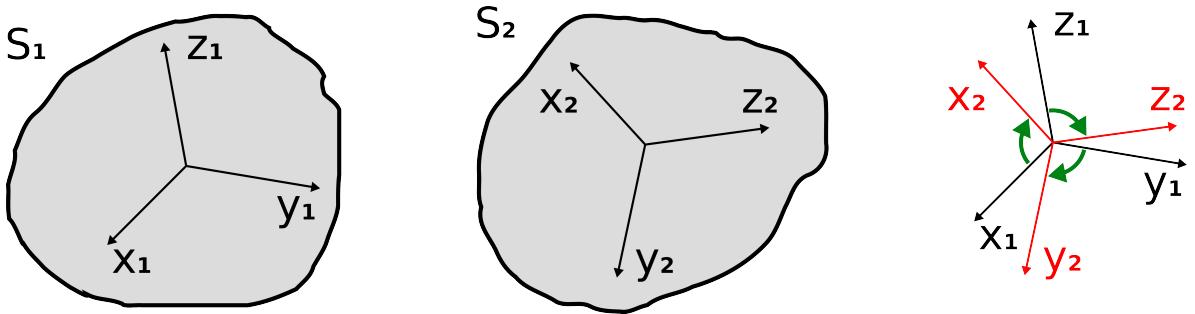


Figura 5.2: Orientación relativa entre dos sólidos rígidos

Otra forma de verlo es que la rotación relativa es la rotación que tendría el sólido 2 si el sistema de coordenadas del sólido 1 fuera el global. De esta forma se podrá calcular el cuaternion que define la orientación relativa entre los dos sólidos aplicando una rotación a ambos de forma que el sistema de coordenadas del sólido 1 coincida con el global. Se llamará q_1 al cuaternion que define la orientación del sólido 1 y q_2 al que define la orientación del sólido q_2 . Si se aplica una rotación q_1^* a ambos, esto es, una rotación inversa a la del sólido 1, se obtiene lo siguiente:

$$\text{Sólido 1: } q'_1 = q_1^* q_1 = 1$$

$$\text{Sólido 2: } q'_2 = q_1^* q_2 = q_{S_2 S_1}$$

donde $q_{S_2 S_1}$ es la orientación relativa del sólido 2 con respecto al sólido 1. Generalizando este resultado se obtiene una expresión general para hallar la rotación relativa de un sólido A con respecto a un sólido B :

$$q_{S_A S_B} = q_B^* q_A \quad (5.43)$$

Esta expresión tiene interés a lo hora del cálculo de los ángulos de Euler en la articulación de un brazo articulado si se conocen los cuaterniones de rotación globales de los dos eslabones que forman la articulación.

5.3. Cálculo de la posición del brazo

Una vez sean conocidas las orientaciones de los sensores, sabiendo que dichos sensores se encuentran solidarios al brazo y cuyo eje x se ha orientado en sentido longitudinal de acuerdo a la figura 5.3, resulta muy sencillo obtener las posiciones de cada punto del brazo sabiendo su posición inicial.

Un brazo articulado simple se definirá por un conjunto de 3 longitudes $\{L_1, L_2, L_3\}$ que representan las distancias hombro-codo, codo-muñeca y muñeca-punta de dedo respectivamente, y por 3 cuaterniones $\{q_1, q_2, q_3\}$ que indican el estado de orientación de brazo, antebrazo y mano respectivamente. Dado que para los sensores xsens la posición inicial es aquella en la que su eje x apunta al norte magnético, y el eje z es vertical hacia arriba, se establecerá como posición inicial del brazo aquella en la que el eje longitudinal de todos los eslabones apunta hacia el norte magnético. En esta posición se cumple que:

$$q_1 = q_2 = q_3 = 1$$

Además como el eje x se corresponde con el elemento base i , los puntos iniciales de hombro, codo, muñeca y punta de dedo son:

$$p_0^i = 0 \quad p_1^i = L_1 i \quad p_2^i = p_1^i + L_2 i = (L_1 + L_2) i \quad p_3^i = p_2^i + L_3 i = (L_1 + L_2 + L_3) i$$

Se tomará como origen de coordenadas global la posición del hombro p_0 . Cada eslabón tendrá por origen de coordenadas local la articulación inmediatamente anterior.

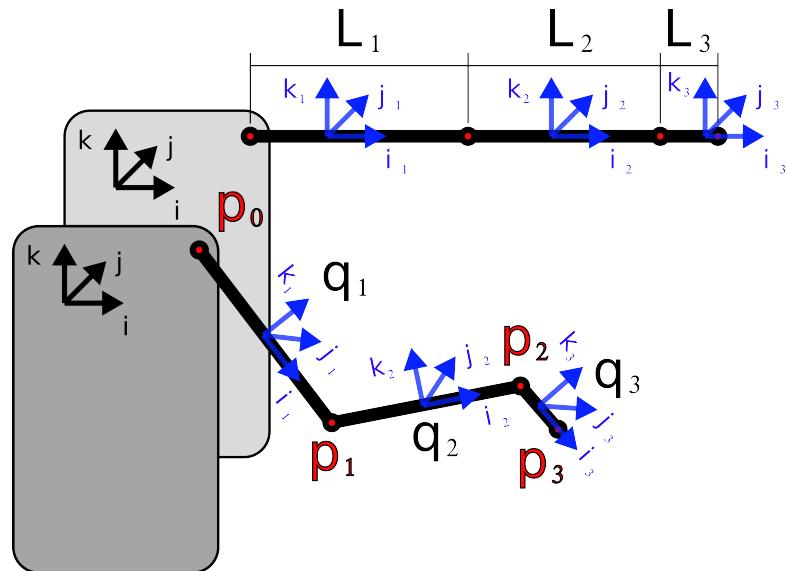


Figura 5.3: Posiciones del brazo inicial y genérica. L_1 , L_2 y L_3 son las longitudes de brazo, antebrazo y mano respectivamente, y q_1 , q_2 y q_3 son sus respectivas orientaciones. p_0 , p_1 , p_2 y p_3 son las posiciones de hombro, codo, muñeca y punta de dedo respectivamente.

Dada una posición genérica del brazo definida por el conjunto de cuaterniones $\{q_i\} = \{q_1, q_2, q_3\}$, las posiciones de cada articulación se pueden calcular de esta forma:

$$\text{Hombro: } p_0 = 0$$

$$\text{Codo: } p_1 = q_1(L_1 i)q_1^*$$

$$\text{Muñeca: } p_2 = p_1 + q_2(L_2 i)q_2^*$$

$$\text{Punta de dedo: } p_3 = p_2 + q_3(L_3 i)q_3^*$$

Además se pueden obtener las posiciones local y global de un punto cualquiera r (no necesariamente articulación) perteneciente a un eslabón K , del que se conoce su posición local inicial r_L^i :

$$r_L = q_K r_L^i q_K^*$$

$$r_G = p_{K-1} + r_L = p_{K-1} + q_K r_L^i q_K^*$$

donde p_{K-1} es la posición de la articulación $K - 1$. Se puede generalizar este resultado a la posición de un punto r cualquiera situado en la articulación K de un brazo articulado, no necesariamente humano, con N eslabones, definido por las longitudes de los eslabones $\{L_i\} = \{L_1, L_2, \dots, L_N\}$ y orientaciones $\{q_i\} = \{q_1, q_2, \dots, q_N\}$:

$$r_G = p_{K-1} + q_K r_L^i q_K^* = p_{K-2} + q_{K-1}(L_{K-1} i)q_{K-1}^* + q_K r_L^i q_K^* = \dots$$

$$r_G = q_K r_L^i q_K^* + \sum_{m=1}^{K-1} q_m(L_m i)q_m^* \quad (5.44)$$

donde r_L^i es la posición local inicial de r dentro del eslabón.

5.4. Obtención de los ángulos de Euler a partir del cuaternión de orientación

En ciertas aplicaciones, como por ejemplo cuando se pretende mover las articulaciones de un brazo robótico, resultan más útiles los ángulos de Euler que el cuaternión o la matriz de rotación. Normalmente los brazos robóticos presentan articulaciones de un grado de libertad en las que la característica que define el estado de la articulación es el ángulo. Se puede definir un estado de rotación mediante tres ángulos –ángulos de Euler– sobre tres los tres ejes principales x, y, z . A estas rotaciones se les denomina rotaciones principales. En este apartado se verá como realizar el cálculo de los ángulos de Euler a partir del cuaternión de rotación.

Para la obtención de los ángulos de Euler se usará la convención para los ángulos $\langle \psi : yaw(Z), \theta : pitch(Y), \phi : roll(X) \rangle$ en un sistema de coordenadas intrínsecas, lo que equivale a $\langle \phi, \theta, \psi \rangle$ en coordenadas extrínsecas. Se supondrá el sistema de coordenadas local situado sobre la superficie de una esfera de radio unitario, con un sistema de coordenadas global, de tal modo que el eje x del sistema de coordenadas local sea siempre normal a la superficie de la esfera y el eje z se tomará por el momento apuntando al norte de la esfera y paralelo a su superficie. Se tomará como origen de coordenadas (O) la posición en que coinciden los sistemas de coordenadas local y global.

Se mueve un punto desde el origen de coordenadas O hasta un punto P , tal como se muestra en la figura 5.4. El punto P tiene por coordenadas:

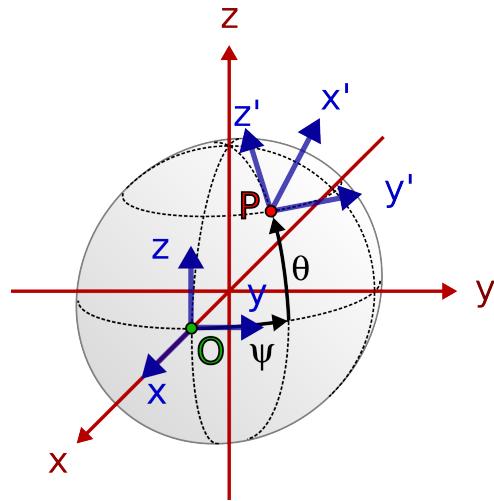


Figura 5.4: Sistema de coordenadas local sobre la esfera

$$P = \cos \psi \cos \theta i + \sin \psi \cos \theta j + \sin \theta k$$

Como la esfera es de radio unitario, resulta obvio que el punto P tiene las mismas coordenadas que el vector i' perteneciente al sistema de coordenadas local, si éste tuviera por origen el centro de la esfera.

$$i' = \cos \psi \cos \theta i + \sin \psi \cos \theta j + \sin \theta k \quad (5.45)$$

Utilizando las componentes del vector i' resulta sencillo obtener los ángulos ψ y θ :

$$\frac{i'_y}{i'_x} = \frac{\sin \psi \cos \theta}{\cos \psi \cos \theta} = \tan \psi \iff \psi = \arctan \left(\frac{i'_y}{i'_x} \right) \quad (5.46)$$

$$i'_z = \sin \theta \iff \theta = \arcsin i'_z \quad (5.47)$$

A continuación se realiza la tercera rotación sobre el eje x' . Los ejes y'' y z'' así obtenidos permanecerán tangentes a la superficie de la esfera, mientras el eje x'' coincidirá con x' . Esta situación final se muestra en la figura 5.5.

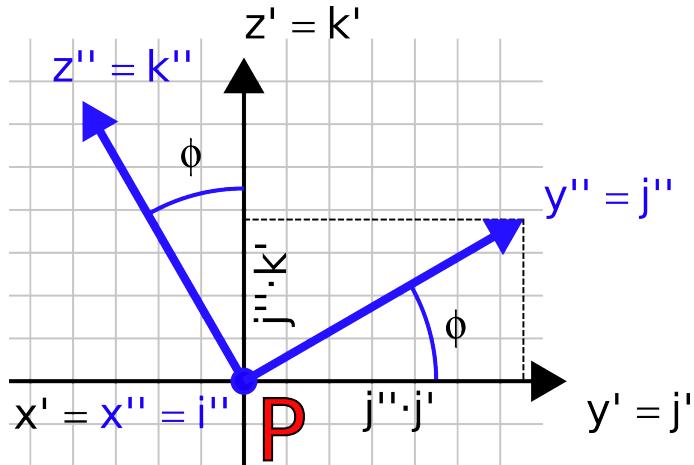


Figura 5.5: Tercera rotación en el sistema de coordenadas local sobre el punto P de la esfera

El estado del sistema de coordenadas tras la tercera rotación es el que va a ser dado por el cuaternion de orientación que nos proporciona el sensor, al que se denominará q_s . Sin embargo, para el cálculo del último de los ángulos (ϕ) se necesitará conocer el hipotético estado anterior a dicha rotación. Sabiendo que el vector i'' es igual a i' , es posible obtener el sistema de coordenadas en el estado intermedio.

El sistema de coordenadas final es el siguiente:

$$\begin{cases} i'' = q_s i q_s^* = i''_x i + i''_y j + i''_z k \\ j'' = q_s j q_s^* = j''_x i + j''_y j + j''_z k \\ k'' = q_s k q_s^* = k''_x i + k''_y j + k''_z k \end{cases}$$

Para el cálculo del conjunto $\{i', j', k'\}$ se parte de que $i' = i''$. Además se sabe que el vector j' será perpendicular a k y a la proyección de i'' sobre el plano xy de la esfera. Esta proyección será la resultante de eliminar la componente en k de i'' :

$$i''_{xy} = i''_x i + i''_y j$$

De la ecuación (5.45) se puede deducir que el módulo de la proyección de este vector será:

$$\|i''_{xy}\| = \sqrt{(\cos \psi \cos \theta)^2 + (\sin \psi \cos \theta)^2} = \cos \theta$$

Para el cálculo de j' será necesario utilizar el vector i''_{xy} normalizado ya que si no se obtendría un vector cuyo módulo no sería unitario:

$$(i''_{xy})_u = \frac{i''_{xy}}{\cos \theta} = \frac{1}{\cos \theta} (i''_x i + i''_y j)$$

Ahora se puede calcular j' :

$$\begin{aligned} j' &= k \times (i''_{xy})_u = \frac{1}{\cos \theta} k \times (i''_x i + i''_y j) \\ j' &= \frac{1}{\cos \theta} (-i''_y i + i''_x j) \end{aligned} \tag{5.48}$$

Finalmente, una vez conocidos i' y j' , es posible obtener k' :

$$k' = i' \times j'$$

En resumen:

$$\begin{cases} i' = i''_x i + i''_y j + i''_z k \\ j' = \frac{1}{\cos \theta} (-i''_y i + i''_x j) \\ k' = i' \times j' \end{cases} \tag{5.49}$$

Finalmente, conociendo el vector j' se puede obtener el último de los ángulos buscados:

$$\tan \phi = \frac{\sin \phi}{\cos \phi} = \frac{j'' \cdot k'}{j'' \cdot j'} \iff \phi = \arctan \frac{j'' \cdot k'}{j'' \cdot j'} \quad (5.50)$$

El algoritmo de cálculo de los ángulos de Euler puede resumirse en los siguientes pasos, conocido el cuaternion de orientación del sensor q_s ³:

Algoritmo de cálculo de los ángulos de Euler a partir del cuaternion de orientación		
1º	Cálculo de i'' y j''	$i' = i'' = q_s i q_s^*$ $j'' = q_s j q_s^*$
2º	Obtención de los ángulos ψ y θ	$\psi = \arctan \left(\frac{i'_y}{i'_x} \right)$ $\theta = \arcsin i'_z$
3º	Cálculo del vector j' y k'	$j' = \frac{1}{\cos \theta} (-i''_y i + i''_x j)$ $k' = i' \times j'$
4º	Obtención del tercer ángulo, ϕ	$\phi = \arctan \frac{j'' \cdot k'}{j'' \cdot j'}$

Cuadro 5.2: Algoritmo de cálculo de los ángulos de Euler a partir del cuaternion de orientación.

5.4.1. Ampliación del intervalo de los ángulos obtenidos

El algoritmo presentado en el cuadro 5.2 da una solución para el ángulo θ dentro del intervalo $(-\frac{\pi}{2}, \frac{\pi}{2})$. Puede pasar que la articulación del brazo robótico cuyo eje se refiera al ángulo θ se mueva en un intervalo más amplio. O que se quieran mover únicamente dos articulaciones mediante un sensor prescindiendo de la tercera rotación, y se necesite que $\phi \approx 0$ ya que de otro modo los otros dos ángulos no se corresponden muy bien con la realidad. por ello resultará de interés poder obtener soluciones de θ que no se restrinjan al intervalo $(-\frac{\pi}{2}, \frac{\pi}{2})$. En los apartados siguientes se dará un par de posibles soluciones a este problema.

Generación de una segunda solución a partir de la primera

Si se amplía el intervalo de θ a $(-\pi, \pi)$, se obtiene que cada posición en la esfera puede representarse de dos formas distintas mediante los ángulos ψ y θ .

En la figura 5.6) se muestran los ángulos de las dos posibles soluciones. Sea una solución $s_1 = \{\psi_1, \theta_1, \phi_1\}$ obtenida mediante el algoritmo descrito en el cuadro 5.2; Se deduce que la segunda solución es:

$$s_2 = \begin{cases} \psi_2 = \begin{cases} \psi_1 + \pi : & \psi_1 < 0 \\ \psi_1 - \pi : & \psi_1 \geq 0 \end{cases} \\ \theta_2 = \begin{cases} -\pi - \theta_1 : & \theta_1 < 0 \\ \pi - \theta_1 : & \theta_1 \geq 0 \end{cases} \\ \phi_2 = \begin{cases} \phi_1 + \pi : & \phi_1 < 0 \\ \phi_1 - \pi : & \phi_1 \geq 0 \end{cases} \end{cases}$$

³A la hora de implementar el algoritmo, se utilizará la función atan2 en lugar de arctan

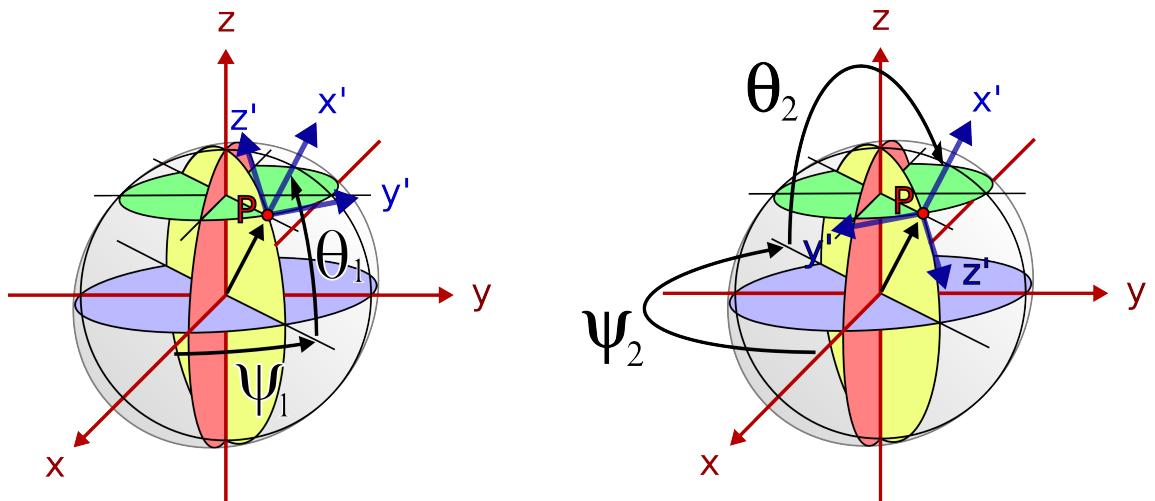


Figura 5.6: Dos soluciones para θ (*pitch*) y ψ (*yaw*)

Para decidir que solución será la buena se deberá establecer un criterio de selección. Por ejemplo, en el caso de querer mover dos articulaciones, se sabe que ϕ tiene que ser aproximadamente 0. En el momento en que la primera solución nos de un valor de ϕ muy diferente de 0, se escogerá la segunda solución.

En la práctica, este método funciona bastante bien siempre y cuando $\theta \approx \frac{\pi}{2}$. En esta zona de la esfera se tiene que $\cos \theta \approx 0$, y a la hora de calcular j' (5.2), el término $\frac{1}{\cos \theta}$ amplifica enormemente el ruido propio del sensor, lo que genera saltos de magnitud apreciable en los valores de los ángulos obtenidos, aún cuando el sensor permanece estático. Esta solución se aplica en el caso del conjunto de dos articulaciones que unen el brazo robótico a la base móvil, ya que aunque sigue habiendo problemas en la zona de *gimbal lock*, ahora es posible asignar a la segunda articulación que forma el par un ángulo mayor de 90°. En el siguiente apartado se verá cómo eliminar del todo el problema, aunque esta solución sólo será posible aplicarla a articulaciones de un solo grado de libertad.

Giro de los ejes locales

Sea una articulación cuyo eje de rotación coincide con el eje y de los dos sensores dispuestos para medir su estado. Es esta situación se deduce de antemano que el vector base i local siempre se localizará en torno a un círculo máximo de la esfera, por lo que habrá zonas en las que nunca se situará. En la figura 5.7 se representa la situación de la esfera unitaria después de realizar un giro de los ejes locales del sensor de 90° alrededor del eje x . Se había comprobado que la zona de *gimbal lock* en el caso del algoritmo utilizado se situaba en las dos zonas de corte del eje z con la esfera. Este giro dado al sistema de coordenadas del sensor evita que su zona de movimiento (representada en verde en la figura) pase por las zonas de *gimbal lock* (en magenta). Esta es la solución aplicada en el caso de las articulaciones del robot que giran alrededor del eje y .

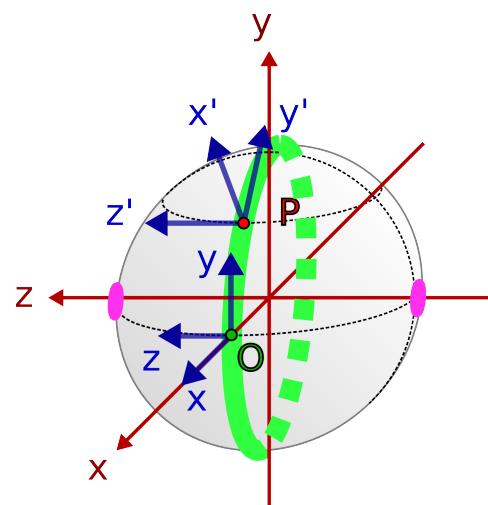


Figura 5.7: Representación en la esfera de un giro de los ejes locales alrededor de x para evitar pasar por las zonas de *gimbal lock* (magenta).

Capítulo 6

IMPLEMENTACIÓN DEL SOFTWARE

En este capítulo se describirá detalladamente el funcionamiento de los programas realizados para el presente proyecto. En concreto, se dará una idea del funcionamiento general de cada programa, las funciones que realizan las clases creadas y cómo se relacionan entre ellas, y se proporcionará una pequeña documentación de los métodos para interactuar con ellas. Dada la extensión resultante del código fuente, se ha optado por no incluir en el presente documento la implementación de las clases y programas, la cual se detalla en el archivo *código_fuente.pdf* y en el propio *stack* que se proporciona en el CD adjunto a este proyecto. Todo el software se ha incluido dentro de un *stack* de ROS –llamado *youbot-xsens-controller*–, de forma que su instalación¹ en otros ordenadores con ROS sea sencilla y rápida.

El stack *youbot-xsens-controller* está compuesto por los siguientes paquetes:

- **xsens_driver**: paquete que contiene el programa *xsens_node*, que actúa de driver para la comunicación sensor/máster - PC.
- **dfv**: librería para el manejo de cuaterniones, vectores y matrices. En ella se implementan las respectivas clases y métodos para operar con estas estructuras.
- **youbot_controller**: programa que realiza el control de las articulaciones del robot *YouBot*, o de la simulación del brazo robótico del mismo, utilizando los datos de los sensores *xsens*.
- **arm_visualizer**: programa que se encarga de gestionar la visualización en el simulador Gazebo del modelo de un brazo humano.

Para la creación de los distintos programas se han utilizado las herramientas que proporciona ROS para la compilación de programas y gestión de paquetes. En la figura 6.1 se presenta un esquema simplificado de las comunicaciones entre los distintos programas que forman el sistema.

¹Se describirá el proceso de instalación en el anexo *INSTALACIÓN DEL SOFTWARE*

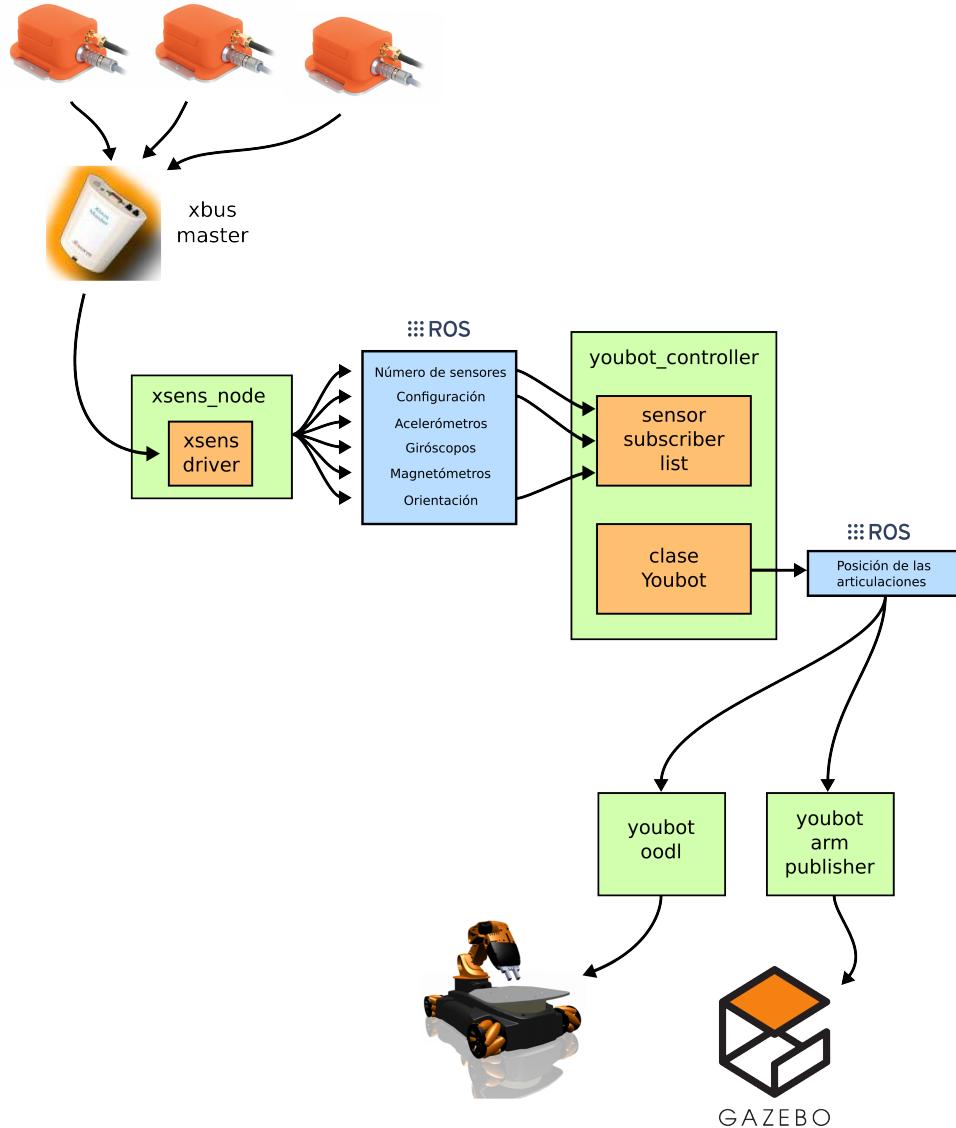


Figura 6.1: Esquema simplificado de las comunicaciones entre programas. En azul se representan los topics y parámetros de ROS, en verde los nodos y en naranja las clases principales.

6.1. Librería para manejo de cuaterniones, vectores y matrices

Dentro del paquete *dfv* se ha creado una librería que contiene las clases *dfv::Quaternion*, *dfv::Vector3* y *dfv::Matrix*, en las que se realiza una implementación de los algoritmos y fórmulas vistos en el capítulo *CÁLCULO DE LAS POSICIONES Y ORIENTACIONES DE LOS SENSORES*. Estas clases son utilizadas por el resto de paquetes del stack para realizar los cálculos correspondientes.

6.1.1. La clase *dfv::Quaternion*

La clase *dfv::Quaternion* es la implementación del concepto de cuaternion y las operaciones que se pueden realizar con él. A continuación se muestra una breve documentación de las operaciones más importantes que se pueden realizar con esta clase.

▪ **Constructores:**

- **Quaternion():** Constructor sin argumentos. Crea un cuaternión cuyas cuatro componentes son nulas.
- **Quaternion(double w_, double x_, double y_, double z_):** Constructor del cuaternión genérico (ecuación (5.1)). Crea un cuaternión cuyas componentes son los valores que se le pasan a modo de argumento.
- **explicit Quaternion(const Vector3& v):** Constructor del cuaternión asociado al vector v que se le pasa como parámetro (ecuación (5.17)).

▪ **Operador de asignación:**

- **Quaternion& operator=(const Quaternion& q):** realiza una copia del cuaternión que se le pasa como parámetro.

▪ **Operadores de asignación compuestos:**

- **Quaternion& operator+=(const Quaternion& q):** Suma al cuaternión original el cuaternión que se le pasa como parámetro.
- **Quaternion& operator-=(const Quaternion& q):** Resta al cuaternión original el cuaternión que se le pasa como parámetro.
- **Quaternion& operator*=(const double k):** Multiplica el cuaternión original por una constante k .

▪ **Operadores aritméticos binarios:**

- **const Quaternion operator+(const Quaternion& q) const:** devuelve la suma de dos cuaterniones.
- **const Quaternion operator-(const Quaternion& q) const:** devuelve la diferencia entre dos cuaterniones
- **friend const Quaternion operator*(double k, Quaternion& q):** devuelve la multiplicación de una constante por el cuaternión que se le pase como argumento.
- **const Quaternion operator*(double k) const:** misma operación de multiplicación por una constante, pero pudiéndose especificar en orden inverso.
- **const Quaternion operator*(const Quaternion& q) const:** devuelve el producto de Hamilton de dos cuaterniones

▪ **Operadores de comparación:**

- **bool operator==(const Quaternion& q) const:** devuelve *true* si las componentes de los dos cuaterniones son iguales, y *false* en caso contrario.
- **bool operator!=(const Quaternion& q) const:** devuelve *true* si los dos cuaterniones son diferentes.
- **double GetModulus() const:** devuelve el módulo del cuaternión (ecuación (5.13)).

- **void Normalize()**: realiza un normalización del cuaternión, esto es, multiplica el cuaternión por el inverso de su módulo, de forma que se obtiene un cuaternión de magnitud unitaria.
- **const Quaternion GetConjugate() const**: devuelve el conjugado del cuaternión (ecuación (5.12)).
- **static const Quaternion GetRotationQuaternion(...)**: función para construir un cuaternión a partir de los argumentos especificados.
 - Construcción a partir de las componentes del eje de rotación y ángulo de rotación.
 - Construcción a partir del vector de rotación y ángulo de rotación.
 - Construcción del cuaternión que define la rotación de un vector, especificando el vector antes y después de ser rotado.
 - Construcción del cuaternión que define la rotación de un sistema de dos vectores antes y después de ser rotados. Se supone que los dos vectores son perpendiculares entre si.
- **void GetAxisAndAngle(Vector3& vector, double& angle) const**: función que devuelve el vector y ángulo de rotación.
- **void GetRPY(double& roll, double& pitch, double& yaw, unsigned int solution = 1)**: implementación del algoritmo descrito en la tabla 5.2 para la obtención de los ángulos de Euler a partir del cuaternión de rotación.
- **static const Quaternion GetDifference(const Quaternion& q1, const Quaternion& q2)**: diferencia entre dos cuaterniones (no confundir con resta). Devuelve el cuaternión de rotación relativo entre los cuaterniones q_1 y q_2 (ecuación (5.43)).

6.1.2. La clase *dfv::Vector3*

La clase *dfv::Vector3* es la implementación de un vector de 3 componentes para la representación de puntos en un espacio euclídeo \mathbb{R}^3 .

- **Constructores:**
 - **Vector3()**: Constructor sin argumentos. Crea un vector cuyas tres componentes son nulas.
 - **Vector3(double x_, double y_, double z_)**: Constructor de un vector genérico cuyas componentes son los valores que se le pasan a modo de argumento.
 - **explicit Vector3(const Quaternion& q)**: Constructor del vector asociado al cuaternión q que se le pasa como parámetro. Se obvia la componente w del cuaternión. (ecuación (5.17))
- **Operador de asignación:**
 - **Vector3& operator=(const Vector3& v)**: realiza una copia del vector que se le pasa como parámetro.

- Operadores de asignación compuestos:

- **Vector3& operator+=(const Vector3& v)**: suma al vector original el vector que se le pasa como parámetro.
- **Vector3& operator-=(const Vector3& v)**: resta al vector original el vector que se le pasa como parámetro.
- **Vector3& operator*=(const double k)**: multiplica el vector original por una constante k .

- Operadores aritméticos binarios:

- **const Vector3 operator+(const Vector3& v) const**: devuelve la suma de dos vectores.
- **const Vector3 operator-(const Vector3& v) const**: devuelve la resta de dos vectores.
- **friend const Vector3 operator*(double k, Vector3& v)**: devuelve la multiplicación de un escalar por un vector.
- **const Vector3 operator*(double k) const**: devuelve la multiplicación de un vector por un escalar. Es una operación idéntica a la anterior, pero especificada en orden inverso.
- **const double operator*(const Vector3& v) const**: devuelve el producto escalar de dos vectores.
- **const Vector3 operator^(const Vector3& v) const**: devuelve el vector resultado de realizar el producto vectorial de dos vectores.

- Operadores de comparación:

- **bool operator==(const Vector3& v) const**: devuelve *true* si los dos vectores son iguales, esto es, si sus componentes también lo son, y *false* en caso contrario.
- **bool operator!=(const Vector3& v) const**: devuelve *true* si los dos vectores son diferentes, y *false* en caso contrario.
- **double GetMagnitude() const**: devuelve la longitud del vector.
- **void Normalize()**: realiza una normalización del vector, multiplicándolo por la inversa de su longitud para obtener un vector de longitud unitaria.
- **const Vector3 GetNormalized() const**: devuelve el vector normalizado sin modificar el original.
- **const Vector3 GetScaled(double k) const**: devuelve el vector escalado por una constante. Es la misma operación que la multiplicación del vector por un escalar.
- **Vector3& Rotate(const Quaternion q)**: realiza una rotación del vector definida por el cuaternion q . Ecuación (5.16).
- **const Vector3 GetRotated(const Quaternion& q) const**: Devuelve el vector rotado sin modificar el original.

6.1.3. La clase `dfv::Matrix`

En la clase `dfv::Matrix` se realiza una implementación del concepto de matriz. Las operaciones más importantes que se pueden realizar son las siguientes:

- **Constructores:**

- **`Matrix()`:** Constructor sin argumentos. Crea una matriz inválida de 0 filas y 0 columnas.
- **`Matrix(unsigned int size)`:** Constructor de una matriz cuadrada de tamaño `size`, cuyos elementos son todos 0.
- **`Matrix(unsigned int rows, unsigned int columns)`:** Constructor de una matriz de `rows` filas y `columns` columnas cuyos elementos son todos 0.

- **Operador de asignación:**

- **`Matrix& operator=(const Matrix& q)`:** realiza una copia de la matriz que se le pasa como parámetro.

- **Operadores de asignación compuestos:**

- **`Matrix& operator+=(const Matrix& q)`:** suma a la matriz original la matriz `q`.
- **`Matrix& operator-=(const Matrix& q)`:** resta a la matriz original la matriz `q`.
- **`Matrix& operator*=(const double k)`:** multiplica la matriz por un escalar `k`, esto es, realiza una multiplicación de cada uno de sus elementos.

- **Operadores aritméticos binarios:**

- **`const Matrix operator+(const Matrix& q) const`:** definición del operador "+" que realiza la suma de dos matrices.
- **`const Matrix operator-(const Matrix& q) const`:** definición del operador "-", resta de dos matrices.
- **`friend const Matrix operator*(double k, Matrix& q)`:** multiplicación de un escalar por una matriz (multiplicación de cada elemento de la matriz por dicho escalar).
- **`const Matrix operator*(double k) const`:** multiplicación de una matriz por un escalar.
- **`const Matrix operator*(const Matrix& q) const`:** implementación de la multiplicación de dos matrices.

- **Operadores de comparación:**

- **`bool operator==(const Matrix& v) const`:** devuelve *true* si las matrices son del mismo tamaño y sus elementos son iguales, y *false* en caso contrario.
- **`bool operator!=(const Matrix& v) const`:** operador cuyo resultado es el inverso del operador "==".

- **Matrix& Create(unsigned int rows, unsigned int columns, double value = 0):** crea una matriz de *rows* filas y *columns* columnas a cuyos elementos se les asigna el valor *value* pasado como parámetro.
- **double Get(unsigned int row, unsigned int col) const:** devuelve el valor situado en la fila *row* y columna *col*.
- **void Set(unsigned int row, unsigned int col, double value):** asigna al elemento situado en la fila *row* y columna *col* el valor *value*.
- **unsigned int GetRows() const:** devuelve el número de filas de la matriz.
- **unsigned int GetColumns() const:** devuelve el número de columnas de la matriz.
- **Matrix GetMinor(unsigned int row, unsigned int column) const:** devuelve el menor de la matriz asociado a la fila *row* y columna *col*.
- **void Randomize():** asigna a cada elemento de la matriz un valor aleatorio entre 0 y 1.
- **double GetDeterminant() const:** devuelve el determinante de la matriz.
- **const Matrix GetTransposed() const:** devuelve la traspuesta de la matriz.
- **const Matrix GetAdjoint() const:** devuelve el adjunto de la matriz.
- **const Matrix GetAdjugate() const:** devuelve la matriz traspuesta conjugada.
- **const Matrix GetInverse() const:** devuelve la inversa de la matriz.
- **double operator()(unsigned int row, unsigned int column) const:** operador equivalente a la función *Get()*.

6.1.4. Cómo usar la librería dfv en otro paquete de ROS

Si se quisiera hacer uso de la librería *dfv* dentro de otro paquete de ROS, se deberán seguir los siguientes pasos:

1. Añadir como dependencia en el archivo *manifest.xml* el paquete *dfv*. Como ejemplo se muestra el archivo *manifest.xml* del paquete *youbot_controller*:

```

1 <package>
2   <description brief="youbot_controller">
3     youbot_controller
4
5   </description>
6   <author>Daniel</author>
7   <license>BSD</license>
8   <review status="unreviewed" notes="" />
9   <url>http://ros.org/wiki/youbot\_controller</url>
10  <depend package="std_msgs"/>
11  <depend package="roscpp"/>
```

```

14 <depend package="xsens_driver"/>
15 <depend package="brics_actuator"/>
16 <depend package="dfv"/>
17
18 </package>

```

2. Enlazar la librería *dfv* en el archivo *CMakeLists.txt*:

```

1 target_link_libraries(nombre_del_ejecutable dfv)

```

3. Incluir el archivo de cabecera *dfv.h* en el programa donde se quiera utilizar la librería:

```

1 #include <dfv/dfv.h>

```

Ahora se pueden utilizar las clases *dfv::Quaternion*, *dfv::Vector3* y *dfv::Matrix* y otras utilidades de la librería en el nuevo programa.

En el archivo *example.cpp* dentro de la carpeta *dfv/examples/* hay un pequeño ejemplo de uso de la librería *dfv*.

6.2. Driver para la comunicación xbus Master/xsens - PC

Uno de los problemas más importantes a resolver en este proyecto es conseguir una comunicación fluida de los datos de los sensores hacia el PC. La comunicación entre un dispositivo y el ordenador se realiza mediante un programa llamado *driver*, que actúa de interfaz entre el dispositivo físico y el sistema operativo del PC, utilizando algún tipo de canal de datos (USB, bluetooth, etc.). En el caso del driver para los sensores xsens, la comunicación se realizará a través del puerto USB.

En la documentación del *Kit de Desarrollo de Software* de *xsens* vienen proporcionadas algunas clases y estructuras en lenguaje C++ para la comunicación de bajo nivel con el sensor/máster. Para la realización del driver se han aprovechado estas clases y se han encapsulado dentro de una clase *wrapper*, llamada *xsens::Driver*, que proporciona una interfaz sencilla hacia las clases de bajo nivel. El programa principal del *driver* realiza la configuración de los sensores, y lee los datos a través del *wrapper*, publicándolos en *topics* de ROS.

Los archivos de la documentación que se han incluido en el driver son los siguientes (incluyen los archivos de extensión .cpp y .h):

- **cmt1**: implementa las comunicaciones de bajo nivel con el puerto serie.
- **cmt2**: interfaz para mensajes y ficheros de registro.
- **cmt3**: capa de más alto nivel para las comunicaciones con el sensor. Proporciona una interfaz independiente del sistema operativo.

- **cmtdef**: definiciones de constantes y tipos de datos.
- **cmtmessage**: gestión de mensajes.
- **cmtpacket**: gestión de los paquetes de información en los que se basa la comunicación entre el sensor y el PC.
- **cmtscan**: funciones para el escaneo de los puertos del PC.
- **pstdint**: una versión portable de la librería stdint, que incluye definiciones precisas de tipos de datos enteros.
- **xsens_exception**: definición de la clase Exception usada por el driver.
- **xsens_fifoqueue**: contiene la implementación de una cola FIFO.
- **xsens_file**: definición de algunos tipos de datos.
- **xsens_janitors**: implementación de distintos tipos de *janitors*, que realizan funciones de limpieza de objetos al perder su *scope*.
- **xsens_list**: implementación de una clase lista utilizada en el driver.
- **xsens_std**: definición de los valores de retorno de las funciones del driver.
- **xsens_time**: implementación de funciones y constantes para la gestión del tiempo.

En la siguiente lista se muestran los archivos creados en este paquete y la función que realizan dentro del driver:

- **xsens_driver**: implementación de la clase *xsens::Driver*, la cual básicamente realiza una encapsulación de todas las clases que se incluyen en los archivos mencionados en la lista anterior y actúa de interfaz con el nodo principal *xsens-node*.
- **xsens_sensor**: la clase *xsens::Sensor* es en esencia una estructura de datos donde se almacena la configuración de cada uno de los sensores conectados y los datos leídos en cada ciclo de lectura.
- **xsens_node**: programa principal del driver. Se encarga de publicar en ROS los datos leídos a través de la clase *xsens::Driver*.
- **xsens_sensor_subscriber**: no pertenece al driver propiamente dicho, sino que este archivo es parte de la implementación de la librería *xsens_driver* que puede utilizarse como interfaz de lectura en otros programas de los datos publicados por el driver.

A continuación se realiza una descripción en detalle de la implementación y funcionamiento del driver.

6.2.1. Funcionamiento de la clase xsens::Driver

La clase *xsens::Driver* actúa de interfaz hacia las clases de más bajo nivel encargadas de la comunicación del puerto serie. El objetivo perseguido con su creación es encapsular y automatizar la configuración de los sensores y la lectura de datos, de forma que no sea necesario realizar todo este proceso manualmente. Resumiendo brevemente, las funciones que realiza esta clase son:

1. Realizar un escaneo de los puertos USB del PC con el objetivo de detectar tipo y número de sensores conectados.
2. Configurar los sensores en el modo deseado (datos que devolverá, forma de representar las orientaciones, etc.).
3. Establecer los sensores en modo de medición.
4. Actualizar los datos miembro cada vez que llega una nueva medición de los sensores.

Esta clase contiene una serie de métodos miembro que permiten interactuar con ella. En la siguiente lista se muestran los de mayor utilidad:

- Métodos para establecer la configuración del sensor:

- **void SetOutputMode(CmtOutputMode output_mode)**

Permite establecer el modo de salida del sensor, el cual define el tipo de datos que devolverá. Acepta un parámetro de tipo *CmtOutputMode*, que puede ser combinación de varios de los siguientes valores:

- CMT_OUTPUTMODE_RAW: datos crudos de los acelerómetros, giróscopos y magnetómetros sin calibrar.
- CMT_OUTPUTMODE_TEMP: temperatura registrada por el sensor.
- CMT_OUTPUTMODE_CALIB: datos calibrados de los acelerómetros, giróscopos y magnetómetros.
- CMT_OUTPUTMODE_ORIENT: orientación del sensor.

Por ejemplo, para configurar el sensor de modo que devuelva los datos calibrados, la orientación y la temperatura se llamará a la función de la siguiente forma:

```
1 driver.SetOutputMode(CMT_OUTPUTMODE_CALIB | CMT_OUTPUTMODE_ORIENT |  
CMT_OUTPUTMODE_TEMP);
```

- **void SetOutputSettings(CmtOutputSettings output_settings)**

Permite establecer la forma de representación de la orientación del sensor. Acepta un parámetro de tipo *CmtOutputSettings*, que puede ser uno de los siguientes valores:

- CMT_OUTPUTSETTINGS_ORIENTMODE_EULER: Orientación del sensor en forma de ángulos de Euler
- CMT_OUTPUTSETTINGS_ORIENTMODE_MATRIX: Matriz de rotación.
- CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION: Cuaternión de orientación.

Para configurar el sensor de modo que devuelva su orientación forma de cuaternion, se pasará a la función el siguiente parámetro:

```
1 driver.SetOutputSettings(CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION);
```

- **void SetAlignmentMatrix(unsigned int sensor_index, CmtMatrix alignment_matrix)**

Establece una matriz de rotación inicial al sensor cuyo identificador se le pase al parámetro *sensor_index* *sensor_index*.

Para establecer una matriz de rotación inicial identidad al sensor de id 0:

```
1 driver.SetAlignmentMatrix(0,
    xsens::DfvToCmtMatrix(dfv::Matrix::Identity(3)));
```

- **bool Initialize()**

Se llamará a esta función después de haber establecido los parámetros del sensor mediante las funciones descritas anteriormente. Esta función pone al sensor en modo configuración, lo prepara para que devuelva los datos deseados y lo pone en modo de medida.

- Métodos para leer datos del sensor:

- **unsigned int GetMtCount()**

Devuelve el número de dispositivos xsens conectados al PC, excluyendo el máster xbus.

- **CmtOutputMode GetOutputMode() const**

Devuelve el modo de salida de los sensores. Los valores de retorno son los mismos que los que puede aceptar el parámetro de la función *SetOutputMode()* descrita anteriormente.

- **CmtOutputSettings GetOutputSettings() const**

Devuelve la forma de representación de la orientación de los sensores. Los valores de retorno son los mismos que los que puede aceptar el parámetro de la función *SetOutputSettings()* descrita anteriormente.

- **bool SpinOnce()**

Esta función realiza un ciclo en la toma de datos del sensor. Se queda esperando a que llegue un mensaje procedente del sensor, y a continuación actualiza todas las estructuras de datos con los nuevos valores obtenidos. Se ha de llamar a esta función antes de ejecutar alguna de las funciones descritas a continuación. Devuelve *true* si todos los datos fueron leídos de forma correcta, y *false* en caso contrario.

- **CmtQuat& GetOriQuat(int mt_index = 0)**

Devuelve el cuaternion de orientación del sensor con número de identificación especificado. Si se deja el parámetro en blanco, devuelve el cuaternion de orientación del sensor de id 0.

- **CmtMatrix& GetOriMatrix(int mt_index = 0)**

Devuelve la matriz de orientación del sensor con número de identificación especificado. Si se deja el parámetro en blanco, devuelve la matriz de orientación del sensor de id 0.

- **CmtEuler& GetOriEuler(int mt_index = 0)**
Devuelve los ángulos de Euler del sensor con número de identificación especificado. Si se deja el parámetro en blanco, devuelve los ángulos de Euler del sensor de id 0.
- **CmtRawData& GetRawData(int mt_index = 0)**
Devuelve los datos crudos de los acelerómetros, giróscopos y magnetómetros, además de la temperatura del sensor con número de identificación especificado. Si se deja el parámetro en blanco, devuelve los datos crudos del sensor de id 0. Los datos crudos contienen los datos de salida de los conversores analógico-digitales de los sensores, y tienen la forma de enteros de 16 bits. Los datos de los acelerómetros, giróscopos y magnetómetros están compuestos cada uno de un vector de 3 enteros de 16 bits (ejes XYZ), y la temperatura es otro entero de 16 bits.
- **CmtCalData& GetCalData(int mt_index = 0)**
Devuelve los datos calibrados de los acelerómetros, giróscopos y magnetómetros del sensor con número de identificación especificado. Si se deja el parámetro en blanco, devuelve los datos calibrados del sensor de id 0.

Datos calibrados de los sensores		
Dato	Tipo	Unidades
acelerómetros	array de 3 floats	$\frac{m}{s^2}$
giróscopos	array de 3 floats	rad
magnetómetros	array de 3 floats	$\frac{s}{mGauss}$
temperatura	float	$^{\circ}C$

Cuadro 6.1: Unidades de los datos calibrados devueltos por los sensores.

A continuación se presenta un ejemplo mínimo de cómo utilizar la clase xsens::Driver para tomar datos de un sensor:

```

1 #include <xSENS_driver/xsens_driver.h>
2 #include <dfv/dfv.h>
3 #include <xSENS_driver/utils.h>
4
5 int main(int argc, char** argv)
6 {
7     xsens::Driver driver;
8
9     // Configuración del sensor
10    driver.SetOutputMode(CMT_OUTPUTMODE_CALIB | CMT_OUTPUTMODE_ORIENT |
11                           CMT_OUTPUTMODE_TEMP);
12    driver.SetOutputSettings(CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION);
13    driver.SetAlignmentMatrix(0,
14                             xsens::DfvToCmtMatrix(dfv::Matrix::Identity(3)));
15    driver.Initialize();
16
// Toma de datos
while(driver.SpinOnce())

```

```

17     {
18         // Código para lectura y procesamiento de datos
19         // ...
20     }
21
22     return 0;
23 }
```

6.2.2. Programa principal: xsens_node

El nodo *xsens_node* es el encargado de publicar en topics de ROS los datos leídos de los sensores. Para la lectura de los datos del sensor hace uso de la clase `xsens::Driver` descrita en el apartado anterior. Podría considerarse que este programa actúa de interfaz entre dicha clase y ROS. Este programa realiza las siguientes tareas:

1. Detección del número de sensores conectados al PC.
2. Configuración de los sensores:
 - Modo de salida: datos calibrados y orientación.
 - Forma de representación de la orientación: cuaternión.
 - Matriz de rotación inicial identidad.
3. Inicialización y creación de los topics y parámetros de ROS. En concreto se crean los topics y parámetros descritos en los cuadros 6.2 y 6.3 respectivamente.
4. Lectura de los datos del sensor y publicación en los topics de ROS correspondientes.

Topics		
Nombre	Tipo	Descripción
/xsens_node/sensorX/acc	geometry_msgs::Vector3Stamped	Acelerómetros calibrados
/xsens_node/sensorX/gyr	geometry_msgs::Vector3Stamped	Giróscopos calibrados
/xsens_node/sensorX/mag	geometry_msgs::Vector3Stamped	Magnetómetros calibrados
/xsens_node/sensorX/raw_acc	geometry_msgs::Vector3Stamped	Acelerómetros crudos
/xsens_node/sensorX/raw_gyr	geometry_msgs::Vector3Stamped	Giróscopos crudos
/xsens_node/sensorX/raw_mag	geometry_msgs::Vector3Stamped	Magnetómetros crudos
/xsens_node/sensorX/ori_quat	geometry_msgs::QuaternionStamped	Cuaternión de orientación
/xsens_node/sensorX/ori_matrix	std_msgs::Float64MultiArray	Matriz de orientación
/xsens_node/sensorX/ori_euler	std_msgs::Float64MultiArray	Ángulos de Euler

Cuadro 6.2: Topics publicados por el programa *xsens_node*. La X en *sensorX* representa el número de identificación del sensor. Se crea un topic de cada tipo por cada sensor conectado al máster. Los topics creados también dependen de la configuración de los sensores. Por ejemplo, si los sensores son configurados para que den sólo los datos calibrados, no se crearán los topics de orientación.

Parámetros		
Nombre	Tipo	Descripción
/xsens_node/sensor_count	entero	Número de sensores
/xsens_node/output_mode	entero	Modo de salida
/xsens_node/output_settings	entero	Forma de representación de la orientación

Cuadro 6.3: Parámetros publicados por el programa *xsens_node*.

6.2.3. Cómo tomar datos del sensor desde otro programa

Junto con el paquete *xsens_driver* se incluye además una librería con las clases *xsens::SensorSubscriber* y *xsens::SensorSubscriberList*. Estas clases proporcionan una forma muy sencilla de acceder a los topics publicados por el programa *xсens_node*. Al declarar un objeto del tipo *xsens::SensorSubscriberList*, automáticamente dicho objeto se autoconfigura leyendo los parámetros de ROS con la información de configuración de los sensores (cuadro 6.3) e inmediatamente se pone a leer los datos de los topics correspondientes (cuadro 6.2 a excepción de los datos en crudo), sin tener que hacer nada más que pasarle como parámetro el *handle* del nodo en que sea declarado.

Un objeto de clase *xsens::SensorSubscriberList* proporciona los siguientes métodos para lectura de datos del sensor:

- **unsigned int GetMtCount() const**
Devuelve el número de sensores xsens detectados.
- **const dfv::Vector3 GetAcc(unsigned int mt_index) const**
Devuelve un vector con los datos de los acelerómetros del sensor de número de identificación que se le pase como parámetro.
- **const dfv::Vector3 GetGyr(unsigned int mt_index) const**
Devuelve un vector con los datos de los giróscopos del sensor de número de identificación que se le pase como parámetro.
- **const dfv::Vector3 GetMag(unsigned int mt_index) const**
Devuelve un vector con los datos de los magnetómetros del sensor de número de identificación que se le pase como parámetro.

Para incluir la librería *xsens_driver* en otro proyecto se tendrá que incluir la siguiente línea al final del archivo *CMakeLists.txt* del proyecto de ROS:

```
1 target_link_libraries(nombre_del_ejecutable dfv xsens_driver)
```

Además se deberá incluir los paquetes *dfv* y *xsens_driver* en el archivo *manifest.xml*:

```
1 <package>
2   <description brief="mi_paquete">
3
4     mi_paquete
5
6   </description>
7   <author>Daniel</author>
```

```

8 <license>BSD</license>
9 <review status="unreviewed" notes="" />
10 <url>http://ros.org/wiki/mi_paquete</url>
11 <depend package="std_msgs"/>
12 <depend package="roscpp"/>
13 <depend package="dfv"/>
14 <depend package="xsens_driver"/>
15
16 </package>
```

En el siguiente cuadro se presenta un ejemplo de como usar la clase *xsens::SensorSubscriberList* para tomar datos de una red de sensores xsens. Este esquema es el usado en el programa *youbot_controller*:

```

1 #include <xSENS_driver/xsens_sensor_subscriber.h>
2
3 int main(int argc, char** argv)
4 {
5     // Inicialización de ROS
6     ros::init(argc, argv, "xsens_sensr_subscriber_node");
7     ros::NodeHandle node_handle;
8
9     // Declaracion del objeto sensor_subscriber_list
10    // Se autoconfigura de acuerdo a los parámetros detectados
11    // e inmediatamente comienza la lectura de los datos
12    xsens::SensorSubscriberList sensor_subscriber_list(node_handle);
13
14    while(ros::ok())
15    {
16        // Código para lectura y procesamiento de datos de
17        // sensor_subscriber_list
18        // ...
19
20        // Bucle a 10 Hz
21        ros::spinOnce();
22        ros::Duration(0.1).sleep();
23    }
}
```

6.3. Controlador del brazo robótico del robot Youbot real

En el paquete *youbot_controller* se implementa un controlador de las articulaciones del brazo de un robot Youbot. Este programa se ha de ejecutar a la vez que los drivers *xsens_node* y *youbot_oodl*, para poder tomar los datos de los sensores xsens y transmitir las órdenes al robot. Este controlador realiza las tareas resumidas a continuación:

1. Detección de número y configuración de los sensores xsens a través de los topics y parámetros publicados por el driver *xsens_node*. Si no hay 3 sensores conectados se

termina el programa imprimiendo en pantalla una advertencia de que son necesarios 3 sensores para que el programa funcione correctamente.

2. Si detecta que todo está correcto, inmediatamente el programa comienza a leer los cuaterniones de orientación de los tres sensores. Con los tres cuaterniones, el programa halla los cuaterniones relativos entre cada sensor, y con éstos obtiene los ángulos de Euler de la articulación correspondiente². En la figura 6.2 se muestran las articulaciones del brazo y su identificación.

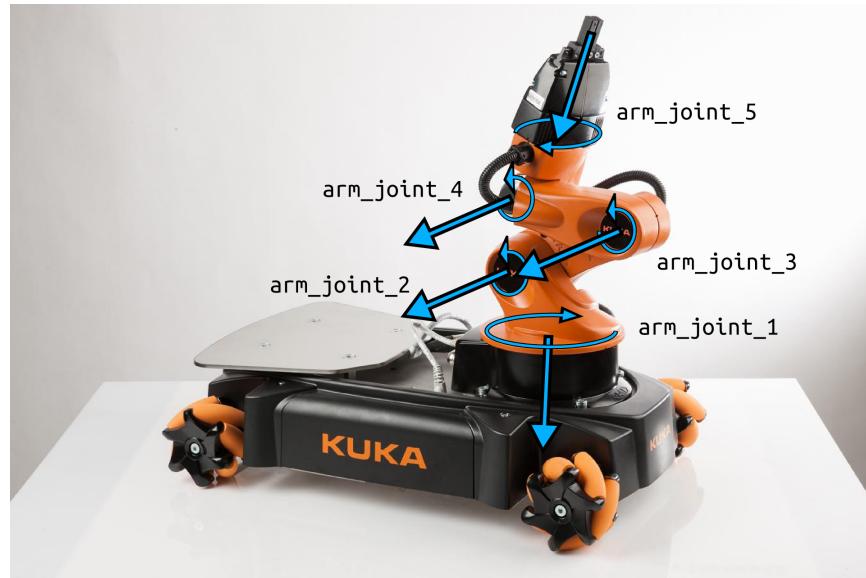


Figura 6.2: Articulaciones del brazo robótico del YouBot y su identificación.

3. A continuación se asignan determinados ángulos obtenidos a las articulaciones correspondientes del robot.
4. Finalmente se publica el mensaje en el correspondiente topic de ROS con las posiciones de las articulaciones del robot, para que sea leído por el driver *youbot_oodl*.

6.3.1. El driver *youbot_oodl*

El driver *youbot_oodl* es un programa que crea una capa sobre la API de YouBot con el objetivo de poder realizar el control del robot mediante el envío de mensajes a ciertos topics ROS. En el cuadro 6.4 se muestran el nombre del topic y tipo de mensaje que se pueden utilizar para controlar el robot.

6.3.2. La clase Youbot

Con un objetivo similar al de la creación de la clase `xsens::SensorSubscriberList`, la clase Youbot tiene como fin encapsular las comunicaciones de ROS entre el programa *youbot_controller* y el driver *youbot_oodl* para que no interfieran en el desarrollo del programa. Esta clase es la encargada de gestionar la publicación de mensajes en los topics de ROS.

²En el capítulo CÁLCULO DE LAS POSICIONES Y ORIENTACIONES DE LOS SENSORES se muestra una explicación detallada de los cálculos matemáticos utilizadas para realizar esta tarea.

Topics a los que está suscrito el driver youbot_oodl		
Nombre (espacio /arm_1)	Tipo	Descripción
/arm_controller/position_command	brics_actuator::JointPositions	Comando con la posición objetivo de las articulaciones
/arm_controller/velocity_command	brics_actuator::JointVelocities	Comando con la velocidad objetivo de las articulaciones
/gripper_controller/position_command	brics_actuator::JointPositions	Comando con la posición objetivo del gripper

Cuadro 6.4: Topics a los que está suscrito el driver youbot_oodl de los que lee la posición y velocidad deseadas para cada articulación.

que utiliza el driver *youbot_oodl* (cuadro 6.4).

Esta clase tiene un funcionamiento muy sencillo. Solamente es necesario pasarle el *handle* del nodo en el que se declare la clase. Para modificar los valores objetivo de las articulaciones y el *gripper* hay que modificar las variables miembro *joint_positions[]* y *gripper_positions[]*. Con el método *PublishMessage()*, se publica en los topics correspondientes los mensajes con las posiciones de las articulaciones. En el siguiente ejemplo se muestra un esquema básico de cómo trabajar con la clase:

```

1 #include <youbot_controller/youbot.h>
2
3 int main(int argc, char** argv)
4 {
5     // inicialización de ROS
6     ros::init(argc, argv, "youbot_example");
7     ros::NodeHandle node_handle;
8
9     // Declaración de un objeto Youbot
10    Youbot youbot(node_handle);
11
12    while(ros::ok())
13    {
14        // Asignación de los valores objetivo de cada articulación
15        youbot.joint_positions[0] = 3.141592;
16        youbot.joint_positions[1] = 2.718182;
17        youbot.joint_positions[2] = -1.618033;
18        youbot.joint_positions[3] = 1.414213;
19        youbot.joint_positions[4] = 0.577215;
20
21        // Publicación del mensaje
22        youbot.PublishMessage();
23
24        ros::spinOnce();
25        ros::Duration(0.1).sleep();
26    }
27

```

```

28     return 0;
29 }
```

La función *PublishMessage()* admite un parámetro de tipo *bool*, que si es *false* (valor por defecto) no publica el mensaje correspondiente a la posición del *gripper*. Si se quisiera controlar el *gripper*, habría que pasar *true* como argumento, pero hay que tener en cuenta que no es posible enviar este mensaje con la misma frecuencia que el mensaje de las articulaciones debido a que el robot no parece poder procesar el mensaje del *gripper* con la suficiente rapidez y se producen comportamientos extraños del robot. De diversas pruebas se ha deducido que la frecuencia más alta de publicación del mensaje del *gripper* para no producir problemas es de 1 Hz.

6.4. Visualizador de la posición del brazo

El paquete *arm_visualizer* proporciona una forma de poder visualizar la posición del brazo humano en el simulador 3D Gazebo. Contiene el nodo *arm_visualizer*, que básicamente se encarga de leer los topics con los cuaterniones de orientación de tres sensores, con ellos calcula las posiciones y orientaciones de cada eslabón del brazo –brazo, antebrazo y mano– y publica los correspondientes mensajes de ROS para mover la simulación del brazo en el programa Gazebo. En la sección *Ejecución del simulador del brazo robótico del robot Youbot* del anexo PUESTA EN MARCHA DEL SOFTWARE se detalla cómo poner en marcha el visualizador junto con el simulador Gazebo.

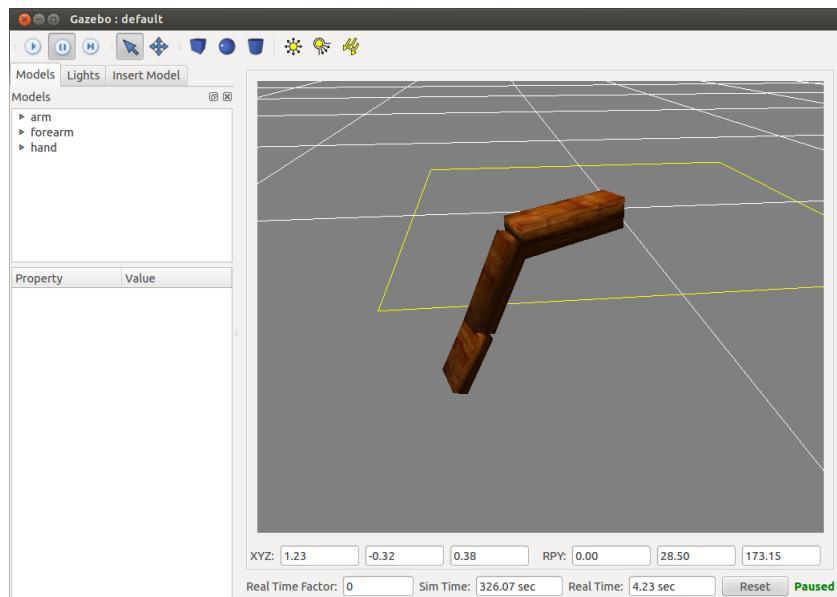


Figura 6.3: Programa Gazebo con un modelo de un brazo humano controlado con el programa *arm_visualizer*. La articulación del hombro se sitúa en el origen de coordenadas. Las cajas representan el brazo, el antebrazo y la mano

6.4.1. La clase *gazebo::CModelList*

La clase *gazebo::CModelList* tiene por objetivo crear una representación de un brazo articulado con las relaciones entre articulaciones y eslabones para poder realizar los cálculos

de las posiciones de cada parte del brazo. Además esta clase también posee la capacidad de crear un mensaje con la descripción y el estado del brazo que puede ser utilizado por el simulador Gazebo para su visualización.

En la siguiente lista se muestran las funciones miembro de la clase:

- **CModelList(ros::NodeHandle& node_handle_):** Constructor de un objeto CModelList. Es necesario pasarle el *handle* del nodo donde se declare para poder realizar las comunicaciones mediante ROS. En el constructor no se crea ningún eslabón ni ninguna articulación.
- **void AddModel
(std::string name, dfv::Vector3 position, std::string xml_model):** Añade un eslabón a la descripción del brazo. Es necesario pasar tres parámetros a esta función:
 - **name:** string con el nombre que se le quiera dar al eslabón.
 - **position:** posición inicial del eslabón. Este parámetro es relevante si el eslabón no va a ser hijo de ningún otro eslabón. En caso contrario la posición se calculará teniendo en cuenta las posiciones y orientaciones de la cadena de eslabones de los que es hijo.
 - **xml_model:** string con la dirección del archivo de extensión xml que contiene la descripción del modelo 3D del eslabón en formato *URDF*³.
- **void Spawn() const:** Crea el mensaje con el que se llamará al servicio de Gazebo */gazebo/spawn_urdf_model* encargado de realizar la inserción del modelo en la simulación. Se ha de llamar a esta función cuando ya se haya realizado la especificación del modelo del brazo, esto es, después de haber construido el modelo mediante las funciones *AddModel()* y *SetJoint()*.
- **void SetJoint
(unsigned int parent_index, unsigned int child_index, dfv::Vector3 parent_joint_position):** Crea una articulación entre los eslabones de índice *parent_index* y *child_index*. El parámetro *parent_joint_position* sirve para especificar la posición de la articulación relativa al origen de coordenadas del eslabón padre.
- **void SetOrientation(unsigned int index, dfv::Quaternion orientation):** Establece la orientación del eslabón de índice *index*.
- **void PublishMessage() const:** Genera y publica un mensaje en el servicio */gazebo/set_model_state* con las orientaciones y posiciones de cada eslabón del brazo. Llamando a esta función se actualiza automáticamente la visualización del modelo en Gazebo.
- **dfv::Vector3 GetPosition(unsigned int index) const:** Devuelve la posición del origen de coordenadas del eslabón de índice *index*. Esta posición tiene en cuenta las posiciones y orientaciones de la cadena de eslabones de los que es hijo el eslabón considerado.

³El formato URDF (Unified Robot Description Format) es una descripción en lenguaje XML de un robot usada por el simulador Gazebo. Un archivo URDF contiene la descripción de las articulaciones, eslabones, propiedades físicas (masas, momentos de inercia, etc), propiedades visuales (color, forma, etc.), entre otras más cosas, que forman un robot.

- **unsigned int GetCount() const:** Devuelve el número de eslabones que se han añadido a la descripción.

```
1 #include <arm_visualizer/gazebo_model_list.h>
2
3 int main(int argc, char** argv)
4 {
5     ros::init(argc, argv, "arm_visualizer");
6     ros::NodeHandle node_handle;
7
8     // Declaración del objeto CModelList
9     // para crear el modelo del brazo
10    gazebo::CModelList arm(node_handle);
11
12    double arm_length = 0.30; // longitud del brazo
13    double forearm_length = 0.25; // longitud del antebrazo
14    double hand_length = 0.17; // longitud de la mano
15
16    // Creación de los eslabones para el brazo, antebrazo y mano
17    arm.AddModel("arm", dfv::Vector3(0.0,0.0,0.0), "model/arm.xml");
18    arm.AddModel("forearm", dfv::Vector3(0.0,0.0,0.0), "model/forearm.xml");
19    arm.AddModel("hand", dfv::Vector3(0.0,0.0,0.0), "model/hand.xml");
20
21    // Creación de las articulaciones para codo y muñeca
22    arm.SetJoint(0, 1, dfv::Vector3(arm_length, 0.0, 0.0));
23    arm.SetJoint(1, 2, dfv::Vector3(forearm_length, 0.0, 0.0));
24
25    // Inserción del brazo en la simulación
26    arm.Spawn();
27
28    while(ros::ok())
29    {
30
31        /*
32         * Asignamos las orientaciones de los eslabones
33         arm.SetOrientation(0, ...);
34         arm.SetOrientation(1, ...);
35         arm.SetOrientation(2, ...);
36        */
37
38        // Publicamos el mensaje con el estado de los eslabones
39        arm.PublishMessage();
40        ros::spinOnce();
41    }
42
43    return 0;
44 }
```

6.5. Controlador de un simulador del brazo robótico del robot Youbot

En la documentación del robot Youbot existe un paquete de ROS que contiene la descripción del brazo del robot para funcionar en Gazebo, y un nodo de ROS –*youbot_arm_publisher*– capaz de controlar las posiciones de las articulaciones del brazo y el *gripper*. Esta simulación está preparada para funcionar tal y como lo hace el robot real; Se puede controlar la posición de las articulaciones con los mismos topics de ROS y los ángulos se corresponden a los del robot real, por lo que se puede usar esta simulación a modo de prueba previa de los programas que se crean para el robot antes de ejecutarlos en el robot real. También puede utilizarse a modo de visualizador de la posición del robot en tiempo real en otro ordenador. En la sección *Instalación del simulador del brazo robótico del YouBot* del anexo *INSTALACIÓN DEL SOFTWARE* se detalla cómo realizar la instalación de esta simulación, y en la sección *Ejecución del simulador del brazo robótico del YouBot* del anexo *PUESTA EN MARCHA DEL SOFTWARE* se describe como lanzar el simulador.

Capítulo 7

PRUEBAS Y RESULTADOS OBTENIDOS

En este capítulo se hará uso de los programas y librerías creados para realizar una serie de pruebas en las que se comprobará cómo funciona el sistema de monitorización en casos de creciente complejidad. En cada prueba se hará también un breve análisis de los resultados obtenidos.

7.1. Captura de datos de un sensor Xsens MTi-G

En esta primera prueba se realizará una simple toma de datos de un sensor Xsens MTi-G (figura 4.3). Se leerá el topic de aceleraciones y se comprobará con qué frecuencia se están publicando datos.

Para comenzar se conectará el sensor Xsens a un ordenador portátil, y se procederá a lanzar el programa *xsens_node* del driver *xsens_driver* mediante los siguientes comandos:

```
$ roscore  
$ rosrun xsens_driver xsens_node
```

El programa reconoce el sensor y comienza a publicar los datos en *topics* de ROS. Ejecutando el comando para obtener los topics que están siendo publicados se obtiene lo siguiente:

```
$ rostopic list  
/rosout  
/rosout_agg  
/xsens_node/sensor0/acc  
/xsens_node/sensor0/gyr  
/xsens_node/sensor0/mag  
/xsens_node/sensor0/ori_quat
```

Se comprueba de la misma forma los parámetros publicados en el servidor de parámetros de ROS:

```
$ rosparam list  
/rosdistro
```

```
/roslaunch/uris/host_daniel_lenovo_g580__54586
/rosversion
/run_id
/xsens_node/output_mode
/xsens_node/output_settings
/xsens_node/sensor_count
```

En ambos casos se comprueba que el programa funciona y los topics están siendo publicados. A continuación se llama a la herramienta *rxplot*, que realiza una gráfica de los datos del *topic* que se le pase como parámetro. En el caso del *topic* de los acelerómetros se obtiene la salida de la figura 7.1 con el sensor puesto sobre el suelo en posición totalmente horizontal. El valor de la aceleración *z* se corresponde aproximadamente a los $9,8 \frac{m}{s^2}$ de la aceleración de la gravedad.

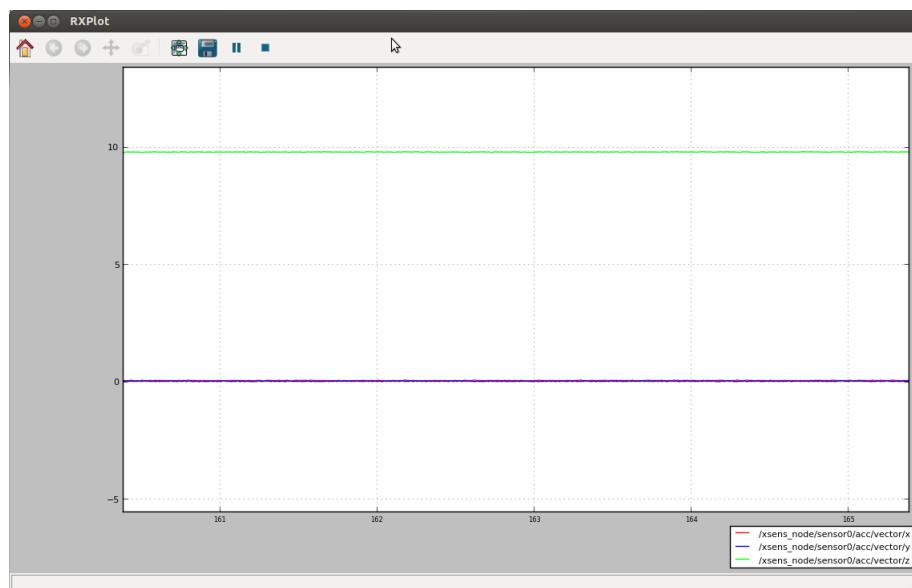


Figura 7.1: Representación gráfica de los datos de los acelerómetros con el sensor puesto de forma totalmente horizontal. La aceleración *z* (en verde) se corresponde con la gravedad.

Por último se hallará la frecuencia a la que se publican los datos. Para hacerlo se redirigirá la salida del comando *rostopic* a un archivo de texto. El mensaje de los acelerómetros tiene su *stamp* de tiempo, por lo que sabiendo el número de mensajes que se han publicado y el tiempo que ha pasado se puede obtener la frecuencia de publicación.

```
$ rostopic echo /xsens_node/sensor0/acc > sensor_output.txt
```

Resultados de la prueba	
Nº de secuencia	tiempo (s)
343058	1374484117,613551113
343578	1374484122,813544786

Cuadro 7.1: Resultados de la prueba para obtener la frecuencia de un sensor Xsens MTi-G.

En la tabla 7.1 se recogen los resultados obtenidos. Se han publicado en total 520 datos en 5,20 segundos, lo que supone una **frecuencia de lectura** de $100,0 Hz$ para un solo sensor.

7.2. Captura de datos de una red de sensores Xsens MTx conectados a un máster Xbus

De la misma forma que para un solo sensor, se realizará la prueba para la obtención de la frecuencia de lectura con una red de sensores Xbus a los que se ha conectado 3 sensores MTx.

Resultados de la prueba	
Nº de secuencia	tiempo (s)
2621	1374485363,188063945
2972	1374485370,208524364

Cuadro 7.2: Resultados de la prueba para obtener la frecuencia de lectura de una red de 3 sensores MTx conectados a un máster Xbus.

En este caso se han publicado 351 mensajes a lo largo de 7,02 segundos, lo que da una **frecuencia de lectura de datos** de $50,0Hz$ para un máster con 3 sensores. Es decir, la frecuencia de lectura ha bajado a la mitad –comprensible, ya que se están obteniendo datos de varios sensores a la vez–. Aún así, es una frecuencia bastante buena para cualquier aplicación.

Se comprueba además con el comando *rostopic* que los *topics* publicados se corresponden a los de acelerómetros, giróscopos, magnetómetros y cuaternión de orientación de los tres sensores (cuatro *topics* por cada sensor):

```
$ rostopic list
/rosout
/rosout_agg
/xsens_node/sensor0/acc
/xsens_node/sensor0/gyr
/xsens_node/sensor0/mag
/xsens_node/sensor0/ori_quat
/xsens_node/sensor1/acc
/xsens_node/sensor1/gyr
/xsens_node/sensor1/mag
/xsens_node/sensor1/ori_quat
/xsens_node/sensor2/acc
/xsens_node/sensor2/gyr
/xsens_node/sensor2/mag
/xsens_node/sensor2/ori_quat
```

7.3. Utilización de los datos obtenidos para el control del brazo robótico de un robot Youbot

Una vez se ha probado el buen funcionamiento del programa *xsens_node* para la lectura de los sensores Xsens, se comprobará como desempeña su tarea junto al programa que realiza el control del brazo robótico de un robot YouBot –el nodo *youbot_controller*–.

Una vez puesto en marcha el robot, se procederá a conectar la red de sensores Xsens al robot y a lanzar el programa *xsens_node* en el PC embebido del YouBot, se forma análoga a como se ha realizado en la prueba anterior. A continuación se ejecuta el driver del YouBot:

```
$ rosrun youbot_oodl youbot_oodl_driver.launch
```

Este programa se queda escuchando ciertos *topics* (tabla 6.4) a la espera de los comandos de posición de las articulaciones. A continuación se colocan los sensores en el brazo humano y se ejecuta el programa *youbot_controller* para la generación y envío de los mensajes a dichos *topics* a partir de los datos de la red de sensores:

```
$ rosrun youbot_controller youbot_controller
```

Ahora es posible realizar el control del brazo robótico mediante el movimiento de los sensores Xsens. El robot responde de forma rápida y fluida, y se puede comprobar que la posición del brazo del robot se corresponde con la del brazo de la persona. Como se anticipó en el capítulo *CÁLCULO DE LAS POSICIONES DE LOS SENSORES*, en la zona de *gimbal lock* se producen saltos en la posición del brazo robótico, pero en el resto de posiciones el conjunto de programas y sensores responde bastante bien.

7.4. Distribución del sistema anterior en varias unidades de procesamiento

Esta prueba consiste en ejecutar los mismos programas que la prueba anterior, pero dividiendo el procesamiento en dos PCs conectados a la misma red wi-fi. Uno de ellos es el propio robot y el otro un ordenador portátil. En máster de ROS estará localizado en el ordenador portátil, por lo que se definirán las correspondientes variables de entorno para esta situación (tabla 7.3).

Valor de las variables de entorno para el sistema distribuido		
	ROS_IP	ROS_MASTER_URI
Portátil	156.35.152.83	http://156.35.152.83:11311
PC YouBot	156.35.152.95	http://156.35.152.83:11311

Cuadro 7.3: Valor de las variables de entorno para el sistema distribuido.

Una vez definidas las variables de entorno, se conectará el máster Xbus con tres sensores al portátil y a continuación se iniciará el *roscore* en el portátil. Una vez el máster de ROS esté en funcionamiento, se ejecutará el driver *youbot_oodl* en el robot –los drivers tienen que ser ejecutados en el ordenador al que está conectado el hardware–, y posteriormente el resto de programas –*xsens_node* y *youbot_controller*– en el portátil. Además se aprovechará que el portátil tiene mayor capacidad de procesamiento que el robot para ejecutar en el primero la simulación del brazo robótico del YouBot.

Ahora el sistema completo está funcionando y se puede proceder a mover los sensores para ver como responde el robot. Se comprueba que el control del brazo robótico funciona bastante bien siempre y cuando se trabaje en entornos con buena conectividad al wi-fi.

Si se mueve el robot a zonas donde hay peor conexión, el control no va tan fluido y hay ocasiones en las que se pierde el control del robot por uno o dos segundos. Sin embargo, estos problemas son por causa de la conectividad wi-fi y no tienen que ver con el propio sistema de monitorización. En zonas donde la conectividad es buena, el robot responde prácticamente de la misma manera que cuando se pone en marcha el sistema en el propio PC del robot.

7.5. Conclusiones

Se ha creado un sistema de monitorización fiable y de alta flexibilidad –se ha comprobado que funciona tanto en el cuerpo humano como en modelos de máquinas, en general en cualquier sistema articulado– aprovechando la gran cantidad de herramientas que proporciona la plataforma ROS. En este sistema de monitorización se han creado además varias librerías que facilitan enormemente su uso por otros desarrolladores en futuras aplicaciones (librería *xsens_driver* y librería *dfv*), y se ha implementado una arquitectura que posibilita la extensión del sistema incorporando un mayor número de sensores del mismo o distinto tipo, o incorporando todo tipo de sistemas actuadores que funcionen en la plataforma ROS. Esta arquitectura además posibilita extender la ejecución de los distintos nodos a otros PCs y así aprovechar, si estuviera disponible, una mayor capacidad de procesamiento para tareas más exigentes computacionalmente. Finalmente cabe mencionar que por la condición de prototipo de este proyecto no se ha buscado un producto final cerrado, si no que se ha dejado la puerta abierta a la modificación e incorporación de otros sistemas de la forma más sencilla posible.

Capítulo 8

BIBLIOGRAFÍA

- Documentación de la plataforma ROS. *ros.org*
- Documentación del simulador Gazebo. *gazebosim.org*
- Documentación de los sensores Xsens:
 - *MT Low-Level Communication Protocol Documentation*, Xsens Technologies, 2010.
 - *MTi-G User Manual and Technical Documentation*, Xsens Technologies, 2010.
 - *MTi and MTx User Manual and Technical Documentation*, Xsens Technologies, 2010.
 - *XB-M Technical Documentation*, Xsens Technologies, 2010.
- Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A., *ROS: an open-source Robot Operating System*, 2009.
- Penrose, R., *El Camino a la Realidad*, Debate Ed. 2006, **11**, 293-316.
- Jazar, R. N., *Theory of Applied Robotics*, Springer Ed. 2^a, 2010, **I.2**, **I.3**, 56-170.

Parte II

Anexos

Apéndice A

INSTALACIÓN DEL SOFTWARE

En el presente capítulo se describirá como instalar las siguientes herramientas:

- ROS Fuerte
- Simulador Gazebo
- Simulador del brazo robótico del Youbot
- Stack de ROS con los programas creados para este proyecto

Se parte de la suposición de que se tiene disponible un ordenador con el sistema operativo Ubuntu 12.04 LTS instalado de forma nativa.

A.1. Instalación de ROS Fuerte

La versión de ROS que se utilizará es ROS Fuerte. Esta elección se debe a que dicha versión es compatible con el simulador Gazebo, con el que posteriormente se realizará la visualización en 3D del modelo. Además es la versión instalada en el PC embebido del robot Youbot.

A.1.1. Configuración de los repositorios de Ubuntu

Para comenzar con la instalación se procederá a abrir el Centro de Software de Ubuntu y en la barra de menú de dicho programa, se seleccionará en el menú Edit la opción Software Sources. En la pestaña Ubuntu Software se comprobará que están seleccionados los repositorios restricted, universe y multiverse, tal como aparece en la figura A.1.

A.1.2. Configuración del archivo sources.list

El archivo sources.list le dice al gestor de paquetes de Ubuntu de dónde puede obtener cada paquete de ROS. Se abrirá un terminal y se ejecutará el siguiente comando:

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

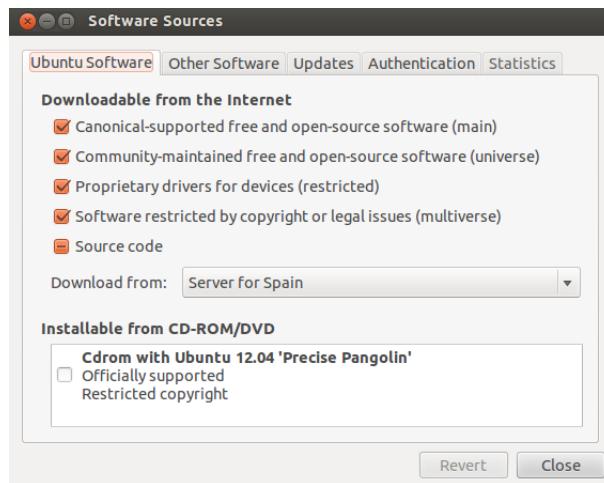


Figura A.1: Ventana Software Sources

Este comando crea un archivo de texto en la ruta especificada como parámetro, que contiene la dirección de donde descargar los paquetes para la versión específica de ROS que se tenga instalada, en este caso ROS Fuerte.

A.1.3. Configuración de la keys

En el terminal se ejecutará el siguiente comando:

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

A.1.4. Descarga e instalación

Se actualizará el índice de paquetes de Ubuntu para tener la seguridad de que el servidor de ROS.org está indexado:

```
$ sudo apt-get update
```

A continuación se procederá a descargar e instalar la versión completa de ROS. El terminal se ejecutará el siguiente comando:

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

Esta instalación traerá consigo las herramientas mostradas en la siguiente lista, entre muchas otras:

- ROS
- rx (herramientas para interfaz gráfica: rxbag, rxgraph, rxplot, ...)
- rviz (herramienta de visualización 3D)
- librerías genéricas para robots
- Simuladores 2D/3D (entre ellos Gazebo)
- Navegación y percepción 2D y 3D

A.1.5. Configuración del entorno

Cada vez que se inicie un nuevo terminal es necesario añadir las variables de entorno. Si se quisiera, se puede automatizar dicha tarea ejecutando el comando:

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

Este comando añade la línea source /opt/ros/fuerte/setup.bash al archivo /.bashrc. Este archivo contiene la configuración inicial del terminal, y se ejecuta cada vez que abrimos un nuevo terminal. Posteriormente se ejecutará el archivo anterior para actualizar el terminal. De esta forma reconocerá los nuevos comandos de ROS:

```
$ . ~/.bashrc
```

A.2. Instalación del simulador Gazebo

Si se ha realizado el proceso de instalación de ROS detallado anteriormente, el simulador Gazebo ya vendrá de serie con dicha versión de ROS, y por lo tanto no haría falta seguir los pasos de este apartado. Si se hubiera instalado ROS de otra manera y éste no hubiera venido con el programa Gazebo, se tendría que instalar el programa de forma manual. Para instalar la versión de Gazebo preparada para comunicarse con ROS se ejecutará el siguiente comando:

```
$ sudo apt-get install ros-fuerte-simulator-gazebo
```

A.3. Instalación del simulador de brazo robótico del YouBot

En el robot Youbot ya vienen por defecto los paquetes con los que ejecutar el simulador del brazo robótico con Gazebo. Si se quisiera instalar la simulación en otro ordenador habría que descargarse los paquetes necesarios de la página web de KUKA. Los pasos a seguir son los siguientes:

1. Descarga de los paquetes con dependencias:

```
$ sudo apt-get install ros-fuerte-pr2-controllers
$ sudo apt-get install ros-fuerte-pr2-simulator
```

2. Instalación desde git de los paquetes con la simulación. Suponemos que se ha creado un *workspace* para trabajar con ROS, y dentro de éste hay una carpeta llamada *sandbox*:

```
$ roscd
$ cd sandbox
$ git clone https://github.com/youbot/youbot-ros-pkg.git
$ git clone https://github.com/ipa320/cob_common.git
```

3. Compilación de los paquetes

```
$ rosmake brics_actuator youbot_description
```

A.4. Instalación del stack youbot-xsens-controller

Para instalar el stack youbot-xsens-controller desde el CD adjunto, simplemente se copiará la carpeta *youbot-xsens-controller* en el directorio en el que se halle el *workspace* de ROS. A continuación se compilará mediante el comando:

```
$ rosmake youbot-xsens-controller
```

Este stack tiene por dependencia los stacks cuya instalación se detalla en el apartado *Instalación del simulador de brazo robótico del YouBot* de este mismo capítulo, por lo que es necesario haber seguido los pasos descritos en dicho apartado antes de realizar la compilación del stack.

Apéndice B

PUESTA EN MARCHA DEL SOFTWARE

B.1. Iniciación del driver del sensor Xsens

Para ejecutar el driver *xsens_node* es necesario iniciar antes el máster de ROS. Para ello en un terminal se ejecutará el siguiente comando:

```
$ roscore
```

A continuación, una vez que los sensores Xsens y el máster Xbus estén conectados al PC, se iniciará el programa driver:

```
$ rosrun xsens_driver xsens_node
```

Si no se ha producido ningún error, ahora el driver estará publicando en ROS los mensajes con los datos leídos del sensor.

B.2. Iniciación del driver del robot Youbot

Para lanzar el driver *youbot_oodl* se ejecutará el siguiente comando en el PC del robot YouBot:

```
$ roslaunch youbot_oodl youbot_oodl_driver.launch
```

B.3. Ejecución del programa *youbot_controller*

Para lanzar el programa *youbot_controller* se ejecutará el siguiente comando en el PC correspondiente:

```
$ rosrun youbot_controller youbot_controller
```

B.4. Ejecución del simulador del brazo robótico del robot Youbot

Para lanzar el simulador del brazo robótico se ejecutará el siguiente comando en el PC correspondiente (se evitará lanzarlo en el PC del robot YouBot ya que Gazebo consume muchos recursos y el robot no responde muy bien en este caso):

```
$ rosrun youbot_description youbot_arm_publisher.launch
```

En caso de que Gazebo no se abriese, véase el capítulo *SOLUCIÓN DE PROBLEMAS*.

B.5. Ejecución de la visualización del modelo del brazo humano

Para poner en marcha este programa hace falta haber iniciado primero el driver de los sensores Xsens y el simulador Gazebo. Los pasos a seguir son los siguientes:

1. Una vez conectados los sensores al PC, en un terminal ejecutamos los siguientes comandos para iniciar el programa *xsens_node*:

```
$ roscore  
$ rosrun xsens_driver xsens_node
```

2. Una vez se compruebe que los sensores están publicando en ROS, se procederá a iniciar Gazebo con el archivo *empty_world.launch*:

```
$ rosrun gazebo_gui empty_world.launch
```

3. Se abrirá el GUI de Gazebo, en el que habrá una simulación de un plano (figura B.1).

4. Para la correcta simulación del brazo se tendrá que eliminar el plano de la simulación (figura B.2).

5. También será necesario poner la simulación en pausa, ya que de otra manera se realizará la simulación de la gravedad sobre el brazo y se obtendrán resultados no deseados (figura B.3).

6. A continuación se procederá a ejecutar el programa *arm_visualizer*, el cuál insertará el brazo en la simulación y se encargará de comunicar a Gazebo las posiciones de cada eslabón del brazo. Para ello en un terminal se ejecutará el siguiente comando:

```
$ rosrun arm_visualizer arm_visualizer
```

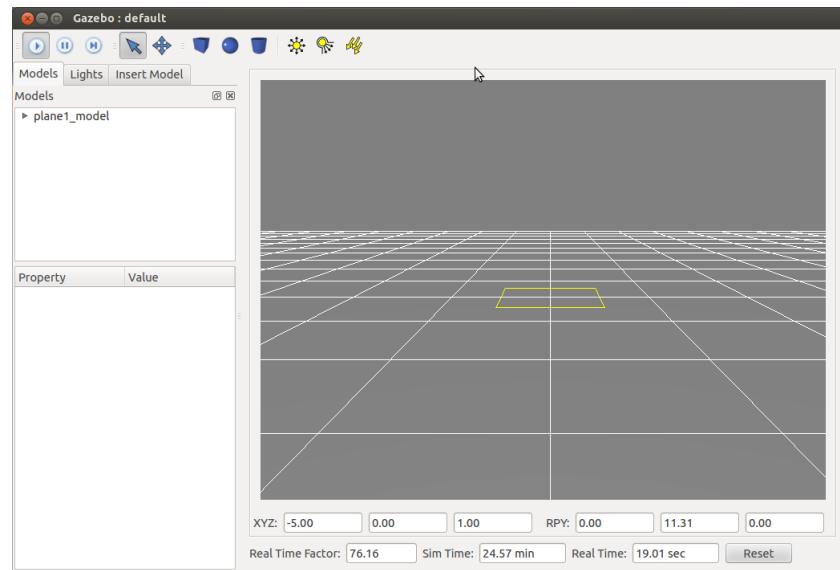


Figura B.1: Ventana de Gazebo abierta con el archivo empty_world.launch

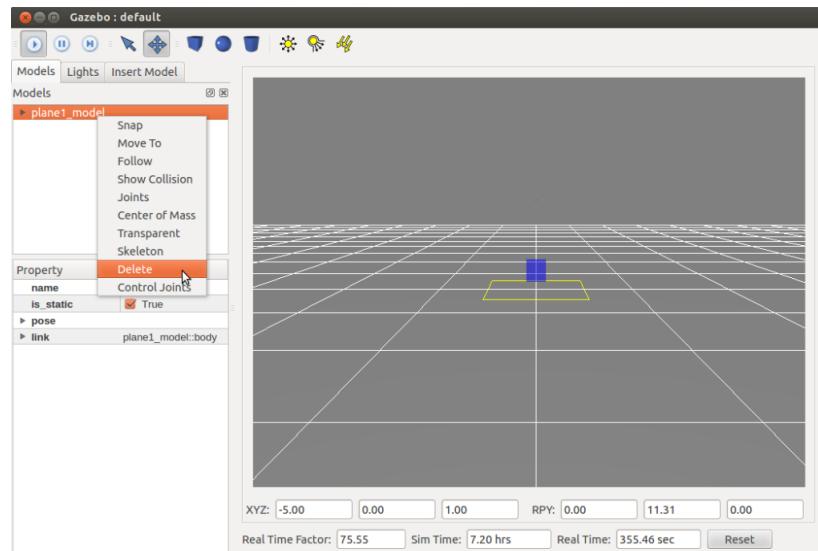


Figura B.2: Eliminación del plano en la simulación de Gazebo.

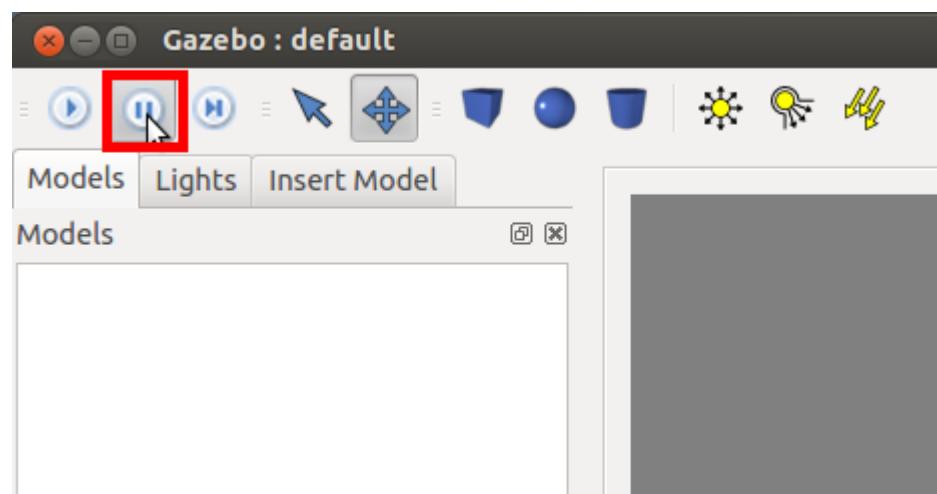


Figura B.3: Puesta en pausa de la simulación de Gazebo.

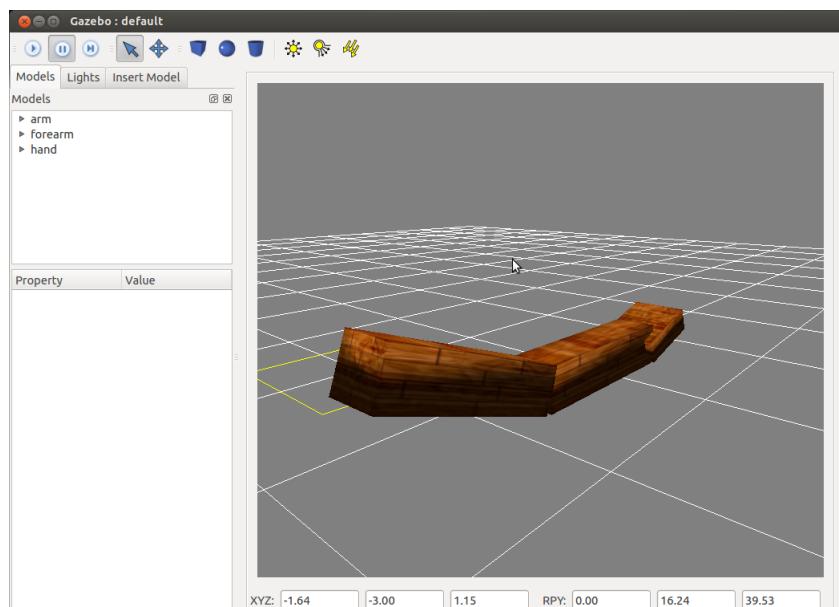


Figura B.4: Visualización del brazo en Gazebo.

Apéndice C

SOLUCIÓN DE PROBLEMAS

En este capítulo se da solución a determinados problemas que se han encontrado de forma frecuente al poner en marcha el sistema de monitorización.

C.1. Error al iniciar Gazebo

Este error consiste en que al iniciar Gazebo con la simulación de un mundo vacío, este no se abre y en la terminal aparecen mensajes parecidos a los que se muestran a continuación:

```
Msg Waiting for master
Msg Connected to gazebo master @ http://localhost:11345
Exception [Master.cc:69] Unable to start server[Address already in use]

terminate called after throwing an instance of 'gazebo::common::Exception'
Aborted (core dumped)
[gazebo-1] process has died [pid 2795, exit code 134, cmd
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo/scripts/gazebo
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo_worlds/worlds/empty.world __name:=gazebo
__log:=/home/daniel/.ros/log/772c2f96-ab75-11e2-a2fc-001de05009b5/gazebo-1.log].
log file: /home/daniel/.ros/log/772c2f96-ab75-11e2-a2fc-001de05009b5/gazebo-1*.log
LightListWidget::OnLightMsg
```

Esto es debido a que ya existe un proceso Gazebo ejecutándose en el ordenador pero por alguna razón no ha abierto su interfaz gráfica.

Solución: Ejecutar comando:

```
$ ps ax | grep [gz]
```

Se comprobará si hay un proceso gzserver

```
3118 ? S1 12:47
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo/bin/gzserver
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo_worlds/worlds/empty.world __name:=gazebo
__log:=/home/daniel/.ros/log/8188cc76-ab73-11e2-a4ec-001de05009b5/gazebo-1.log -s
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo/lib/libgazebo_ros_paths_plugin.so -s
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo/lib/libgazebo_ros_api_plugin.so
```

Si lo hay, se ejecutará el programa *System Monitor* y se matará el proceso *gzserver*.

C.2. Al ejecutar el simulador del brazo robótico no se abre la interfaz gráfica de Gazebo

Es posible que al lanzar Gazebo con el archivo *youbot_arm_publisher.launch* no se abra la ventana de Gazebo. Para solucionarlo es necesario abrir dicho archivo y comprobar que sea como el siguiente:

```

1 <?xml version="1.0"?>
2 <launch>
3     <!-- start gazebo -->
4     <!<param name="/use_sim_time" value="true" /> -->
5     <param name="/use_sim_time" value="true" />
6     <node name="gazebo" pkg="gazebo" type="gazebo" args="$(find
7         gazebo_worlds)/worlds/empty.world" respawn="false" output="screen">
8         <env name="GAZEBO_RESOURCE_PATH" value="$(find
9             youbot_description):$(find gazebo_worlds):$(find
10                gazebo)/gazebo/share/gazebo" />
11     </node>
12
13     <!-- Abrir GUI de Gazebo -->
14     <node name="gazebo_gui" pkg="gazebo" type="gui" respawn="false"
15         output="screen"/>
16
17     <!-- send youbot urdf to param server -->
18     <param name="robot_description" command="$(find xacro)/xacro.py '$(find
19         youbot_description)/robots/youbot_arm.urdf.xacro'" />
20
21     <!-- push robot_description to factory and spawn robot in gazebo -->
22     <node name="youbot_gazebo_model" pkg="gazebo" type="spawn_model"
23         args="-urdf -param robot_description -model youBot -x 0.4 -y 2 -z
24             0.05" respawn="false" output="screen" />
25
26     <include file="$(find
27         youbot_description)/launch/control/youbot_arm_control.launch" />
28     <include file="$(find
29         youbot_description)/launch/control/youbot_gripper_control.launch" />
30 </launch>
```

Este archivo se encuentra en la carpeta *launch/* dentro del paquete *youbot-ros-pkg/youbot_common/youbot_description*

C.3. El driver del Xsens no contacta con los sensores

Es posible que en los primeros instantes tras conectar el sensor o red de sensores al PC, el programa *xsens_node* no reconozca los sensores y finalice su ejecución. En este caso se deberán esperar 5-10 segundos antes de volver ejecutar el programa.

Parte III

Presupuesto

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN				
PRESUPUESTO				
Nº	Nº de unidades	Designación	Precio unitario	Coste (€)
1	70 horas	Horas de documentación	50 €	3500 €
2	200 horas	Horas de programación	25 €	5000 €
3	50 horas	Elaboración de la documentación	50 €	2500 €
4	0,1 unidades	Usufructo del sistema Xsens	9000 €	900 €
5	0,1 unidades	Usufructo del robot YouBot	20000 €	2000 €
		Subtotal		13900 €
		IVA(21 %)		2919 €
		TOTAL		16819 €

El coste total del proyecto asciende a DIECISÉIS MIL OCHOCIENTOS DIECINUEVE Euros.

En Gijón a 22 de Julio de 2013.

El proyectante: