Sistema de monitorización inercial del movimiento de las extremidades superiores

Daniel Fernández Villanueva

28 de junio de 2013

Índice general

Índice de figuras

Índice de cuadros

Parte I

Memoria

Capítulo 1

ANTECEDENTES

Capítulo 2

OBJETIVO DEL PROYECTO

Este proyecto tiene como objetivo la implementación de un sistema de monitorización en tiempo real del movimiento de las extremidades superiores del cuerpo humano. Para la realización de este sistema se tendrá que dar solución a los siguientes

1. **Aplicación para la adquisición de datos de los sensores inerciales**:
   Realización de un sistema que permitirá obtener en tiempo real los datos que proporcionan los sensores. En concreto se desarrollará un driver para un sensor xsens o una red de sensores xsens conectados mediante un master xbus. Este driver permitirá leer los datos que proporcionan los acelerómetros, giróscopos, magnetómetros y sensores de temperatura, además de la orientación de cada uno de los sensores conectados al PC. Este driver se encargará también de crear una interfaz para la posterior utilización de los datos en otros programas de foma sencilla.

2. **Tratamiento de los datos para obtener los ángulos de rotación entre cada sensor**:
   Una vez sea posible la adquisición de los datos con el driver anterior, se creará otro programa con el que se obtendrán los ángulos de rotación entre cada sensor teniendo en cuenta además la geometría de las articulaciones del brazo o modelo sobre las que se situarán los sensores.

3. **Utilización de los datos para el objetivo deseado**:
   En esta última fase se crearán los sistemas necesarios para la utilización de los datos con el objetivo deseado:

   - Para la visualización de la posición del brazo en el simulador 3D Gazebo, se creará un modelo del brazo y una interfaz etre ROS y el simulador que permitirá la visualización de la posición del brazo en tiempo real.
   - Para el control del robot mediante el movimiento del brazo o modelo del robot físico, se creará otra interfaz entre ROS y el driver del propio robot.

Capítulo 3

ESPECIFICACIONES DE DISEÑO

Capítulo 4

DISEÑO DEL SISTEMA

4.1. Estudio de soluciones

4.2. Simulación

Capítulo 5

IMPLEMENTACIÓN FÍSICA

5.1. Selección de componentes

5.2. Montaje

5.3. Ajuste

Capítulo 6

PROTOCOLO DE PRUEBAS. REDISEÑO

Capítulo 7

RESULTADOS OBTENIDOS

Capítulo 8

HERRAMIENTAS UTILIZADAS

En este capítulo se realizará una breve descripción de los elementos utilizados en el proyecto, tanto de hardware como de software.

8.1. Hardware

8.1.1. Sensores XSENS

Sensor MTi-G

XBUS Master

8.1.2. Brazo humano

8.1.3. Robot Youbot

8.1.4. Modelo del robot Youbot

8.2. Software

8.2.1. ROS

¿Qué es 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.

¿Por qué usar ROS?

ROS proporciona solución a diversos problemas que vienen dados inherentemente al objetivo de este proyecto:

- **Creación y compilación de programas**
  - **Gestor de paquetes**
- **Comunicación entre programas:** ROS incluye:
  - **Máster**
  - **Topics**
  - **Servicios**
  - **Servidor de parámetros**

- **Visualización de datos**
- **Otras herramientas**
  - **Bag**
  - **rxplot**

8.2.2. El sistema operativo Ubuntu

¿Qué es Ubuntu?

Ubuntu es un sistema operativo con núcleo Linux. Es gratuito y es distribuido como software *open source*. Se trata de la distribución GNU/Linux más popular en equipos personales

¿Por qué usar Ubuntu?

8.2.3. El lenguaje de programación C++

8.2.4. Simulador Gazebo

Gazebo es programa para simulación en 3D de un robot o una población de robots interactuando entre sí y el ambiente. Incorpora simulación de la física de sólidos rígidos y de la respuesta de sensores. Además incorpora un visualizador 3D bastante potente que permite ver en tiempo real la posición de los robots de forma realista.

La versión de Gazebo que se va a utilizar proporciona también una serie de herramientas para la comunicación con ROS.

8.2.5. Visualizador RViz

8.2.6. Control de versiones: git

Capítulo 9

PROCESO DE REALIZACIÓN

En este capítulo se detallará el proceso de realización de cada una de las fases del proyecto.

9.1. Creación del driver para la adquisición de datos de los sensores xsens

En está primera fase se tratará de encontrar un método para la toma de datos de la red de sensores inerciales. Estos sensores estarán conectados a un máster, que irá conectado al PC mediante conexión USB. Los datos así obtenidos se publicarán en *topics* de ROS.

Método seguido

Para la realización del driver se ha partido del código incluido en la documentación de los sensores

9.1.1. El paquete xsens_driver

9.2. Creación de una librería matemática en C++ que permita trabajar con posiciones y orientaciones

9.2.1. La librería dfv

La clase Quaternion

La clase Vector3

La clase Matrix

Capítulo 10

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 sistemas. 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 previa de los algoritmos cuya implementación se mostrará más adelante.

10.1. Formas de representar orientaciones espaciales

Existen multitud de formas de representar 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.

10.1.1. 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.

10.1.2. Matriz de rotación

Una matriz de rotación es una matriz usada para expresar una rotación en un espacio euclideo. 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*.

10.1.3. 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:

$$q = w + xi + yj + zk$$

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	$q1 = 1q = q$

Cuadro 10.1: 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 facilmente un cuaternión a partir de un eje y un ángulo, y viceversa. Estas operaciones son más complejas para matrices de rotación y angulos 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 cumplir unas de propiedades. Los cuaterniones tienen que ser unitarios y la matriz de rotación tiene que ser ortogonal. Es bastante más facil 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 facil componer una interpolación esférica (llamada *slerp - spherical linear interpolation*) para producir una rotación suave a lo largo del tiempo.

10.1.4. Conclusión

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.

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

Sea un vector unitario:

$$\vec{e} = e_x i + e_y j + e_z k, \quad \|e\| = \sqrt{e_x^2 + e_y^2 + e_z^2} = 1 \tag{10.1}$$

Se definirá el cuaternión de rotación con ángulo θ sobre el eje dado por el vector $\vec{e}$:

$$q(\theta, \vec{e}) = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) e_x i + \sin\left(\frac{\theta}{2}\right) e_y j + \sin\left(\frac{\theta}{2}\right) e_z k \tag{10.2}$$

Se puede descomponer el cuaternión como suma de un número real y un cuaternión imaginario puro multiplicado por otro número real:

$$q(\theta, \vec{e}) = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) e \tag{10.3}$$

Donde e es el cuaternión 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:

$$c_i = \cos\left(\frac{\theta_i}{2}\right) \tag{10.4}$$

$$s_i = \sin\left(\frac{\theta_i}{2}\right) \tag{10.5}$$

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

$$q(\theta_i, \vec{e_i}) = c_i + s_i e_i \tag{10.6}$$

Si se tienen en cuenta las propiedades del cuadro 10.1, el producto de dos cuaterniones se puede expresar de esta forma:

$$q_1 q_2 = (c_1 + s_1 e_1)(c_2 + s_2 e_2) = c_1 c_2 + c_1 s_2 e_2 + c_2 s_1 e_1 + s_1 s_2 e_1 e_2 \tag{10.7}$$

En donde:

$$\begin{aligned} e_1 e_2 &= (e_{1_x} i + e_{1_y} j + e_{1_z} k)(e_{2_x} i + e_{2_y} j + e_{2_z} k) = \\ &= e_{1_x} i(e_{2_x} i + e_{2_y} j + e_{2_z} k) + e_{1_y} j(e_{2_x} i + e_{2_y} j + e_{2_z} k) + e_{1_z} k(e_{2_x} i + e_{2_y} j + e_{2_z} k) = \\ &= -(e_{1_x} e_{2_x} + e_{1_y} e_{2_y} + e_{1_z} e_{2_z}) + i(e_{1_y} e_{2_z} + e_{1_z} e_{2_y}) + j(-e_{1_x} e_{2_z} + e_{1_z} e_{2_x}) + k(e_{1_x} e_{2_y} + e_{1_y} e_{2_x}) \end{aligned} \tag{10.8}$$

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

Definición 10.2.1. *Sean dos cuaterniones imaginarios puros q_1 y q_2 tales que:*

$$q_1 = x_1 i + y_1 j + z_1 k$$

$$q_2 = x_2 i + y_2 j + z_2 k$$

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

$$q_1 \cdot q_2 = x_1 x_2 + y_1 y_2 + z_1 z_2$$

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 10.2.2. *Sean dos cuaterniones imaginarios puros q_1 y q_2 tales que:*

$$q_1 = x_1 i + y_1 j + z_1 k$$

$$q_2 = x_2 i + y_2 j + z_2 k$$

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

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

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 (10.8):

$$e_1 e_2 = -e_1 \cdot e_2 + e_1 \times e_2 \tag{10.9}$$

Por lo tanto, sustituyendo (10.9) en la ecuación (10.7):

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

$$q_1 q_2 = c_1 c_2 - s_1 s_2 (e_1 \cdot e_2) + c_1 s_2 e_2 + c_2 s_1 e_1 + s_1 s_2 (e_1 \times e_2) \tag{10.10}$$

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:

$$conj(q) = q^* = c - se \tag{10.11}$$

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

$$\|q\| = 1, \quad q^{-1} = \frac{q^*}{\|q\|^2} = q^* \tag{10.12}$$

De tal forma que:

$$q q^* = q^* q = 1 \tag{10.13}$$

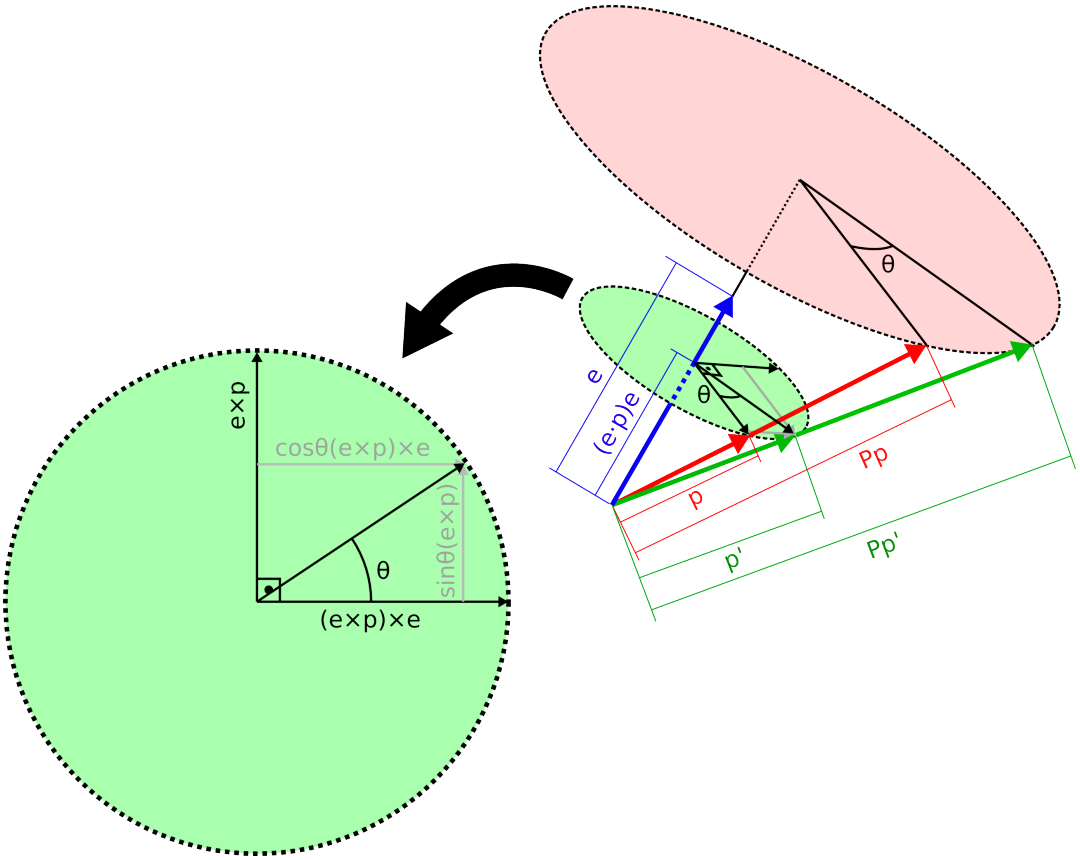

Figura 10.1: Rotación de un vector mediante la operación $q_1 p q_1^*$

10.2.1. 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:

$$p' = qpq^*, \quad q = q(\theta, \vec{e}) \tag{10.14}$$

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

$$\vec{p} = p_x\vec{i} + p_y\vec{j} + p_z\vec{k} \quad \Rightarrow \quad p = p_x i + p_y j + p_z k \tag{10.15}$$

Se comprueba que en efecto, desarrollando la expresión para un cuaternión genérico q y un vector expresado como cuaternión Pp, donde P es el módulo del vector y p es el cuaternión 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:

$$q(Pp)q^* = P(c^2p + 2cs(e \times p) + s^2(e \cdot p)e - s^2(e \times p) \times e) \tag{10.16}$$

Se le puede asignar a cada término un significado geométrico:

- $(e \cdot p)e$ es la proyección de p sobre e
- $(e \times p)$ 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 10.1 se muestra más claramente el significado de cada término.

Podemos descomponer así el término c^2p como suma de dos de éstos vectores:

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

Sustituyendo el resultado en (10.16):

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

Reorganizando términos:

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

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

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

De la figura 10.1 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 vector rotado.

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

$$q_1^*p'q_1 = q_1^*(q_1pq_1)q_1^* \tag{10.17}$$

$$q_1^*p'q_1 = p \tag{10.18}$$

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

$$q = q\left(\theta, \vec{e}\right) \quad \Rightarrow \quad q^* = q\left(\theta, -\vec{e}\right) = q\left(-\theta, \vec{e}\right) \tag{10.19}$$

10.2.2. Composición de rotaciones en coordenadas extrínsecas

El cuaternión de rotación definido por $q\left(\theta, \vec{e}\right)$ 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 $q_1 = q\left(\theta_1, \vec{e_1}\right)$, se obtendrá una rotación compuesta por una primera rotación seguida de otra rotación caracterizada por $q_2 = q\left(\theta_2, \vec{e_2}\right)$:

$$\begin{aligned} \text{Primera rotación:} \quad & p' = q_1pq_1^*, \quad q_1 = q(\theta_1, \vec{e_1}) \\ \text{Segunda rotación:} \quad & p'' = q_2p'q_2^*, \quad q_2 = q(\theta_2, \vec{e_2}) \end{aligned}$$

$$p'' = q_2(q_1pq_1^*)q_2^* = (q_2q_1)p(q_1^*q_2^*) = q_{12}pq_{12}^* \tag{10.20}$$

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:

$$q_{12} = q_2q_1 \tag{10.21}$$

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

$$q_{12} = (q_2q_1)^* = q_1^*q_2^* \tag{10.22}$$

De forma análoga, para n cuaterniones:

$$q_{12\ldots(n-1)n} = q_n q_{n-1} \cdots q_2 q_1 \tag{10.23}$$

$$\left(q_{12\ldots(n-1)n}\right)^* = (q_n q_{n-1} \cdots q_2 q_1)^* = q_1^* q_2^* \cdots q_{n-1}^* q_n^* \tag{10.24}$$

10.2.3. 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 cuaternión 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 cuaternión q_1. Se quiere realizar una rotación de ángulo θ_2 sobre un eje e_2^L en coordenadas locales. El eje expresado en coordenadas globales será:

$$e^G = q_1 e^L q_1^*$$

Teniendo en cuenta la ecuación (10.6), el cuaternión de rotación asociado al ángulo θ_2 y el vector local e_2^L despueś de que el sólido haya sufrido una rotación definida por el cuaternión q_1 será:

$$q_{2_{q_1}} = q(\theta_2, \vec{e}^G) = \cos\left(\frac{\theta_2}{2}\right) + \sin\left(\frac{\theta_2}{2}\right) e^G$$

$$q_{2_{q_1}} = c_2 + s_2(q_1 e^L q_1^*) \tag{10.25}$$

10.2.4. Relación entre rotaciones intrínsecas y extrínsecas

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 cuaternión de rotación asociado al eje local $\vec{e_2}^L$ después de haber sufrido una rotación definida por el cuaternión $q_1 = q\left(\theta_1, \vec{e_1}\right)$ como:

$$q_{2_{q_1}} = q\left(\theta_2, \vec{e_2}^L\right), \quad \vec{e_2}^L = (q_1 e_2 q_1^*)_{\vec{v}} \tag{10.26}$$

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 [1], por lo que:

$$q_1^G = q_1^L = q_1 \tag{10.27}$$

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:

$$q_2^L q_1 = q_1 q_2^G \tag{10.28}$$

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

$$q_2^L q_1 = q_1 q_2^G \iff q_2^L = q_1 q_2^G q_1^* \tag{10.29}$$

Desarrollo del lado izquierdo de la igualdad:

$$q_2^L = c_2 + s_2(q_1 e_2 q_1^*) \tag{10.30}$$

$$\begin{aligned} q_1 e_2 q_1^* &= (c_1 + s_1 e_1) e_2 (c_1 - s_1 e_1) = (c_1 e_2 + s_1 e_1 e_2)(c_1 - s_1 e_1) = \\ &= c_1^2 e_2 + s_1 c_1 e_1 e_2 - c_1 s_1 e_2 e_1 - s_1^2 e_1 e_2 e_1 = c_1^2 e_2 + s_1 c_1 (e_1 e_2 - e_2 e_1) - s_1^2 e_1 e_2 e_1 \end{aligned} \tag{10.31}$$

[1] Por definición el origen del cuaternión 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 cuaternión de rotación será el mismo.

De (10.9) se tiene lo siguiente:

$$e_1e_2 = -e_1 \cdot e_2 + e_1 \times e_2$$
$$e_2e_1 = -e_2 \cdot e_1 + e_2 \times e_1 = -e_1 \cdot e_2 - e_1 \times e_2$$

$$e_1e_2 - e_2e_1 = 2(e_1 \times e_2) \tag{10.32}$$

$$e_1e_2e_1 = (-e_1 \cdot e_2 + e_1 \times e_2)e_1 = (-e_1 \cdot e_2)e_1 + (e_1 \times e_2)e_1$$

De esta ecuación:

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

$(e_1 \times e_2)$ será un vector perpendicular a e_1, por lo que $(e_1 \times e_2) \cdot e_1 = 0$:

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

Por lo tanto:

$$e_1e_2e_1 = -(e_1 \cdot e_2)e_1 + (e_1 \times e_2) \times e_1 \tag{10.33}$$

Sustituyendo en (10.31):

$$q_1e_2q_1^* = c_1^2e_2 + s_1c_1(e_1e_2 - e_2e_1) - s_1^2e_1e_2e_1 =$$
$$c_1^2e_2 + 2s_1c_1(e_1 \times e_2) - s_1^2(-(e_1 \cdot e_2)e_1 + (e_1 \times e_2) \times e_1)$$

$$q_1e_2q_1^* = c_1^2e_2 + 2s_1c_1(e_1 \times e_2) + s_1^2(e_1 \cdot e_2)e_1 - s_1^2(e_1 \times e_2) \times e_1 \tag{10.34}$$

Finalmente, sustituyendo en (10.30):

$$q_2^L = c_2 + s_2(c_1^2e_2 + 2s_1c_1(e_1 \times e_2) + s_1^2(e_1 \cdot e_2)e_1 - s_1^2(e_1 \times e_2) \times e_1)$$

$$q_2^L = c_2 + s_2c_1^2e_2 + 2s_1s_2c_1(e_1 \times e_2) + s_1^2s_2(e_1 \cdot e_2)e_1 - s_1^2s_2(e_1 \times e_2) \times e_1 \tag{10.35}$$

Ahora se procederá a desarrollar el lado derecho de la igualdad (10.29):

$$q_1q_2^Gq_1^* = (c_1 + s_1e_1)(c_2 + s_2e_2)(c_1 - s_1e_1) =$$

$$= (c_1c_2 + s_2c_1e_2 + s_1c_2e_1 + s_1s_2e_1e_2)(c_1 - s_1e_1) =$$

$$= c_1^2c_2 + s_2c_1^2e_2 + s_1c_1c_2e_1 + s_1s_2c_1e_1e_2 - s_1c_1c_2e_1 - s_1s_2c_1e_2e_1 - s_1^2c_2e_1e_1 - s_1^2s_2e_1e_2e_1$$

Como $\|e_1\| = 1$, se tendrá que:

$$e_1e_1 = -e_1 \cdot e_1 + e_1 \times e_1 = -1 \tag{10.36}$$

Sustituyendo y reorganizando:

$$q_1q_2^Gq_1^* = c_2 + s_2c_1^2e_2 + s_1s_2c_1(e_1e_2 - e_2e_1) - s_1^2s_2e_1e_2e_1$$

De (10.32) y (10.33):

$$q_1q_2^Gq_1^* = c_2 + s_2c_1^2e_2 + 2s_1s_2c_1(e_1 \times e_2) - s_1^2s_2(-(e_1 \cdot e_2)e_1 + (e_1 \times e_2) \times e_1)$$

$$q_1q_2^Gq_1^* = c_2 + s_2c_1^2e_2 + 2s_1s_2c_1(e_1 \times e_2) + s_1^2s_2(e_1 \cdot e_2)e_1 - s_1^2s_2(e_1 \times e_2) \times e_1 \tag{10.37}$$

Finalmente se compara (10.35) con (10.37):

$$q_2^L = c_2 + s_2c_1^2e_2 + 2s_1s_2c_1(e_1 \times e_2) + s_1^2s_2(e_1 \cdot e_2)e_1 - s_1^2s_2(e_1 \times e_2) \times e_1$$

$$q_1q_2^Gq_1^* = c_2 + s_2c_1^2e_2 + 2s_1s_2c_1(e_1 \times e_2) + s_1^2s_2(e_1 \cdot e_2)e_1 - s_1^2s_2(e_1 \times e_2) \times e_1$$

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

$$q_2^L = q_1q_2^Gq_1^* \tag{10.38}$$

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:

$$q_2^Lq_1 = q_1q_2^G \tag{10.39}$$

Ahora se va a suponer que el cuaternión q_1 está compuesto de otras dos rotaciones, que se van a expresar de forma global y local según la ecuación (10.39):

$$q1 = q_{1b}^Lq_{1a} = q_{1a}q_{1b}^G$$

Sustituyendo en (10.39):

$$q_2^Lq_{1b}^Lq_{1a} = q_{1a}q_{1b}^Gq_2^G$$

Renombrando los términos:

$$q_3^Lq_2^Lq_1 = q_1q_2^Gq_3^G$$

Se puede generalizar el resultado a n rotaciones:

$$q_n^L \cdots q_3^Lq_2^Lq_1 = q_1q_2^Gq_3^G \cdots q_n^G \tag{10.40}$$

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.

10.2.5. 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.

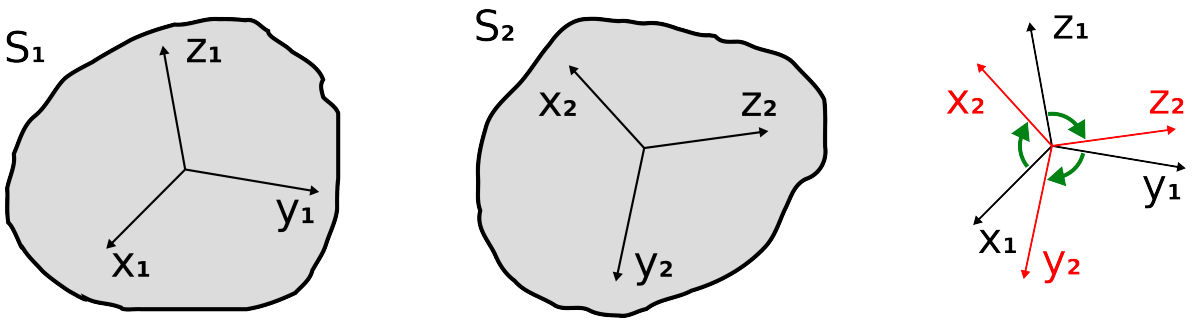

Figura 10.2: Orientación relativa entre dos sólidos rígidos

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 cuaternión 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 cuaternión que define la orientación del sólido 1 y q_2 al que define la orientación del sólido q_2. Si aplicamos 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:

$$q_{S_A S_B} = q_B^* q_A \tag{10.41}$$

10.3. Cálculo de la posición del brazo

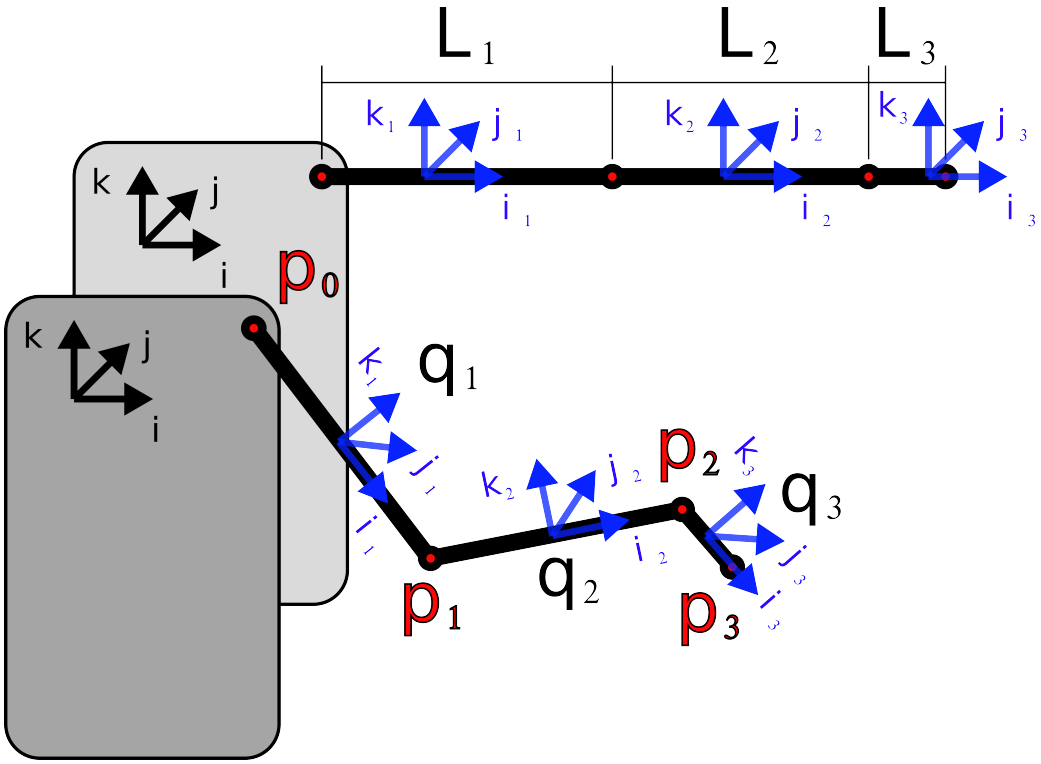

Figura 10.3: Posiciones del brazo inicial y genérica

10.4. 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. Para definir un estado de rotación hacen falta tres ángulos sobre tres ejes perpendiculares entre sí. 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 $< \psi : yaw, \theta : pitch, \phi : roll >$ en un sistema de coordenadas intrínsecas, lo que equivale a $< \phi, \theta, \psi >$ 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.

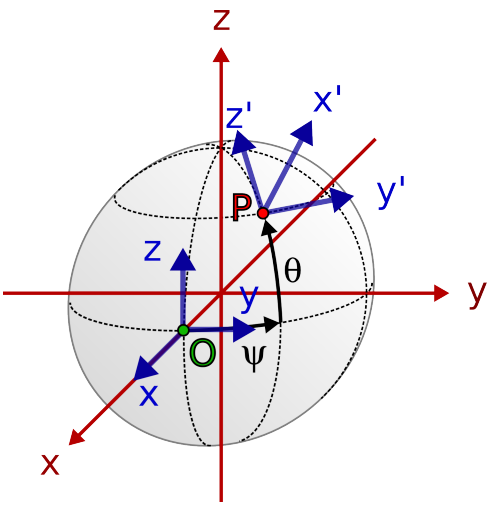

Figura 10.4: Sistema de coordenadas local sobre la esfera

Se mueve un punto desde el origen de coordenadas O hasta un punto P, tal como se muestra en la figura 10.5. El punto P tiene por coordenadas:

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

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.

$$i' = \cos\psi\cos\theta i + \sin\psi\cos\theta j + \sin\theta k \tag{10.42}$$

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

$$\frac{i'_y}{i'_x} = \frac{\sin\psi\cos\theta}{\cos\psi\cos\theta} = \tan\psi \iff \psi = \arctan\left(\frac{i'_y}{i'_x}\right) \tag{10.43}$$

$$i'_z = \sin\theta \iff \theta = \arcsin i'_z \tag{10.44}$$

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 10.5.

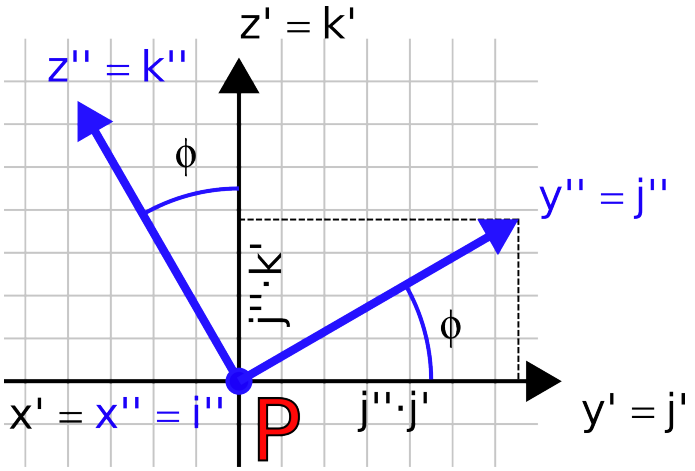

Figura 10.5: Tercera rotación en el sistema de coordenadas local sobre el punto P de la esfera

El estado del sistema de coordenadas tras la tercera rotación es el que va a ser dado por el cuaternión 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 (10.42) 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:

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

Ahora se puede calcular j':

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

$$j' = \frac{1}{\cos\theta}\left(-i''_y i + i''_x j\right) \tag{10.45}$$

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

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

En resumen:

$$\begin{cases} i' = i''_x i + i''_y j + i''_z k \\ \\ j' = \dfrac{1}{\cos\theta}\left(-i''_y i + i''_x j\right) \\ \\ k' = i' \times j' \end{cases} \tag{10.46}$$

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

$$\tan\phi = \frac{\sin\phi}{\cos\phi} = \frac{j'' \cdot k'}{j'' \cdot j'} \quad \Longleftrightarrow \quad \phi = \arctan\frac{j'' \cdot k'}{j'' \cdot j'} \tag{10.47}$$

El algoritmo de cálculo de los ángulos de Euler puede resumirse en los siguientes pasos, conocido el cuaternión de orientación del sensor q_s [2]:

10.4.1. Ampliación del intervalo de los ángulos obtenidos

El algoritmo presentado en el cuadro 10.2 da una solución para el ángulo θ dentro del intervalo $\left(\frac{-\pi}{2}, \frac{\pi}{2}\right)$. Puede pasar que la articulación del brazo robótico cuyo eje se refiera al ángulo θ se mueva en un intervalo más amplio, y por ello resultará de interés poder obtener soluciones de θ que no se restrinjan al intervalo $\left(\frac{-\pi}{2}, \frac{\pi}{2}\right)$. En los apartados siguientes se dará un par de posibles soluciones a este problema.

[2] A la hora de implementar el algorimo, se utilizará la función atan2 en lugar de arctan

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

Cuadro 10.2: Algoritmo de cálculo de los ángulos de euler a partir del cuaternión de orientación.

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 (figura 10.6).

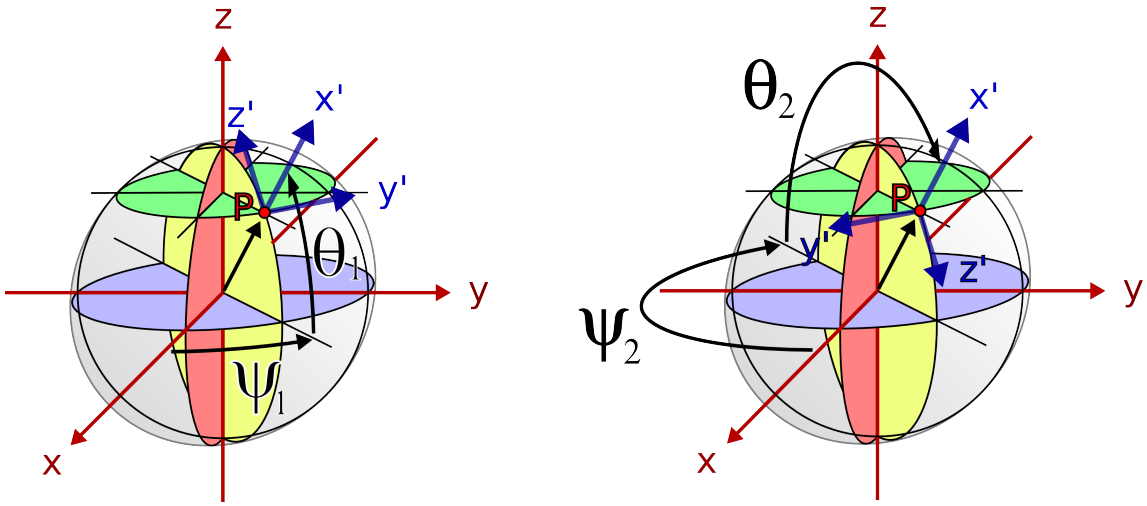

Figura 10.6: Dos soluciones para θ (*pitch*) y ψ (*yaw*)

Capítulo 11

IMPLEMENTACIÓN DEL SOFTWARE

11.0.2. Visualizador de la posición del brazo

Obtención de los ángulos de rotación entre cada segmento del brazo

Cálculo de las posiciones de cada segmento del brazo

Implementación

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

11.0.4. Controlador del brazo robótico del robot Youbot real

Capítulo 12

RESULTADOS EXPERIMENTALES

Capítulo 13

CONCLUSIONES

Capítulo 14

BIBLIOGRAFÍA

Parte II

Anexos

Apéndice A

INSTALACIÓN Y PUESTA EN MARCHA DEL SOFTWARE

Apéndice B

Instalación y configuración del software necesario

B.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.

Para realizar la instalación de ROS se partirá de una instalación previa de Ubuntu, pudiendo ser éste de cualquiera de estas *releases*:

- 10.04 LTS (Lucid Lynx)
- 11.04 (Oneiric Ocelot)
- 12.04 LTS (Precise Pangolin)

B.1.1. Configuración de los repositorios de Ubuntu

Se procederá a abrir el Centro de Sofware 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:

B.1.2. Configuración del archivo sources.list

El archivo sorces.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 que dependerá de la versión de Ubuntu que tengamos instalada:

Ubuntu 10.04 (Lucid)

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

Ubuntu 11.10 (Oneiric)

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

Ubuntu 12.04 (Precise)

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

Este comando lo que hace es crear 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 tengamos.

B.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 -
```

B.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 el terminal se ejecutará el siguiente comando:

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

Esta instalación traerá consigo las siguientes herramientas, entre 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

B.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
```

B.1.6. Otras herramientas

Se instalarán dos herramientas que permitirán obtener los paquetes necesarios para obtener los datos de los sensores XSENS MTi-G. Para ello, en el terminal se ejecutará el siguiente comando:

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

B.2. Instalación del simulador Gazebo

Para instalar la versión de Gazebo preparada para comunicarse con ROS se ejecutará el siguiente comando:

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

Apéndice C

CÓDIGO FUENTE

C.1. Driver Xsens

C.1.1. xsens_node.cpp

```
/*
 * Programa que publica en topics de ROS
 * los datos obtenidos de los sensores Xsens
 *
 * Autor: Daniel Fernández Villanueva
 * Mayo de 2013
 *
 */

#include <iostream>
#include <cmath>
#include <xsens_driver/xsens_driver.h>
#include <ros/ros.h>
#include <geometry_msgs/Vector3Stamped.h>
#include <geometry_msgs/QuaternionStamped.h>
#include <std_msgs/Float64MultiArray.h>
#include <dfv/dfv.h>
#include <xsens_driver/utils.h>

int main(int argc, char** argv)
{
    // Declaración de un objeto driver.
    // Valores por defecto:
    // OutputMode: CMT_OUTPUTMODE_CALIB | CMT_OUTPUTMODE_ORIENT
    // OutputSettings: CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION
    xsens::Driver driver;

    // Aquí podemos cambiar la configuración del sensor

    // Número de dispositivos detectados (sin contar el Xbus Master)
    ROS_INFO("Detected sensor count: %d", driver.GetMtCount());

    // Asignamos a los sensores una matriz de pre-rotación unitaria
    for(unsigned int i = 0; i < driver.GetMtCount(); ++i)
    {
        driver.SetAlignmentMatrix(i, xsens::DfvToCmtMatrix(dfv::Matrix::Identity(3)));
    }

    // Ejemplo para cambiar el modo de salida del sensor
    // para que nos de la matriz de rotación en lugar
    // del cuaternión de orientación:

```

```
43      //driver.SetOutputSettings(CMT_OUTPUTSETTINGS_ORIENTMODE_EULER);
44
45      // Inicializamos el driver. Esto realizará la configuración del sensor
46      // con los valores que le hayamos asignado hasta ahora
47      // y lo pondrá en modo de medida
48      if(driver.Initialize() == false)
49      {
50          std::cout << "ERROR: No Xsens IMUs found. Quitting..." << std::endl;
51          return -1;
52      }
53
54      // Inicialización de ROS
55      ROS_INFO("Initializing ROS...");
56      ros::init(argc, argv, "xsens_node");
57      ros::NodeHandle node_handle("~");
58
59      // Asignamos valor a algunos parámetros
60      node_handle.setParam("sensor_count", (int)driver.GetMtCount());
61      node_handle.setParam("output_mode", (int)driver.GetOutputMode());
62      node_handle.setParam("output_settings", (int)driver.GetOutputSettings());
63
64      // Creamos un NodeHandle para cada sensor
65      std::vector<ros::NodeHandle> sensor_node_handles(driver.GetMtCount());
66      for(unsigned int i = 0; i < driver.GetMtCount(); i++)
67      {
68          std::stringstream ss;
69          ss << "sensor" << i;
70          sensor_node_handles[i] = ros::NodeHandle(node_handle, ss.str());
71      }
72
73
74      // Declaramos los publicadores
75      std::vector<ros::Publisher> acc_publishers(driver.GetMtCount());
76      std::vector<ros::Publisher> gyr_publishers(driver.GetMtCount());
77      std::vector<ros::Publisher> mag_publishers(driver.GetMtCount());
78
79      std::vector<ros::Publisher> raw_acc_publishers(driver.GetMtCount());
80      std::vector<ros::Publisher> raw_gyr_publishers(driver.GetMtCount());
81      std::vector<ros::Publisher> raw_mag_publishers(driver.GetMtCount());
82
83      std::vector<ros::Publisher> ori_quat_publishers(driver.GetMtCount());
84      std::vector<ros::Publisher> ori_matrix_publishers(driver.GetMtCount());
85      std::vector<ros::Publisher> ori_euler_publishers(driver.GetMtCount());
86
87      std::vector<ros::Publisher> pos_lla_publishers(driver.GetMtCount());
88
89      std::vector<ros::Publisher> gps_llh_publishers(driver.GetMtCount());
90      std::vector<ros::Publisher> gps_vel_publishers(driver.GetMtCount());
91
92      // Creamos los topics a publicar
93      for(unsigned int i = 0; i < driver.GetMtCount(); i++)
94      {
95          // Datos calibrados
96          if((driver.GetOutputMode() & CMT_OUTPUTMODE_CALIB) != 0)
97          {
98              acc_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("acc", 1000);
99              gyr_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("gyr", 1000);
100             mag_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("mag", 1000);
101         }
102
```

```
103         // Datos crudos
104         if((driver.GetOutputMode() & CMT_OUTPUTMODE_RAW) != 0)
105         {
106             raw_acc_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("raw_acc", 1000);
107             raw_gyr_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("raw_gyr", 1000);
108             raw_mag_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("raw_mag", 1000);
109         }
110
111         if((driver.GetOutputMode() & CMT_OUTPUTMODE_POSITION) != 0)
112         {
113             pos_lla_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("pos_lla", 1000);
114         }
115
116         if((driver.GetOutputMode() & CMT_OUTPUTMODE_GPSPVT_PRESSURE) != 0)
117         {
118             gps_llh_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("gps_llh", 1000);
119             gps_vel_publishers[i] =
                    sensor_node_handles[i].advertise<geometry_msgs::Vector3Stamped>("gps_vel", 1000);
120         }
121
122         // Datos de orientación
123         if((driver.GetOutputMode() & CMT_OUTPUTMODE_ORIENT) != 0)
124         {
125             // Cuaternión de orientación
126             if((driver.GetOutputSettings() & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                    CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION)
127             {
128                 ori_quat_publishers[i] =
                        sensor_node_handles[i].advertise<geometry_msgs::QuaternionStamped>("ori_quat",
                        1000);
129             }
130
131             // Matriz de orientación
132             if((driver.GetOutputSettings() & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                    CMT_OUTPUTSETTINGS_ORIENTMODE_MATRIX)
133             {
134                 ori_matrix_publishers[i] =
                        sensor_node_handles[i].advertise<std_msgs::Float64MultiArray>("ori_matrix",
                        1000);
135             }
136
137             // Ángulos de Euler
138             if((driver.GetOutputSettings() & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                    CMT_OUTPUTSETTINGS_ORIENTMODE_EULER)
139             {
140                 ori_euler_publishers[i] =
                        sensor_node_handles[i].advertise<std_msgs::Float64MultiArray>("ori_euler",
                        1000);
141             }
142         }
143     }
144
145     // Contador
146     int count = 0;
147
148     // Empezamos a publicar los datos
149     ROS_INFO("Now publishing data...");
150
```

```
151     while(driver.SpinOnce() && ros::ok())
152     {
153         for(unsigned int i = 0; i < driver.GetMtCount(); i++)
154         {
155             if((driver.GetOutputMode() & CMT_OUTPUTMODE_CALIB) != 0)
156             {
157                 geometry_msgs::Vector3Stamped msg;
158 
159                 msg =
                        xsens::ToVector3StampedMsg(xsens::CmtToDfvVector(driver.GetCalData(i).m_acc));
160                 msg.header.seq = count;
161                 acc_publishers[i].publish(msg);
162 
163                 msg =
                        xsens::ToVector3StampedMsg(xsens::CmtToDfvVector(driver.GetCalData(i).m_gyr));
164                 msg.header.seq = count;
165                 gyr_publishers[i].publish(msg);
166 
167                 msg =
                        xsens::ToVector3StampedMsg(xsens::CmtToDfvVector(driver.GetCalData(i).m_mag));
168                 msg.header.seq = count;
169                 mag_publishers[i].publish(msg);
170             }
171 
172             if((driver.GetOutputMode() & CMT_OUTPUTMODE_RAW) != 0)
173             {
174                 geometry_msgs::Vector3Stamped msg;
175 
176                 msg =
                        xsens::ToVector3StampedMsg(xsens::CmtToDfvShortVector(driver.GetRawData(i).m_acc));
177                 msg.header.seq = count;
178                 raw_acc_publishers[i].publish(msg);
179 
180                 msg =
                        xsens::ToVector3StampedMsg(xsens::CmtToDfvShortVector(driver.GetRawData(i).m_gyr));
181                 msg.header.seq = count;
182                 raw_gyr_publishers[i].publish(msg);
183 
184                 msg =
                        xsens::ToVector3StampedMsg(xsens::CmtToDfvShortVector(driver.GetRawData(i).m_mag));
185                 msg.header.seq = count;
186                 raw_mag_publishers[i].publish(msg);
187             }
188 
189             if((driver.GetOutputMode() & CMT_OUTPUTMODE_POSITION) != 0)
190             {
191                 geometry_msgs::Vector3Stamped msg;
192 
193                 msg =
                        xsens::ToVector3StampedMsg(xsens::CmtToDfvVector(driver.GetPositionLLA(i)));
194                 msg.header.seq = count;
195                 pos_lla_publishers[i].publish(msg);
196             }
197 
198             if((driver.GetOutputMode() & CMT_OUTPUTMODE_GPSPVT_PRESSURE) != 0)
199             {
200                 CmtGpsPvtData data = driver.GetGpsPvtData(i);
201 
202                 geometry_msgs::Vector3Stamped llh_msg;
203                 llh_msg.vector.x = (float)data.m_latitude;
204                 llh_msg.vector.y = (float)data.m_longitude;
205                 llh_msg.vector.z = (float)data.m_height;
206                 llh_msg.header.stamp = ros::Time::now();
```

```
207                 llh_msg.header.seq = count;
208                 gps_llh_publishers[i].publish(llh_msg);
209 
210                 geometry_msgs::Vector3Stamped vel_msg;
211                 vel_msg.vector.x = (float)data.m_veln;
212                 vel_msg.vector.y = (float)data.m_vele;
213                 vel_msg.vector.z = (float)data.m_veld;
214                 vel_msg.header.stamp = ros::Time::now();
215                 vel_msg.header.seq = count;
216                 gps_vel_publishers[i].publish(vel_msg);
217 
218             }
219 
220             // Datos de orientación
221             if((driver.GetOutputMode() & CMT_OUTPUTMODE_ORIENT) != 0)
222             {
223                 // Cuaternión de orientación
224                 if((driver.GetOutputSettings() & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                        CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION)
225                 {
226                     geometry_msgs::QuaternionStamped msg;
227                     msg.quaternion.w = driver.GetOriQuat(i).m_data[0];
228                     msg.quaternion.x = driver.GetOriQuat(i).m_data[1];
229                     msg.quaternion.y = driver.GetOriQuat(i).m_data[2];
230                     msg.quaternion.z = driver.GetOriQuat(i).m_data[3];
231                     msg.header.seq = count;
232                     msg.header.stamp = ros::Time::now();
233                     ori_quat_publishers[i].publish(msg);
234                 }
235 
236                 // Matriz de orientación
237                 if((driver.GetOutputSettings() & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                        CMT_OUTPUTSETTINGS_ORIENTMODE_MATRIX)
238                 {
239                     std_msgs::Float64MultiArray msg;
240                     msg.data.clear();
241                     msg.data.resize(9);
242                     msg.data[0] = driver.GetOriMatrix(i).m_data[0][0];
243                     msg.data[1] = driver.GetOriMatrix(i).m_data[0][1];
244                     msg.data[2] = driver.GetOriMatrix(i).m_data[0][2];
245                     msg.data[3] = driver.GetOriMatrix(i).m_data[1][0];
246                     msg.data[4] = driver.GetOriMatrix(i).m_data[1][1];
247                     msg.data[5] = driver.GetOriMatrix(i).m_data[1][2];
248                     msg.data[6] = driver.GetOriMatrix(i).m_data[2][0];
249                     msg.data[7] = driver.GetOriMatrix(i).m_data[2][1];
250                     msg.data[8] = driver.GetOriMatrix(i).m_data[2][2];
251                     ori_matrix_publishers[i].publish(msg);
252                 }
253 
254                 // Ángulos de Euler
255                 if((driver.GetOutputSettings() & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                        CMT_OUTPUTSETTINGS_ORIENTMODE_EULER)
256                 {
257                     std_msgs::Float64MultiArray msg;
258                     msg.data.clear();
259                     msg.data.resize(3);
260                     msg.data[0] = driver.GetOriEuler(i).m_roll;
261                     msg.data[1] = driver.GetOriEuler(i).m_pitch;
262                     msg.data[2] = driver.GetOriEuler(i).m_yaw;
263                     ori_euler_publishers[i].publish(msg);
264                 }
265             }
266         }
```

```
267
268         ++count;
269         ros::spinOnce();
270         //ros::Duration(0.1).sleep();
271
272
273     }
274
275     ROS_INFO("Finishing program...");
276
277     return 0;
278 }
```

C.1.2. xsens_driver.h

```
/*
 * Clase Driver encargada de la configuración
 * y toma de datos de los sensores Xsens
 *
 * Autor: Daniel Fernández Villanueva
 * Mayo de 2013
 *
 */

#ifndef XSENS_DRIVER_H
#define XSENS_DRIVER_H

#include <vector>
#include <sstream>

#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#include <ros/ros.h>

#include <xsens_driver/cmtdef.h>
#include <xsens_driver/xsens_time.h>
#include <xsens_driver/xsens_list.h>
#include <xsens_driver/cmtscan.h>
#include <xsens_driver/cmt3.h>
#include <xsens_driver/xsens_sensor.h>

namespace xsens
{

    class Driver
    {
        public:
            Driver();
            ~Driver();

            bool Initialize();

            void SetOutputMode(CmtOutputMode output_mode);
            CmtOutputMode GetOutputMode() const;

            void SetOutputSettings(CmtOutputSettings output_settings);
            CmtOutputSettings GetOutputSettings() const;

            void SetAlignmentMatrix(unsigned int sensor_index, CmtMatrix alignment_matrix);

            bool SpinOnce();
            bool RetrieveData();
            unsigned int GetMtCount();
            CmtOutputMode GetOutputMode();
            CmtOutputSettings GetOutputSettings();

            // functions for getting data
            CmtQuat&       GetOriQuat(int mt_index = 0);
            CmtMatrix&     GetOriMatrix(int mt_index = 0);
            CmtEuler&      GetOriEuler(int mt_index = 0);
            CmtRawData&    GetRawData(int mt_index = 0);
            CmtCalData&    GetCalData(int mt_index = 0);
            CmtVector&     GetPositionLLA(int mt_index = 0);
            CmtGpsPvtData& GetGpsPvtData(int mt_index = 0);
```

```
62
63              // Vector de sensores
64              std::vector<Sensor> v_sensors;
65
66          private:
67              Cmt3              cmt3;
68              unsigned int      mt_count;
69              CmtOutputMode     output_mode;
70              CmtOutputSettings output_settings;
71              short             skip_factor;
72              short             skip_factor_count;
73
74              Packet* lp_packet;
75
76              unsigned short        sample_data;
77
78              bool DoHardwareScan();
79              bool SetConfiguration();
80
81      };
82  };
83
84  #endif
```

C.1.3. xsens_driver.cpp

```
#include <xsens_driver/xsens_driver.h>

namespace xsens
{

    Driver::Driver():
        mt_count(0),
        output_mode(CMT_OUTPUTMODE_CALIB | CMT_OUTPUTMODE_ORIENT),
        output_settings(CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION |
            CMT_OUTPUTSETTINGS_TIMESTAMP_SAMPLECNT),
        skip_factor(10),
        skip_factor_count(0),
        lp_packet(NULL)
    {
        //this->output_settings |= CMT_OUTPUTSETTINGS_TIMESTAMP_SAMPLECNT;
        if(this->DoHardwareScan() == false)
        {
            ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__, __LINE__,
                __FILE__);
            this->cmt3.closePort();
        }
    }

    Driver::~Driver()
    {
        delete this->lp_packet;
        this->cmt3.closePort();
    }

    void Driver::SetOutputMode(CmtOutputMode output_mode)
    {
        this->output_mode = output_mode;
    }

    CmtOutputMode Driver::GetOutputMode() const
    {
        return this->output_mode;
    }

    void Driver::SetOutputSettings(CmtOutputSettings output_settings)
    {
        this->output_settings = (output_settings | CMT_OUTPUTSETTINGS_TIMESTAMP_SAMPLECNT);
    }

    CmtOutputSettings Driver::GetOutputSettings() const
    {
        //return (this->output_settings & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK);
        return this->output_settings;
    }

    void Driver::SetAlignmentMatrix(unsigned int sensor_index, CmtMatrix alignment_matrix)
    {
        this->v_sensors[sensor_index].alignment_matrix = alignment_matrix;
    }

    bool Driver::DoHardwareScan()
    {
        XsensResultValue  res;
        List<CmtPortInfo> port_info;
        unsigned long     port_count = 0;

```

```
60          ROS_INFO("Scanning for connected Xsens devices...");
61          xsens::cmtScanPorts(port_info);
62          port_count = port_info.length();
63          ROS_INFO("Scanning done");
64
65          if (port_count == 0)
66          {
67              ROS_ERROR("No motion trackers found");
68              return false;
69          }
70
71          for (int i = 0; i < (int)port_count; i++)
72          {
73              std::stringstream ss;
74              ss << "Using COM port " << port_info[i].m_portName << " at ";
75              switch (port_info[i].m_baudrate)
76              {
77                  case B9600:
78                      ss << "9k6";
79                      break;
80                  case B19200:
81                      ss << "19k2";
82                      break;
83                  case B38400:
84                      ss << "38k4";
85                      break;
86                  case B57600:
87                      ss << "57k6";
88                      break;
89                  case B115200:
90                      ss << "115k2";
91                      break;
92                  case B230400:
93                      ss << "230k4";
94                      break;
95                  case B460800:
96                      ss << "460k8";
97                      break;
98                  case B921600:
99                      ss << "921k6";
100                     break;
101                 default:
102                     ss << port_info[i].m_baudrate;
103             }
104             ss << " baud" << std::endl;
105             ROS_INFO("%s", ss.str().c_str());
106         }
107         ROS_INFO("Opening ports...");
108
109         // open the port which the device is connected to and connect at the device's baudrate.
110
111         for (int p = 0; p < (int)port_count; p++)
112         {
113             res = this->cmt3.openPort(port_info[p].m_portName,
114                                       port_info[p].m_baudrate);
115             if (res != XRV_OK)
116             {
117                 ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__,
                        __LINE__, __FILE__);
118                 return false;
119             }
120         }
121         std::cout << "Done" << std::endl;
```

```
122
123         // set the measurement timeout to 100 ms (default is 16 ms)
124
125         int timeout = 100;
126         res = this->cmt3.setTimeoutMeasurement(timeout);
127         if (res != XRV_OK)
128         {
129             ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__, __LINE__,
                    __FILE__);
130             return false;
131         }
132         ROS_INFO("Timeout set to %d ms", timeout);
133
134         // get the MT sensor count
135
136         ROS_INFO("Retrieving MT count (excluding attached Xbus Master(s))");
137         this->mt_count = this->cmt3.getMtCount();
138         ROS_INFO("MT count: %d", this->mt_count);
139
140         this->v_sensors.resize(this->mt_count);
141
142         // retrieve the device IDs
143
144         ROS_INFO("Retrieving MT device IDs");
145         for (unsigned int j = 0; j < this->mt_count; j++)
146         {
147             // res = this->cmt3.getDeviceId((unsigned char)(j+1), this->device_ids[j]);
148             res = this->cmt3.getDeviceId((unsigned char)(j+1), this->v_sensors[j].device_id);
149             if (res != XRV_OK)
150             {
151                 ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__,
                        __LINE__, __FILE__);
152                 return false;
153             }
154         }
155
156         return true;
157     }
158
159     bool Driver::SetConfiguration()
160     {
161         XsensResultValue res;
162
163         // set the sensor to config state
164
165         res = this->cmt3.gotoConfig();
166         if (res != XRV_OK)
167         {
168             ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__, __LINE__,
                    __FILE__);
169             ROS_ERROR("Could not go to configuration mode");
170             return false;
171         }
172
173         unsigned short sample_freq;
174         sample_freq = this->cmt3.getSampleFrequency();
175
176         // set the device output mode for the devices
177
178         if ((this->output_mode & CMT_OUTPUTMODE_ORIENT) == 0)
179         {
180             this->output_settings = 0;
181             this->output_settings |= CMT_OUTPUTSETTINGS_TIMESTAMP_SAMPLECNT;
```

```
182         }
183
184         ROS_INFO("Configuring your mode selection");
185         for (unsigned int i = 0; i < this->mt_count; i++)
186         {
187             CmtDeviceMode device_mode(this->output_mode,
188                                       this->output_settings,
189                                       sample_freq);
190             if ((this->v_sensors[i].device_id & 0xFFF00000) != 0x00500000)
191             {
192                         // not an MTi-G, remove all GPS related stuff
193                         device_mode.m_outputMode &= 0xFF0F;
194                 }
195                 res = this->cmt3.setDeviceMode(device_mode, true,
                        this->v_sensors[i].device_id);
196                 if (res != XRV_OK)
197                 {
198                     ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__,
                            __LINE__, __FILE__);
199                     return false;
200                 }
201         }
202
203         // Set aligment Matrix
204         for(unsigned int i = 0; i < this->mt_count; i++)
205         {
206             if(this->cmt3.setObjectAlignmentMatrix(this->v_sensors[i].alignment_matrix,
                    this->v_sensors[i].device_id) != XRV_OK)
207             {
208                 ROS_ERROR("Could not set alignment matrix for object %d", i);
209                 return false;
210             }
211             else
212             {
213                 ROS_INFO("Alignment matrix set for object %d to M", i);
214             }
215         }
216
217         res = this->cmt3.gotoMeasurement();
218         if (res != XRV_OK)
219         {
220             ROS_ERROR("ERROR: go to measurement");
221             return false;
222         }
223
224         return true;
225     }
226
227     bool Driver::Initialize()
228     {
229         /*if (this->DoHardwareScan() == false)
230         {
231             std::cout << "ERROR: DoHardwareScan()" << std::endl;
232             this->cmt3.closePort();
233             return false;
234         }*/
235
236         if (this->mt_count == 0)
237         {
238             //ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__, __LINE__,
                    __FILE__);
239             ROS_ERROR("No Imus found.");
240             this->cmt3.closePort();
```

```
241             return false;
242         }
243 
244         if (this->SetConfiguration() == false)
245         {
246             ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__, __LINE__,
                    __FILE__);
247             return false;
248         }
249 
250         this->lp_packet = new Packet((unsigned short)this->mt_count, this->cmt3.isXm());
251         ROS_INFO("Everything is OK. Retrieving data...");
252 
253         return true;
254     }
255 
256     bool Driver::SpinOnce()
257     {
258         XsensResultValue res = this->cmt3.waitForDataMessage(this->lp_packet);
259         if (res != XRV_OK)
260         {
261             if ((res == XRV_TIMEOUTNODATA) || (res == XRV_TIMEOUT))
262             {
263                 return true;
264             }
265 
266             delete this->lp_packet;
267             this->cmt3.closePort();
268             ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__, __LINE__,
                    __FILE__);
269             return false;
270         }
271 
272         this->sample_data = this->lp_packet->getSampleCounter();
273         if (this->RetrieveData() == false)
274         {
275             //std::cout << "ERROR: RetrieveData()" << std::endl;
276             ROS_ERROR("In function %s at line %d in file %s", __PRETTY_FUNCTION__, __LINE__,
                    __FILE__);
277             return false;
278         }
279 
280         return true;
281     }
282 
283     bool Driver::RetrieveData()
284     {
285         for (unsigned int i = 0; i < this->mt_count; i++)
286         {
287             if ((this->output_mode & CMT_OUTPUTMODE_RAW) != 0)
288             {
289                 this->v_sensors[i].raw_data.m_acc = this->lp_packet->getRawAcc(i);
290                 this->v_sensors[i].raw_data.m_gyr = this->lp_packet->getRawGyr(i);
291                 this->v_sensors[i].raw_data.m_mag = this->lp_packet->getRawMag(i);
292                 this->v_sensors[i].raw_data.m_temp = this->lp_packet->getRawTemp(i);
293                 continue;
294             }
295 
296             if ((this->output_mode & CMT_OUTPUTMODE_TEMP) != 0)
297             {
298                 this->v_sensors[i].temperature_data = this->lp_packet->getTemp(i);
299             }
300 
```

```
301             if ((this->output_mode & CMT_OUTPUTMODE_CALIB) != 0)
302             {
303                 this->v_sensors[i].calibrated_data = this->lp_packet->getCalData(i);
304             }
305
306             if ((this->output_mode & CMT_OUTPUTMODE_POSITION) != 0)
307                     {
308                             if (this->lp_packet->containsPositionLLA(i))
309                             {
310                                     //CmtVector positionLLA = this->lp_packet->getPositionLLA();
311                                     this->v_sensors[i].position_lla =
                                            this->lp_packet->getPositionLLA(i);
312                                     /*if (this->result_value != XRV_OK)
313                                     {
314                                             std::cout << "ERROR: get position LLA" << std::endl;
315                                     }*/
316
317                                     /*for (int i = 0; i < 2; i++)
318                                     {
319                                             double deg = positionLLA.m_data[i];
320                                             double min = (deg - (int)deg)*60;
321                                             double sec = (min - (int)min)*60;
322                                     }*/
323                             }
324                             else
325                             {
326                                 ROS_ERROR("In function %s at line %d in file %s",
                                        __PRETTY_FUNCTION__, __LINE__, __FILE__);
327                                 ROS_ERROR("No PositionLLA data available");
328                             }
329                     }
330
331                     if((this->output_mode & CMT_OUTPUTMODE_GPSPVT_PRESSURE) != 0)
332                     {
333                         if (this->lp_packet->containsGpsPvtData(i))
334                             {
335                                     this->v_sensors[i].gps_pvt_data =
                                            this->lp_packet->getGpsPvtData(i);
336                                     // ROS_INFO("Retrieving GPS pvt Data");
337                             }
338                             else
339                             {
340                                 ROS_ERROR("In function %s at line %d in file %s",
                                        __PRETTY_FUNCTION__, __LINE__, __FILE__);
341                                 ROS_ERROR("No GpsPvt data available");
342                             }
343                     }
344
345             if ((this->output_mode & CMT_OUTPUTMODE_ORIENT) == 0)
346                     {
347                             continue;
348                     }
349
350                     switch (this->output_settings & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK)
351                     {
352                             case CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION:
353                                     this->v_sensors[i].quaternion_data =
                                            this->lp_packet->getOriQuat(i);
354                                     break;
355                             case CMT_OUTPUTSETTINGS_ORIENTMODE_EULER:
356                                     this->v_sensors[i].euler_data =
                                            this->lp_packet->getOriEuler(i);
357                                     break;
```

```
358                         case CMT_OUTPUTSETTINGS_ORIENTMODE_MATRIX:
359                                 this->v_sensors[i].matrix_data =
                                        this->lp_packet->getOriMatrix(i);
360                                 break;
361                         default:
362                                 break;
363                 }
364
365         }
366
367         return true;
368     }
369
370     unsigned int Driver::GetMtCount()
371     {
372         return this->mt_count;
373     }
374
375     CmtOutputMode Driver::GetOutputMode()
376     {
377         return this->output_mode;
378     }
379
380     CmtOutputSettings Driver::GetOutputSettings()
381     {
382         return this->output_settings;
383     }
384
385     CmtQuat& Driver::GetOriQuat(int mt_index)
386     {
387         return this->v_sensors[mt_index].quaternion_data;
388     }
389
390     CmtMatrix& Driver::GetOriMatrix(int mt_index)
391     {
392         return this->v_sensors[mt_index].matrix_data;
393     }
394
395     CmtEuler& Driver::GetOriEuler(int mt_index)
396     {
397         return this->v_sensors[mt_index].euler_data;
398     }
399
400     CmtRawData& Driver::GetRawData(int mt_index)
401     {
402         return this->v_sensors[mt_index].raw_data;
403     }
404
405     CmtCalData& Driver::GetCalData(int mt_index)
406     {
407         return this->v_sensors[mt_index].calibrated_data;
408     }
409
410     CmtVector& Driver::GetPositionLLA(int mt_index)
411     {
412         return this->v_sensors[mt_index].position_lla;
413     }
414
415     CmtGpsPvtData& Driver::GetGpsPvtData(int mt_index)
416     {
417         return this->v_sensors[mt_index].gps_pvt_data;
418     }
419
```

```
420 }
```

C.1.4. xsens_sensor.h

```
/*
 * Clase Sensor en la que la clase Driver almacenará
 * los datos obtenidos por los sensores físicos Xsens,
 * además de ciertos parámetros de configuración
 *
 * Autor: Daniel Fernández Villanueva
 * Mayo de 2013
 *
 */

#ifndef XSENS_SENSOR_H
#define XSENS_SENSOR_H

#include <xsens_driver/cmtdef.h>

namespace xsens
{
    class Sensor
    {
        public:
            Sensor();
            ~Sensor();

            void SetAlignmentMatrix(const CmtMatrix& matrix);

            CmtMatrix         alignment_matrix;

        protected:
        private:
            CmtCalData        calibrated_data;
            CmtQuat           quaternion_data;
            CmtEuler          euler_data;
            CmtMatrix         matrix_data;
            CmtRawData        raw_data;
            CmtVector         position_lla;
            CmtGpsPvtData     gps_pvt_data;
            double            temperature_data;

            CmtDeviceId       device_id;

            friend class Driver;

    };
}

#endif
```

C.1.5. xsens_sensor.cpp

```
#include <xsens_driver/xsens_sensor.h>

namespace xsens
{
    Sensor::Sensor()
    {
        CmtMatrix matrix;
        matrix.m_data[0][0] = 1.0; matrix.m_data[0][1] = 0.0; matrix.m_data[0][2] = 0.0;
        matrix.m_data[1][0] = 0.0; matrix.m_data[1][1] = 1.0; matrix.m_data[1][2] = 0.0;
        matrix.m_data[2][0] = 0.0; matrix.m_data[2][1] = 0.0; matrix.m_data[2][2] = 1.0;
        this->alignment_matrix = matrix;
    }

    Sensor::~Sensor()
    {
    }

    void Sensor::SetAlignmentMatrix(const CmtMatrix& matrix)
    {
        this->alignment_matrix = matrix;
    }
}
```

C.1.6. xsens_sensor_subscriber.h

```
/*
 * Clases SensorSubscriber y SensorSubscriberList
 *
 * Estas clases no son utilizadas por el driver.
 * Forman parte de la librería xsens_driver, que
 * proporciona una interfaz sencilla para acceder
 * a los datos publicados en ROS por el driver en
 * otros programas.
 *
 * Autor: Daniel Fernández Villanueva
 * Mayo 2013
 *
 */

#ifndef XSENS_SENSOR_SUBSCRIBER_H
#define XSENS_SENSOR_SUBSCRIBER_H

#include <dfv/dfv.h>
#include <sstream>
#include <xsens_driver/cmtdef.h>
#include <ros/ros.h>
#include <std_msgs/Float64MultiArray.h>
#include <geometry_msgs/Vector3Stamped.h>
#include <geometry_msgs/QuaternionStamped.h>

namespace xsens
{
    class SensorSubscriber
    {
        public:
            SensorSubscriber(unsigned int mt_index_, ros::NodeHandle& node_handle_);
            ~SensorSubscriber();

            bool                    SubscribeToTopics();

            // Función que devuelve el vector aceleración
            const dfv::Vector3    GetAcc() const;

            // Función que devuelve el vector giróscopo
            const dfv::Vector3    GetGyr() const;

            // Función que devuelve el vector campo magnético
            const dfv::Vector3    GetMag() const;

            // Función que devuelve el cuaternión de orientación
            const dfv::Quaternion GetOriQuat() const;

            // Función que devuelve la matriz de orientación
            const dfv::Matrix     GetOriMatrix() const;

            // Función que devuelve un vector con los ángulos de Euler
            const dfv::Vector3    GetOriEuler() const;



        private:
            ros::NodeHandle&  node_handle;
            unsigned int      mt_index;

            std::string       acc_topic_name;
            std::string       gyr_topic_name;
```

```
62          std::string       mag_topic_name;
63          std::string       ori_quat_topic_name;
64          std::string       ori_matrix_topic_name;
65          std::string       ori_euler_topic_name;
66
67          CmtOutputMode     output_mode;
68          CmtOutputSettings output_settings;
69
70          dfv::Vector3      acc;
71          dfv::Vector3      gyr;
72          dfv::Vector3      mag;
73
74          dfv::Quaternion   ori_quat;
75          dfv::Matrix       ori_matrix;
76          dfv::Vector3      ori_euler;
77
78          dfv::Vector3      position_lla;
79          double            temperature;
80
81          ros::Subscriber   acc_subscriber;
82          ros::Subscriber   gyr_subscriber;
83          ros::Subscriber   mag_subscriber;
84          ros::Subscriber   ori_quat_subscriber;
85          ros::Subscriber   ori_matrix_subscriber;
86          ros::Subscriber   ori_euler_subscriber;
87
88          void              AccSubCallback(const geometry_msgs::Vector3Stamped::ConstPtr& msg);
89          void              GyrSubCallback(const geometry_msgs::Vector3Stamped::ConstPtr& msg);
90          void              MagSubCallback(const geometry_msgs::Vector3Stamped::ConstPtr& msg);
91          void              OriQuatSubCallback(const
                geometry_msgs::QuaternionStamped::ConstPtr& msg);
92          void              OriMatrixSubCallback(const std_msgs::Float64MultiArray::ConstPtr&
                msg);
93          void              OriEulerSubCallback(const std_msgs::Float64MultiArray::ConstPtr&
                msg);
94      };
95
96      class SensorSubscriberList
97      {
98          public:
99              SensorSubscriberList(ros::NodeHandle& node_handle_);
100             ~SensorSubscriberList();
101
102             // Función que devuelve el número de sensores detectados
103             unsigned int GetMtCount() const;
104
105             // Función que devuelve el vector aceleración
106             const dfv::Vector3    GetAcc(unsigned int mt_index) const;
107
108             // Función que devuelve el vector giróscopo
109             const dfv::Vector3    GetGyr(unsigned int mt_index) const;
110
111             // Función que devuelve el vector campo magnético
112             const dfv::Vector3    GetMag(unsigned int mt_index) const;
113
114             // Función que devuelve el cuaternión de orientación
115             const dfv::Quaternion GetOriQuat(unsigned int mt_index) const;
116
117             // Función que devuelve la matriz de orientación
118             const dfv::Matrix     GetOriMatrix(unsigned int mt_index) const;
119
120             // Función que devuelve un vector con los ángulos de Euler
121             const dfv::Vector3    GetOriEuler(unsigned int mt_index) const;
```

```
122
123         private:
124             ros::NodeHandle node_handle;
125             unsigned int mt_count;
126             SensorSubscriber** sensors;
127
128     };
129 }
130
131 #endif
```

C.1.7. xsens_sensor_subscriber.cpp

```
#include <xsens_driver/xsens_sensor_subscriber.h>

namespace xsens
{
    SensorSubscriber::SensorSubscriber(unsigned int mt_index_, ros::NodeHandle& node_handle_):
        node_handle(node_handle_), mt_index(mt_index_)
    {
        std::stringstream ss;
        ss << "/xsens_node/sensor" << this->mt_index << "/acc";
        this->acc_topic_name = ss.str();

        ss.str(std::string());
        ss << "/xsens_node/sensor" << this->mt_index << "/gyr";
        this->gyr_topic_name = ss.str();

        ss.str(std::string());
        ss << "/xsens_node/sensor" << this->mt_index << "/mag";
        this->mag_topic_name = ss.str();

        ss.str(std::string());
        ss << "xsens_node/sensor" << this->mt_index << "/ori_quat";
        this->ori_quat_topic_name = ss.str();

        ss.str(std::string());
        ss << "/xsens_node/sensor" << this->mt_index << "/ori_matrix";
        this->ori_matrix_topic_name = ss.str();

        ss.str(std::string());
        ss << "/xsens_node/sensor" << this->mt_index << "/ori_euler";
        this->ori_euler_topic_name = ss.str();

        int param;
        this->node_handle.param<int>("/xsens_node/output_mode", param, 0);
        this->output_mode = param;
        this->node_handle.param<int>("/xsens_node/output_settings", param, 0);
        this->output_settings = param;

        this->SubscribeToTopics();

    }

    SensorSubscriber::~SensorSubscriber()
    {

    }

    bool SensorSubscriber::SubscribeToTopics()
    {
        if((this->output_mode & CMT_OUTPUTMODE_CALIB) != 0)
        {
            ROS_INFO("[SensorSubscriber] Subscribing to calibrated data topics...");
            this->acc_subscriber = this->node_handle.subscribe(this->acc_topic_name,
                                                               1,
                                                               &SensorSubscriber::AccSubCallback,
                                                               this);
            this->gyr_subscriber = this->node_handle.subscribe(this->gyr_topic_name,
                                                               1,
                                                               &SensorSubscriber::GyrSubCallback,
                                                               this);
            this->mag_subscriber = this->node_handle.subscribe(this->mag_topic_name,
                                                               1,
```

```
62                                                  &SensorSubscriber::MagSubCallback,
63                                                  this);
64          }
65
66          if((this->output_mode & CMT_OUTPUTMODE_ORIENT) != 0)
67          {
68              if((this->output_settings & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                    CMT_OUTPUTSETTINGS_ORIENTMODE_QUATERNION)
69              {
70                  ROS_INFO("[SensorSubscriber] Subscribing to ori_quat topic...");
71                  this->ori_quat_subscriber =
                        this->node_handle.subscribe(this->ori_quat_topic_name,
72                                                    1,
73                                                    &SensorSubscriber::OriQuatSubCallback,
74                                                    this);
75              }
76
77              if((this->output_settings & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                    CMT_OUTPUTSETTINGS_ORIENTMODE_MATRIX)
78              {
79                  ROS_INFO("[SensorSubscriber] Subscribing to ori_matrix topic...");
80                  this->ori_matrix_subscriber =
                        this->node_handle.subscribe(this->ori_matrix_topic_name,
81                                                    1,
82                                                    &SensorSubscriber::OriMatrixSubCallback
83                                                    this);
84              }
85
86              if((this->output_settings & CMT_OUTPUTSETTINGS_ORIENTMODE_MASK) ==
                    CMT_OUTPUTSETTINGS_ORIENTMODE_EULER)
87              {
88                  ROS_INFO("[SensorSubscriber] Subscribing to ori_euler topic...");
89                  this->ori_euler_subscriber =
                        this->node_handle.subscribe(this->ori_euler_topic_name,
90                                                    1,
91                                                    &SensorSubscriber::OriEulerSubCallback,
92                                                    this);
93              }
94          }
95
96          return true;
97      }
98
99      const dfv::Vector3 SensorSubscriber::GetAcc() const
100     {
101         return dfv::Vector3(this->acc);
102     }
103
104     const dfv::Vector3 SensorSubscriber::GetGyr() const
105     {
106         return dfv::Vector3(this->gyr);
107     }
108
109     const dfv::Vector3 SensorSubscriber::GetMag() const
110     {
111         return dfv::Vector3(this->mag);
112     }
113
114     const dfv::Quaternion SensorSubscriber::GetOriQuat() const
115     {
116         return dfv::Quaternion(this->ori_quat);
117     }
118
```

```
119     const dfv::Matrix SensorSubscriber::GetOriMatrix() const
120     {
121         return dfv::Matrix(this->ori_matrix);
122     }
123 
124     const dfv::Vector3 SensorSubscriber::GetOriEuler() const
125     {
126         return dfv::Vector3(this->ori_euler);
127     }
128 
129     void SensorSubscriber::AccSubCallback(const geometry_msgs::Vector3Stamped::ConstPtr& msg)
130     {
131         this->acc = dfv::Vector3(msg->vector.x, msg->vector.y, msg->vector.z);
132     }
133 
134     void SensorSubscriber::GyrSubCallback(const geometry_msgs::Vector3Stamped::ConstPtr& msg)
135     {
136         this->gyr = dfv::Vector3(msg->vector.x, msg->vector.y, msg->vector.z);
137     }
138 
139     void SensorSubscriber::MagSubCallback(const geometry_msgs::Vector3Stamped::ConstPtr& msg)
140     {
141         this->mag = dfv::Vector3(msg->vector.x, msg->vector.y, msg->vector.z);
142     }
143 
144     void SensorSubscriber::OriQuatSubCallback(const geometry_msgs::QuaternionStamped::ConstPtr&
            msg)
145     {
146         this->ori_quat = dfv::Quaternion(msg->quaternion.w, msg->quaternion.x,
                msg->quaternion.y, msg->quaternion.z);
147     }
148 
149     void SensorSubscriber::OriMatrixSubCallback(const std_msgs::Float64MultiArray::ConstPtr&
            msg)
150     {
151         dfv::Matrix m(3);
152         m.Set(0, 0 , msg->data[0]);
153         m.Set(0, 1 , msg->data[1]);
154         m.Set(0, 2 , msg->data[2]);
155         m.Set(1, 0 , msg->data[3]);
156         m.Set(1, 1 , msg->data[4]);
157         m.Set(1, 2 , msg->data[5]);
158         m.Set(2, 0 , msg->data[6]);
159         m.Set(2, 1 , msg->data[7]);
160         m.Set(2, 2 , msg->data[8]);
161         this->ori_matrix = m;
162     }
163 
164     void SensorSubscriber::OriEulerSubCallback(const std_msgs::Float64MultiArray::ConstPtr& msg)
165     {
166         this->ori_euler = dfv::Vector3(msg->data[0], msg->data[1], msg->data[2]);
167     }
168 
169 // ========================================== //
170 //         Clase SensorSubscriberList       //
171 // ========================================== //
172 
173     SensorSubscriberList::SensorSubscriberList(ros::NodeHandle& node_handle_):
174         node_handle(node_handle_)
175     {
176         int param;
177         this->node_handle.param<int>("/xsens_node/sensor_count", param, 0);
178         this->mt_count = param;
```

```
179         this->sensors = new SensorSubscriber*[this->mt_count];
180 
181         for(int i = 0; i < param; ++i)
182         {
183             this->sensors[i] = new SensorSubscriber(i, this->node_handle);
184         }
185     }
186 
187     SensorSubscriberList::~SensorSubscriberList()
188     {
189         for(int i = 0; i < this->mt_count; ++i)
190         {
191             delete this->sensors[i];
192         }
193         delete this->sensors;
194     }
195 
196     unsigned int SensorSubscriberList::GetMtCount() const
197     {
198         return this->mt_count;
199     }
200 
201     const dfv::Vector3 SensorSubscriberList::GetAcc(unsigned int mt_index) const
202     {
203         return dfv::Vector3(this->sensors[mt_index]->GetAcc());
204     }
205 
206     const dfv::Vector3 SensorSubscriberList::GetGyr(unsigned int mt_index) const
207     {
208         return dfv::Vector3(this->sensors[mt_index]->GetGyr());
209     }
210 
211     const dfv::Vector3 SensorSubscriberList::GetMag(unsigned int mt_index) const
212     {
213         return dfv::Vector3(this->sensors[mt_index]->GetMag());
214     }
215 
216     const dfv::Quaternion SensorSubscriberList::GetOriQuat(unsigned int mt_index) const
217     {
218         return dfv::Quaternion(this->sensors[mt_index]->GetOriQuat());
219     }
220 
221     const dfv::Matrix SensorSubscriberList::GetOriMatrix(unsigned int mt_index) const
222     {
223         return dfv::Matrix(this->sensors[mt_index]->GetOriMatrix());
224     }
225 
226     const dfv::Vector3 SensorSubscriberList::GetOriEuler(unsigned int mt_index) const
227     {
228         return dfv::Vector3(this->sensors[mt_index]->GetOriEuler());
229     }
230 
231 }
```

C.1.8. utils.h

```
#ifndef XSENS_DRIVER_UTILS_H
#define XSENS_DRIVER_UTILS_H

#include <xsens_driver/cmtdef.h>
#include <dfv/dfv.h>

namespace xsens
{
    const CmtMatrix     DfvToCmtMatrix(const dfv::Matrix& m);
    const dfv::Matrix   CmtToDfvMatrix(const CmtMatrix& m);
    const CmtVector     DfvToCmtVector(const dfv::Vector3& v);
    const dfv::Vector3  CmtToDfvVector(const CmtVector& v);
    const CmtShortVector DfvToCmtShortVector(const dfv::Vector3& v);
    const dfv::Vector3  CmtToDfvShortVector(const CmtShortVector& v);


    const geometry_msgs::Vector3 ToVector3Msg(const dfv::Vector3& v);
    const geometry_msgs::Vector3Stamped ToVector3StampedMsg(const dfv::Vector3& v);
}

#endif
```

C.1.9. utils.cpp

```
#include <xsens_driver/utils.h>

namespace xsens
{
    const CmtMatrix DfvToCmtMatrix(const dfv::Matrix& m)
    {
        CmtMatrix result;
        if(m.GetRows() == 3 && m.GetColumns() == 3)
        {
            for(unsigned int j = 0; j < m.GetRows(); ++j)
            {
                for(unsigned int i = 0; i < m.GetColumns(); ++i)
                {
                    result.m_data[j][i] = m.Get(j, i);
                }
            }
        }
        return result;
    }

    const dfv::Matrix CmtToDfvMatrix(const CmtMatrix& m)
    {
        dfv::Matrix result(3, 3);
        for(unsigned int j = 0; j < 3; ++j)
        {
            for(unsigned int i = 0; i < 3; ++i)
            {
                result.Set(j, i, m.m_data[j][i]);
            }
        }
        return result;
    }

    const CmtVector DfvToCmtVector(const dfv::Vector3& v)
    {
        CmtVector result;
        result.m_data[0] = v.x;
        result.m_data[1] = v.y;
        result.m_data[2] = v.z;
        return result;
    }

    const dfv::Vector3 CmtToDfvShortVector(const CmtShortVector& v)
    {
        return dfv::Vector3(v.m_data[0], v.m_data[1], v.m_data[2]);
    }

    const CmtShortVector DfvToCmtShortVector(const dfv::Vector3& v)
    {
        CmtShortVector result;
        result.m_data[0] = v.x;
        result.m_data[1] = v.y;
        result.m_data[2] = v.z;
        return result;
    }

    const dfv::Vector3 CmtToDfvVector(const CmtVector& v)
    {
        return dfv::Vector3(v.m_data[0], v.m_data[1], v.m_data[2]);
    }

```

```
62      const geometry_msgs::Vector3 ToVector3Msg(const dfv::Vector3& v)
63      {
64          geometry_msgs::Vector3 msg;
65          msg.x = v.x;
66          msg.y = v.y;
67          msg.z = v.z;
68          return msg;
69      }
70
71      const geometry_msgs::Vector3Stamped ToVector3StampedMsg(const dfv::Vector3& v)
72      {
73          geometry_msgs::Vector3Stamped msg;
74          msg.header.stamp = ros::Time::now();
75          msg.vector.x = v.x;
76          msg.vector.y = v.y;
77          msg.vector.z = v.z;
78          return msg;
79      }
80  }
```

C.2. Librería dfv

C.2.1. quaternion.h

```
/* Clase Quaternion
 * Incluye operaciones para creación y manipulación
 * de cuaterniones de orientación
 *
 * Autor: Daniel Fernández Villanueva
 */

#ifndef Quaternion_H
#define Quaternion_H

#include <iostream>
#include <cmath>
#include <sstream>
#include <dfv/vector3.h>
#include <dfv/utils.h>
#include <tf/transform_datatypes.h>


namespace dfv
{
    class Vector3;

    class Quaternion
    {
        public:

            Quaternion();
            Quaternion(double w_, double x_, double y_, double z_);
            explicit Quaternion(const Vector3& v);
            ~Quaternion();

            // ******** Operador de asignación ******** //
            Quaternion& operator=(const Quaternion& q);

            // ******** Operadores de asignación compuestos ******** //
            Quaternion& operator+=(const Quaternion& q);
            Quaternion& operator-=(const Quaternion& q);
            Quaternion& operator*=(const double k);

            // ******** Operadores aritméticos binarios ******** //
            const Quaternion operator+(const Quaternion& q) const;
            const Quaternion operator-(const Quaternion& q) const;
            friend const Quaternion operator*(double k, Quaternion& q);
            const Quaternion operator*(double k) const;
            // Producto de Hamilton:
            const Quaternion operator*(const Quaternion& q) const;

            // ******** Operadores de comparación ******** //
            bool operator==(const Quaternion& q) const;
            bool operator!=(const Quaternion& q) const;

            // Función que devuelve una representación
            // del cuaternión como texto:
            std::string          ToString() const;

            // Función que devuelve el módulo del cuaternión:
            double               GetModulus() const;

            // Función que normaliza el cuaternión:
```

```
60          void                Normalize();
61
62          // Función que devuelve el conjugado del cuaternión:
63          const Quaternion    GetConjugate() const;
64
65          // Función que devuelve el cuaternión de rotación
66          // definido por el eje [axisx_, axisy_, axisz_]
67          // y el ángulo angle_:
68          static const Quaternion GetRotationQuaternion(const double axisx_,
69                                                        const double axisy_,
70                                                        const double axisz_,
71                                                        const double angle_);
72
73          // Función que devuelve el cuaternión de rotación
74          // definido por el eje axis_
75          // y el ángulo angle_:
76          static const Quaternion GetRotationQuaternion(const Vector3& axis_,
77                                                        const double angle_);
78
79          // Función que devuelve el cuaternión de rotación
80          // definido por el eje perpendicular a v_before y v_after:
81          static const Quaternion GetRotationQuaternion(const Vector3& v_before,
82                                                        const Vector3& v_after);
83
84          // Función que devuelve el cuaternión de rotación
85          // del conjunto de vectores v1 y v2
86          // Se supone que v1 y v2 son perpendiculares entre sí:
87          static const Quaternion GetRotationQuaternion(const Vector3& v1_before,
88                                                        const Vector3& v1_after,
89                                                        const Vector3& v2_before,
90                                                        const Vector3& v2_after);
91
92
93          // Función que descompone el cuaternión en otros tres cuaterniones
94          // definidos por los ejes v1, v2 y v3, y unos valores iniciales
95          // de los ángulos angle_1, angle_2 y angle_3
96          // Los valores finales de angle_1, angle_2 y angle_3 son
97          // los resultados obtenidos para los ángulos asociados
98          // a los cuaterniones.
99          // Los vectores v1, v2 y v3 tienen que ser linearmente
100         // independientes para asegurar la obtención de un resultado
101         // correcto.
102         void                Decompose(double& angle_1,
103                                       double& angle_2,
104                                       double& angle_3,
105                                       const Vector3& v1,
106                                       const Vector3& v2,
107                                       const Vector3& v3) const;
108
109         // Función para obtener el eje y el ángulo asociados al
110         // cuaternión:
111         void                GetAxisAndAngle(Vector3& vector,
112                                             double& angle) const;
113
114         // Función para obtener el roll, pitch y yaw del cuaternión
115         void                GetRPY(double& roll, double& pitch, double& yaw, unsigned int
                solution = 1);
116
117         // Función que devuelve el cuaternión de diferencia entre dos cuaterniones
118         static const Quaternion GetDifference(const Quaternion& q1, const Quaternion& q2);
119
120         // Quaterniones unitarios colineales a las componentes w, x, y, z:
121         static const Quaternion identity;
```

```
122             static const Quaternion i;
123             static const Quaternion j;
124             static const Quaternion k;
125 
126             // Componentes del cuaternión:
127             double w;
128             double x;
129             double y;
130             double z;
131 
132         protected:
133         private:
134 
135 
136     };
137 
138     std::ostream& operator<<(std::ostream& os, const Quaternion& q);
139 
140 }
141 
142 #endif // Quaternion_H
```

C.2.2. quaternion.cpp

```
#include "dfv/quaternion.h"

namespace dfv
{

    Quaternion::Quaternion(): w(0), x(0), y(0), z(0)
    {
        //ctor
    }

    Quaternion::Quaternion(double w_, double x_, double y_, double z_): w(w_), x(x_), y(y_),
         z(z_)
    {

    }

    Quaternion::Quaternion(const Vector3& v):
        w(0.0), x(v.x), y(v.y), z(v.z)
    {

    }

    Quaternion::~Quaternion()
    {
        //dtor
    }

    Quaternion& Quaternion::operator=(const Quaternion& q)
    {
        if(this != &q)
        {
            this->w = q.w;
            this->x = q.x;
            this->y = q.y;
            this->z = q.z;
        }

        return *this;
    }

    Quaternion& Quaternion::operator+=(const Quaternion& q)
    {
        this->w += q.w;
        this->x += q.x;
        this->y += q.y;
        this->z += q.z;

        return *this;
    }

    Quaternion& Quaternion::operator-=(const Quaternion& q)
    {
        this->w -= q.w;
        this->x -= q.x;
        this->y -= q.y;
        this->z -= q.z;

        return *this;
    }

    Quaternion& Quaternion::operator*=(const double k)
```

```
61      {
62          this->w *= k;
63          this->x *= k;
64          this->y *= k;
65          this->z *= k;
66
67          return *this;
68      }
69
70      const Quaternion Quaternion::operator+(const Quaternion& q) const
71      {
72          return Quaternion(*this) += q;
73      }
74
75      const Quaternion Quaternion::operator-(const Quaternion& q) const
76      {
77          return Quaternion(*this) -= q;
78      }
79
80      const Quaternion operator*(double k, Quaternion& q)
81      {
82          return Quaternion(q) *= k;
83      }
84
85      const Quaternion Quaternion::operator*(double k) const
86      {
87          return Quaternion(*this) *= k;
88      }
89
90      const Quaternion Quaternion::operator*(const Quaternion& q) const
91      {
92          return Quaternion(this->w*q.w - this->x*q.x - this->y*q.y - this->z*q.z,
93                            this->x*q.w + this->w*q.x - this->z*q.y + this->y*q.z,
94                            this->y*q.w + this->z*q.x + this->w*q.y - this->x*q.z,
95                            this->z*q.w - this->y*q.x + this->x*q.y + this->w*q.z);
96      }
97
98      bool Quaternion::operator==(const Quaternion& q) const
99      {
100         return (this->w == q.w) && (this->x == q.x) && (this->y == q.y) && (this->z == q.z);
101     }
102
103     bool Quaternion::operator!=(const Quaternion& q) const
104     {
105         return !(*this == q);
106     }
107
108     std::string Quaternion::ToString() const
109     {
110         std::stringstream ss;
111         ss << this->w << " + " << this->x << "*i + " << this->y << "*j + " << this->z << "*k";
112         return ss.str();
113     }
114
115     double Quaternion::GetModulus() const
116     {
117         return sqrt(pow(this->w, 2) + pow(this->x, 2) + pow(this->y, 2) + pow(this->z, 2));
118     }
119
120     void Quaternion::Normalize()
121     {
122         double l = this->GetModulus();
123         if(l == 0.f)
```

```
124         {
125             this->w = 0.f;
126             this->x = 0.f;
127             this->y = 0.f;
128             this->z = 0.f;
129         }
130         else
131         {
132             this->w /= l;
133             this->x /= l;
134             this->y /= l;
135             this->z /= l;
136         }
137     }
138
139     const Quaternion Quaternion::GetConjugate() const
140     {
141         return Quaternion(this->w, -this->x, -this->y, -this->z);
142     }
143
144     const Quaternion Quaternion::GetRotationQuaternion(const double axisx_,
145                                                        const double axisy_,
146                                                        const double axisz_,
147                                                        const double angle_)
148     {
149         double l = sqrt(axisx_*axisx_ + axisy_*axisy_ + axisz_*axisz_);
150         return Quaternion(cos(angle_ / 2.0),
151                           axisx_/l * sin(angle_ / 2.0),
152                           axisy_/l * sin(angle_ / 2.0),
153                           axisz_/l * sin(angle_ / 2.0));
154     }
155
156     const Quaternion Quaternion::GetRotationQuaternion(const Vector3& axis_,
157                                                        const double angle_)
158     {
159         double l = axis_.GetMagnitude();
160         return Quaternion(cos(angle_ / 2.0),
161                           axis_.x/l * sin(angle_ / 2.0),
162                           axis_.y/l * sin(angle_ / 2.0),
163                           axis_.z/l * sin(angle_ / 2.0));
164     }
165
166     const Quaternion Quaternion::GetRotationQuaternion(const Vector3& v_before,
167                                                        const Vector3& v_after)
168     {
169         Vector3 vu = v_before.GetNormalized();
170         Vector3 vvu = v_after.GetNormalized();
171         if(vu == vvu)
172         {
173             return Quaternion(1.0, 0.0, 0.0, 0.0);
174         }
175         Vector3 r = (vu ^ vvu).GetNormalized();
176         if(r == Vector3(0.0, 0.0, 0.0))
177         {
178             r = ((vu ^ Vector3::i) + (vu ^ Vector3::j)).GetNormalized();
179             return Quaternion::GetRotationQuaternion(r, pi);
180         }
181         else
182         {
183             double sin_theta = (vu ^ vvu).GetMagnitude();
184             double cos_theta = vu * vvu;
185             double theta = atan2(sin_theta, cos_theta);
186             return Quaternion::GetRotationQuaternion(r, theta);
```

```
187         }
188
189     }
190
191     const Quaternion Quaternion::GetRotationQuaternion(const Vector3& v1_before,
192                                                        const Vector3& v1_after,
193                                                        const Vector3& v2_before,
194                                                        const Vector3& v2_after)
195     {
196         Quaternion q1 = Quaternion::GetRotationQuaternion(v1_before, v1_after);
197         Vector3 p2(v2_before);
198         Vector3 pp2 = p2.GetRotated(q1);
199         Vector3 vv2(pp2);
200         Vector3 r2 = v1_after;
201
202         if((vv2 ^ v2_after)*v1_after > 0)
203         {
204             double sin_theta = (vv2 ^ v2_after).GetMagnitude();
205             double cos_theta = vv2*v2_after;
206             double theta = atan2(sin_theta, cos_theta);
207             Quaternion q2 = Quaternion::GetRotationQuaternion(r2, theta);
208             return q2*q1;
209         }
210         else
211         {
212             double sin_theta = -((vv2 ^ v2_after).GetMagnitude());
213             double cos_theta = vv2*v2_after;
214             double theta = atan2(sin_theta, cos_theta);
215             Quaternion q2 = Quaternion::GetRotationQuaternion(r2, theta);
216             return q2*q1;
217         }
218     }
219
220     void Quaternion::Decompose(double& angle_1,
221                                double& angle_2,
222                                double& angle_3,
223                                const Vector3& v1,
224                                const Vector3& v2,
225                                const Vector3& v3) const
226     {
227         Quaternion q1 = Quaternion::GetRotationQuaternion(v1, angle_1);
228         Quaternion q2 = Quaternion::GetRotationQuaternion(v2, angle_2);
229         Quaternion q3 = Quaternion::GetRotationQuaternion(v3, angle_3);
230         Quaternion qt = q3*q2*q1;
231
232         const double err = 0.0001;
233         const double delta_angle = 0.0001;
234         const unsigned int tries = 10000;
235         unsigned int k = 0;
236
237         Quaternion qr = *this;
238
239         double dist = (qr - qt).GetModulus();
240
241         while((dist >= err) && (k < tries))
242         {
243             for(int c1 = -1; c1 <= 1; ++c1)
244             {
245                 for(int c2 = -1; c2 <= 1; ++c2)
246                 {
247                     for (int c3 = -1; c3 <= 1; ++c3)
248                     {
249                         double new_angle_1 = angle_1 + delta_angle*c1;
```

```
250                         double new_angle_2 = angle_2 + delta_angle*c2;
251                         double new_angle_3 = angle_3 + delta_angle*c3;
252
253                         Quaternion new_q1 = Quaternion::GetRotationQuaternion(v1, new_angle_1);
254                         Quaternion new_q2 = Quaternion::GetRotationQuaternion(v2, new_angle_2);
255                         Quaternion new_q3 = Quaternion::GetRotationQuaternion(v3, new_angle_3);
256                         Quaternion new_qt = new_q3*new_q2*new_q1;
257
258                         double new_dist = (qr - new_qt).GetModulus();
259
260                         if(new_dist < dist)
261                         {
262                             angle_1 = new_angle_1;
263                             angle_2 = new_angle_2;
264                             angle_3 = new_angle_3;
265                             dist = new_dist;
266                         }
267                     }
268                 }
269             }
270             ++k;
271         }
272     }
273
274     void Quaternion::GetAxisAndAngle(Vector3& vector, double& angle) const
275     {
276         Quaternion qr = *this;
277         Vector3 e(qr);
278         e.Normalize();
279         double cc = this->w;
280         double ss = ((qr - Quaternion(this->w, 0, 0, 0)) * Quaternion(e).GetConjugate()).w;
281         double ang = 2*atan2(ss, cc);
282
283         vector = e;
284         angle = ang;
285     }
286
287     void Quaternion::GetRPY(double& roll, double& pitch, double& yaw, unsigned int solution)
288     {
289         tf::Quaternion tf_q(this->x, this->y, this->z, this->w);
290         tf::Matrix3x3(tf_q).getRPY(roll, pitch, yaw, solution);
291     }
292
293     const Quaternion Quaternion::GetDifference(const Quaternion& q1, const Quaternion& q2)
294     {
295         return (q1.GetConjugate())*q2;
296     }
297
298     const Quaternion Quaternion::identity = Quaternion(1,0,0,0);
299     const Quaternion Quaternion::i    = Quaternion(0,1,0,0);
300     const Quaternion Quaternion::j    = Quaternion(0,0,1,0);
301     const Quaternion Quaternion::k    = Quaternion(0,0,0,1);
302
303     std::ostream& operator<<(std::ostream& os, const Quaternion& q)
304     {
305         return os << q.ToString();
306     }
307
308 }
```

C.2.3. vector3.h

```
/* Clase Vector3
 * Incluye operaciones para creación y manipulación
 * de vectores de 3 dimensiones
 *
 * Autor: Daniel Fernández Villanueva
 */

#ifndef VECTOR3_H
#define VECTOR3_H

#include <string>
#include <sstream>
#include <cmath>
#include <dfv/quaternion.h>
#include <ros/ros.h>
#include <geometry_msgs/Vector3.h>
#include <geometry_msgs/Vector3Stamped.h>

namespace dfv
{

    class Quaternion;

    class Vector3
    {
        public:
            Vector3();
            Vector3(double x_, double y_, double z_);
            explicit Vector3(const Quaternion& q);
            virtual ~Vector3();

            // ******** Operador de asignación ******** //
            Vector3& operator=(const Vector3& v);

            // ******** Operadores de asignación compuestos ******** //
            Vector3& operator+=(const Vector3& v);
            Vector3& operator-=(const Vector3& v);
            Vector3& operator*=(const double k);

            // ******** Operadores aritméticos binarios ******** //
            const Vector3 operator+(const Vector3& v) const;
            const Vector3 operator-(const Vector3& v) const;
            friend const Vector3 operator*(double k, Vector3& v);
            const Vector3 operator*(double k) const;
            // Producto escalar:
            double operator*(const Vector3& v) const;
            // Producto vectorial:
            const Vector3 operator^(const Vector3& v) const;

            // ******** Operadores de comparación ******** //
            bool operator==(const Vector3& v) const;
            bool operator!=(const Vector3& v) const;

            // Funcion que devuelve la magnitud del vector:
            double        GetMagnitude() const;

            // Funcion que normaliza el vector:
            void          Normalize();

            // Funcion que devuelve el vector normalizado:
            const Vector3 GetNormalized() const;
```

```
62
63              // Funcion que devuelve un vector de longitud
64              // k veces el original con la misma dirección
65              // y sentido:
66              const Vector3  GetScalated(double k) const;
67
68              // Función que rota el vector de acuerdo con el cuaternión
69              // pasado como parámetro:
70              Vector3&       Rotate(const Quaternion q);
71
72              // Devuelve el vector rotado según el cuaternión pasado
73              // como parámetro:
74              const Vector3  GetRotated(const Quaternion& q) const;
75
76              // Función que devuelve una representación
77              // del vector como texto:
78              std::string    ToString() const;
79
80              // Vector unitario codireccional al eje x:
81              static Vector3 i;
82
83              // Vector unitario codireccional al eje y:
84              static Vector3 j;
85
86              // Vector unitario codireccional al eje z:
87              static Vector3 k;
88
89              // Componentes del vector:
90              double x;
91              double y;
92              double z;
93
94          protected:
95          private:
96
97      };
98
99      std::ostream& operator<<(std::ostream& os, const Vector3& v);
100
101 }
102
103 #endif // Vector3_H
```

C.2.4. vector3.cpp

```cpp
#include <dfv/vector3.h>

namespace dfv
{
    Vector3::Vector3():
        x(0), y(0), z(0)
    {

    }

    Vector3::Vector3(double x_, double y_, double z_):
        x(x_), y(y_), z(z_)
    {

    }

    Vector3::Vector3(const Quaternion& q):
        x(q.x), y(q.y), z(q.z)
    {

    }

    Vector3::~Vector3()
    {
        //dtor
    }

    Vector3& Vector3::operator=(const Vector3& v)
    {
        if(this != &v)
        {
            this->x = v.x;
            this->y = v.y;
            this->z = v.z;
        }

        return *this;
    }

    Vector3& Vector3::operator+=(const Vector3& v)
    {
        this->x += v.x;
        this->y += v.y;
        this->z += v.z;

        return *this;
    }

    Vector3& Vector3::operator-=(const Vector3& v)
    {
        this->x -= v.x;
        this->y -= v.y;
        this->z -= v.z;

        return *this;
    }

    Vector3& Vector3::operator*=(const double k)
    {
        this->x *= k;
        this->y *= k;
```

```
62          this->z *= k;
63
64          return *this;
65      }
66
67      const Vector3 Vector3::operator+(const Vector3& v) const
68      {
69          return Vector3(*this) += v;
70      }
71
72      const Vector3 Vector3::operator-(const Vector3& v) const
73      {
74          return Vector3(*this) -= v;
75      }
76
77      const Vector3 operator*(double k, Vector3& v)
78      {
79          return Vector3(v) *= k;
80      }
81
82      const Vector3 Vector3::operator*(double k) const
83      {
84          return Vector3(*this) *= k;
85      }
86
87      double Vector3::operator*(const Vector3& v) const
88      {
89          return this->x * v.x + this->y * v.y + this->z * v.z;
90      }
91
92      const Vector3 Vector3::operator^(const Vector3& v) const
93      {
94          return Vector3(this->y*v.z - this->z*v.y,
95                         this->z*v.x - this->x*v.z,
96                         this->x*v.y - this->y*v.x);
97      }
98
99      bool Vector3::operator==(const Vector3& v) const
100     {
101         return (this->x == v.x) && (this->y == v.y) && (this->z == v.z);
102     }
103
104     bool Vector3::operator!=(const Vector3& v) const
105     {
106         return !(*this == v);
107     }
108
109     double Vector3::GetMagnitude() const
110     {
111         return sqrt(pow(this->x, 2.0) + pow(this->y, 2.0) + pow(this->z, 2.0));
112     }
113
114     void Vector3::Normalize()
115     {
116         double mag = this->GetMagnitude();
117         if(mag > 0)
118         {
119             this->x /= mag;
120             this->y /= mag;
121             this->z /= mag;
122         }
123         else
124         {
```

```
125             this->x = 0.0;
126             this->y = 0.0;
127             this->z = 0.0;
128         }
129 
130     }
131 
132     const Vector3 Vector3::GetNormalized() const
133     {
134         double mag = this->GetMagnitude();
135 
136         if(mag > 0)
137         {
138             Vector3 result;
139             result.x = this->x / mag;
140             result.y = this->y / mag;
141             result.z = this->z / mag;
142             return result;
143         }
144         else
145         {
146             return Vector3(0.0, 0.0, 0.0);
147         }
148     }
149 
150     const Vector3 Vector3::GetScalated(double k) const
151     {
152         return Vector3(this->x*k, this->y*k, this->z*k);
153     }
154 
155     Vector3& Vector3::Rotate(const Quaternion q)
156     {
157         *this = this->GetRotated(q);
158         return *this;
159     }
160 
161     const Vector3 Vector3::GetRotated(const Quaternion& q) const
162     {
163         Quaternion v(*this);
164         return Vector3(q*v*(q.GetConjugate()));
165     }
166 
167     std::string Vector3::ToString() const
168     {
169         std::stringstream ss;
170         ss << "[" << this->x << ", " << this->y << ", " << this->z << "]";
171         return ss.str();
172     }
173 
174     Vector3 Vector3::i = Vector3(1, 0, 0);
175     Vector3 Vector3::j = Vector3(0, 1, 0);
176     Vector3 Vector3::k = Vector3(0, 0, 1);
177 
178     std::ostream& operator<<(std::ostream& os, const Vector3& v)
179     {
180         return os << v.ToString();
181     }
182 }
```

C.2.5. matrix.h

```
/* Clase Matrix
 * Incluye operaciones para creación y manipulación
 * de matrices de cualquier dimensión
 *
 * Autor: Daniel Fernández Villanueva
 */

#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <vector>
#include <sstream>
#include <cstdlib>

namespace dfv
{

    class Matrix
    {
        public:
            Matrix();
            Matrix(unsigned int size);
            Matrix(unsigned int rows, unsigned int columns);
            virtual ~Matrix();

            // ******** Operador de asignación ******** //
            Matrix& operator=(const Matrix& q);

            // ******** Operadores de asignación compuestos ******** //
            Matrix& operator+=(const Matrix& q);
            Matrix& operator-=(const Matrix& q);
            Matrix& operator*=(const double k);

            // ******** Operadores aritméticos binarios ******** //
            const Matrix operator+(const Matrix& q) const;
            const Matrix operator-(const Matrix& q) const;
            friend const Matrix operator*(double k, Matrix& q);
            const Matrix operator*(double k) const;
            // Producto de Matrices:
            const Matrix operator*(const Matrix& q) const;

            // ******** Operadores de comparación ******** //
            bool operator==(const Matrix& q) const;
            bool operator!=(const Matrix& q) const;

            // Función que devuelve una representación
            // de la matriz como texto:
            std::string          ToString() const;

            // Función que crea una matriz de rows filas y columns columnas
            Matrix&           Create(unsigned int rows, unsigned int columns, double value = 0);

            // Función que devuelve el elemento situado en la fila row y columna col
            double            Get(unsigned int row, unsigned int col) const;

            // Función que permie cambiar el valor del elemento situado en la fila row y columna
                col
            void              Set(unsigned int row, unsigned int col, double value);

            // Función que devuelve el número de filas de la matriz
```

```
61            unsigned int      GetRows() const;
62
63            // Función que devuelve el número de columnas de la matriz
64            unsigned int      GetColumns() const;
65
66            // Función que devuleve el menor asociado a la fila row y columna column
67            Matrix            GetMinor(unsigned int row, unsigned int column) const;
68
69            // Función que asigna un valor aleatoria ontre 0 y 1 a cada elemento de la matriz
70            void              Randomize();
71
72            // Función que devuelve el determinante de la matriz
73            double            GetDeterminant() const;
74
75            // Función que devuelve la traspuesta de la matriz
76            const Matrix      GetTransposed() const;
77
78            // Función que devuelve el adjunto de la matriz
79            const Matrix      GetAdjoint() const;
80
81            // Función que devuelve la matriz transpuesta conjugada
82            const Matrix      GetAdjugate() const;
83
84            // Función que devuelve la inversa de la matriz
85            const Matrix      GetInverse() const;
86
87            // Operador equivalente a la función Get()
88            double            operator()(unsigned int row, unsigned int column) const;
89
90            // Función que devuelve una matriz unitaria con la dimensión dada
91            static const Matrix Identity(unsigned int size);
92
93        protected:
94        private:
95            unsigned int      rows;
96            unsigned int      columns;
97            std::vector<double> data;
98    };
99
100   std::ostream& operator<<(std::ostream& os, Matrix& m);
101
102}
103
104#endif // Matrix_H
```

C.2.6. matrix.cpp

```
#include <dfv/matrix.h>

namespace dfv
{

    Matrix::Matrix(): rows(0), columns(0)
    {

    }

    Matrix::Matrix(unsigned int size)
    {
        this->Create(size, size);
    }

    Matrix::Matrix(unsigned int rows, unsigned int columns)
    {
        this->Create(rows, columns);
    }

    Matrix::~Matrix()
    {
        //dtor
    }

    Matrix& Matrix::operator=(const Matrix& m)
    {
        if(this != &m)
        {
            this->Create(m.GetRows(), m.GetColumns());

            for(unsigned int j = 0; j < this->GetRows(); j++)
            {
                for(unsigned int i = 0; i < this->GetColumns(); i++)
                {
                    this->Set(j, i, m.Get(j, i));
                }
            }
        }

        return *this;
    }

    Matrix& Matrix::operator+=(const Matrix& m)
    {
        if(m.GetRows() == this->GetRows() && m.GetColumns() == this->GetColumns())
        {
            for(unsigned int j = 0; j < this->GetRows(); j++)
            {
                for(unsigned int i = 0; i < this->GetColumns(); i++)
                {
                    this->Set(j, i, this->Get(j, i) + m.Get(j, i));
                }
            }

            return *this;
        }
        else
        {
            return *this; // arrojar error
        }
```

```
62      }
63
64      Matrix& Matrix::operator-=(const Matrix& m)
65      {
66          if(m.GetRows() == this->GetRows() && m.GetColumns() == this->GetColumns())
67          {
68              for(unsigned int j = 0; j < this->GetRows(); j++)
69              {
70                  for(unsigned int i = 0; i < this->GetColumns(); i++)
71                  {
72                      this->Set(j, i, this->Get(j, i) - m.Get(j, i));
73                  }
74              }
75
76              return *this;
77          }
78          else
79          {
80              return *this; // arrojar error
81          }
82      }
83
84      Matrix& Matrix::operator*=(const double k)
85      {
86          for(unsigned int j = 0; j < this->GetRows(); j++)
87          {
88              for(unsigned int i = 0; i < this->GetColumns(); i++)
89              {
90                  this->Set(j, i, this->Get(j, i) * k);
91              }
92          }
93
94          return *this;
95      }
96
97      const Matrix Matrix::operator+(const Matrix& m) const
98      {
99          return Matrix(*this) += m;
100     }
101
102     const Matrix Matrix::operator-(const Matrix& m) const
103     {
104         return Matrix(*this) -= m;
105     }
106
107     const Matrix operator*(double k, Matrix& m)
108     {
109         return Matrix(m) *= k;
110     }
111
112     const Matrix Matrix::operator*(double k) const
113     {
114         return Matrix(*this) *= k;
115     }
116
117     const Matrix Matrix::operator*(const Matrix& m) const
118     {
119         Matrix result;
120         if(this->GetColumns() == m.GetRows())
121         {
122             result.Create(this->GetRows(), m.GetColumns());
123             for(unsigned int j = 0; j < result.GetRows(); j++)
124             {
```

```
125                 for(unsigned int i = 0; i < result.GetColumns(); i++)
126                 {
127                     for(unsigned int k = 0; k < this->GetColumns(); k++)
128                     {
129                         result.Set(j, i, result.Get(j, i) + this->Get(j, k)*m.Get(k, i));
130                     }
131                 }
132             }
133         }
134         return result;
135     }
136 
137     bool Matrix::operator==(const Matrix& m) const
138     {
139         if(m.GetRows() == this->GetRows() && m.GetColumns() == this->GetColumns())
140         {
141             for(unsigned int j = 0; j < this->GetRows(); j++)
142             {
143                 for(unsigned int i = 0; i < this->GetColumns(); i++)
144                 {
145                     if(this->Get(j, i) != m.Get(j, i))
146                     {
147                         return false;
148                     }
149                 }
150             }
151 
152             return true;
153         }
154         else
155         {
156             return false;
157         }
158     }
159 
160     bool Matrix::operator!=(const Matrix& m) const
161     {
162         return !(*this == m);
163     }
164 
165     std::string Matrix::ToString() const
166     {
167         std::stringstream ss;
168 
169         for(unsigned int j = 0; j < this->GetRows(); j++)
170         {
171             for(unsigned int i = 0; i < this->GetColumns(); i++)
172             {
173                 if(this->Get(j, i) >= 0)
174                 {
175                     ss << " ";
176                 }
177                 ss << this->Get(j, i) << "\t";
178             }
179             ss << std::endl;
180         }
181 
182         return ss.str();
183     }
184 
185     Matrix& Matrix::Create(unsigned int rows, unsigned int columns, double value)
186     {
187         this->rows = rows;
```

```
188         this->columns = columns;
189         this->data.resize(rows*columns);
190         typename std::vector<double>::iterator it;
191         for(it = this->data.begin(); it != this->data.end(); it++)
192         {
193             *it = value;
194         }
195         return *this;
196     }
197 
198     double Matrix::Get(unsigned int row, unsigned int col) const
199     {
200         return this->data.at(row + col*this->columns);
201     }
202 
203     void Matrix::Set(unsigned int row, unsigned int col, double value)
204     {
205         this->data.at(row + col*this->columns) = value;
206     }
207 
208     unsigned int Matrix::GetRows() const
209     {
210         return this->rows;
211     }
212 
213     unsigned int Matrix::GetColumns() const
214     {
215         return this->columns;
216     }
217 
218     Matrix Matrix::GetMinor(unsigned int row, unsigned int column) const
219     {
220         Matrix result;
221         if(this->GetRows() == this->GetColumns())
222         {
223             result.Create(this->GetRows() - 1, this->GetColumns() - 1);
224             unsigned int row_count = 0;
225             unsigned int col_count = 0;
226             for(unsigned int j = 0; j < this->GetRows(); j++)
227             {
228                 if(j != row)
229                 {
230                     col_count = 0;
231                     for(unsigned int i = 0; i < this->GetColumns(); i++)
232                     {
233                         if(i != column)
234                         {
235                             result.Set(row_count, col_count, this->Get(j, i));
236                             col_count++;
237                         }
238                     }
239                     row_count++;
240                 }
241             }
242         }
243         return result;
244     }
245 
246     void Matrix::Randomize()
247     {
248         for(unsigned int j = 0; j < this->GetRows(); j++)
249         {
250             for(unsigned int i = 0; i < this->GetColumns(); i++)
```

```
251             {
252                 this->Set(j, i, (double)rand() / (double)RAND_MAX);
253             }
254         }
255     }
256 
257     double Matrix::GetDeterminant() const
258     {
259         double result = 0;
260         if(this->GetRows() == this->GetColumns())
261         {
262             if(this->GetRows() == 1)
263             {
264                 return this->Get(0, 0);
265             }
266             else
267             {
268                 for(unsigned int i = 0; i < this->GetColumns(); i++)
269                 {
270                     Matrix minor = this->GetMinor(0, i);
271                     result += (i % 2 == 0? 1.0 : -1.0)*this->Get(0, i)*minor.GetDeterminant();
272                 }
273             }
274         }
275         return result;
276     }
277 
278     const Matrix Matrix::GetTransposed() const
279     {
280         Matrix result;
281         result.Create(this->GetColumns(), this->GetRows());
282         for(unsigned int j = 0; j < this->GetRows(); j++)
283         {
284             for(unsigned int i = 0; i < this->GetColumns(); i++)
285             {
286                 result.Set(j, i, this->Get(i, j));
287             }
288         }
289         return result;
290     }
291 
292     const Matrix Matrix::GetAdjoint() const
293     {
294         Matrix result;
295         result.Create(this->GetRows(), this->GetColumns());
296         for(unsigned int j = 0; j < this->GetRows(); j++)
297         {
298             for(unsigned int i = 0; i < this->GetColumns(); i++)
299             {
300                 Matrix minor;
301                 minor = this->GetMinor(j, i);
302                 double temp = ((j+i)%2 == 0 ? 1.0 : -1.0) * minor.GetDeterminant();
303                 result.Set(j, i, temp);
304             }
305         }
306         return result;
307     }
308 
309     const Matrix Matrix::GetAdjugate() const
310     {
311         Matrix result;
312         result.Create(this->GetRows(), this->GetColumns());
313         for(unsigned int j = 0; j < this->GetRows(); j++)
```

```
314         {
315             for(unsigned int i = 0; i < this->GetColumns(); i++)
316             {
317                 Matrix minor;
318                 minor = this->GetMinor(j, i);
319                 double temp = ((j+i)%2 == 0 ? 1.0 : -1.0) * minor.GetDeterminant();
320                 result.Set(i, j, temp);
321             }
322         }
323         return result;
324     }
325 
326     const Matrix Matrix::GetInverse() const
327     {
328         Matrix result;
329         if((this->GetRows() == this->GetColumns()) && this->GetDeterminant() != 0.0)
330         {
331             Matrix temp = this->GetAdjugate();
332             temp *= (1.0 / this->GetDeterminant());
333             result = temp;
334         }
335         return result;
336     }
337 
338     double Matrix::operator()(unsigned int row, unsigned int column) const
339     {
340         return this->Get(row, column);
341     }
342 
343     const Matrix Matrix::Identity(unsigned int size)
344     {
345         Matrix result(size, size);
346 
347         for(int i = 0; i < size; ++i)
348         {
349             result.Set(i, i, 1.0);
350         }
351 
352         return result;
353     }
354 
355     std::ostream& operator<<(std::ostream& os, Matrix& m)
356     {
357         os << m.ToString();
358         return os;
359     }
360 
361 }
```

C.2.7. utils.h

```
#ifndef UTILS_H
#define UTILS_H

namespace dfv
{
    const long double pi =
        3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803L;

    long double DegToRad(long double deg);
    long double RadToDeg(long double rad);
}

#endif
```

C.2.8. utils.cpp

```cpp
#include <dfv/utils.h>

namespace dfv
{

    long double DegToRad(long double deg)
    {
        return deg * pi / 180.0L;
    }

    long double RadToDeg(long double rad)
    {
        return rad * 180.0L / pi;
    }

}
```

C.2.9. dfv.h

```
#ifndef DFV_H
#define DFV_H

#include <dfv/utils.h>
#include <dfv/vector3.h>
#include <dfv/quaternion.h>
#include <dfv/matrix.h>

#endif
```

C.3. Youbot controller

C.3.1. youbot_controller.h

```
/*
 * Programa que toma los datos de los topics
 * de tres sensores xsens, calcula los ángulos
 * de rotación entre cada sensor y los publica
 * en el topic para mover el brazo robótico del
 * robot Youbot.
 *
 * Autor: Daniel Fernández Villanueva
 * Mayo de 2013
 *
 */

#include <iostream>
#include <ros/ros.h>
#include <dfv/dfv.h>
#include <xsens_driver/xsens_sensor_subscriber.h>
#include <youbot_controller/youbot.h>

// Función que devuelve el equivalente al ángulo
// dentro del rango [0, 2*pi]
double NormalizeAngle(double angle);

// Borrar
void CorrectRPY(double& roll, double& pitch, double& yaw);

// ángulos límite
const double ang_min[] = {0.010, 0.010, -5.026, 0.022, 0.110};
const double ang_max[] = {5.840, 2.617, -0.015, 3.429, 5.641};
const double sec_ang = 0.05;

int main(int argc, char** argv)
{
    ros::init(argc, argv, "youbot_controller");
    ros::NodeHandle node_handle;

    xsens::SensorSubscriberList sensor_subscriber_list(node_handle);
    unsigned int mt_count = sensor_subscriber_list.GetMtCount();

    // Si no hay sensores, salimos del programa
    if(mt_count == 0)
    {
        ROS_ERROR("No MTs found. Quitting...");
        return 1;
    }
    ROS_INFO("Detected IMU count: %d",mt_count);
    if(mt_count != 3)
    {
        ROS_WARN("3 sensors are needed by this demo to work properly. "
            "Please connect 3 Xsens IMUs to the Xbus Master and restart the program. "
            "Quitting...");
        return 1;
    }

    // Esperamos a que ROS llame a las funciones callback para
    // leer los datos de los topics
    ROS_INFO("Waiting for the topics to be read...");
    while(sensor_subscriber_list.GetOriQuat(0) == dfv::Quaternion(0.0, 0.0, 0.0, 0.0))
    {
        ros::Duration(0.05).sleep();
```

```
60          ros::spinOnce();
61      }
62
63      Youbot youbot(node_handle);
64
65      // bucle principal
66      while(ros::ok())
67      {
68          dfv::Quaternion q0 = sensor_subscriber_list.GetOriQuat(0);
69          dfv::Quaternion q1 = sensor_subscriber_list.GetOriQuat(1);
70          dfv::Quaternion q2 = sensor_subscriber_list.GetOriQuat(2);
71
72          dfv::Quaternion q01 = dfv::Quaternion::GetDifference(q0, q1);
73          dfv::Quaternion q12 = dfv::Quaternion::GetDifference(q1, q2);
74
75          // Cálculo de los ángulos entre cada sensor
76
77          double roll_0;
78          double pitch_0;
79          double yaw_0;
80          q0.GetRPY(roll_0, pitch_0, yaw_0, 1);
81          if(fabs(roll_0) > dfv::pi/2.0)
82          {
83              q0.GetRPY(roll_0, pitch_0, yaw_0, 2);
84          }
85          //CorrectRPY(roll_0, pitch_0, yaw_0);
86
87          double roll_01;
88          double pitch_01;
89          double yaw_01;
90          q01.GetRPY(roll_01, pitch_01, yaw_01, 1);
91          //CorrectRPY(roll_01, pitch_01, yaw_01);
92          if(fabs(roll_01) > dfv::pi/2.0)
93          {
94              q01.GetRPY(roll_01, pitch_01, yaw_01, 2);
95          }
96
97          double roll_12;
98          double pitch_12;
99          double yaw_12;
100         q12.GetRPY(roll_12, pitch_12, yaw_12, 1);
101         //CorrectRPY(roll_12, pitch_12, yaw_12);
102         if(fabs(roll_12) > dfv::pi/2.0)
103         {
104             q12.GetRPY(roll_12, pitch_12, yaw_12, 2);
105         }
106
107         // Adaptación de los ángulos obtenidos
108         // teniendo en cuenta los offsets de cada
109         // articulación
110
111         double offsets[] = {2.9, -0.5, -2.5, 1.5, 1.5};
112
113         double angs[5];
114         angs[0] = offsets[0] + NormalizeAngle(-yaw_0);
115         //angs[1] = -pitch_0 < 0 ? 0 : -pitch_0;
116         angs[1] = offsets[1] + 1.0 * (-pitch_0);
117         angs[2] = offsets[2] + 1.0 * (-pitch_01);
118         angs[3] = offsets[3] + 1.0 * (-pitch_12);
119         angs[4] = offsets[4] + 2.0 * (-roll_01);
120
121         for(int i = 0; i < 5; i++)
122         {
```

```
123             std::cout << "ang[" << i << "] " << angs[i] << std::endl;
124         }
125 
126         // Pasamos los ángulos al robot
127         // Los limitamos al intervalo que acepta cada articulación
128 
129         /*youbot.joint_positions[0] = (angs[0] < 0.02) ? 0.02 : ((angs[0] > 5.83) ? 5.83 :
                angs[0]);
130         youbot.joint_positions[1] = (angs[1] < 0.02) ? 0.02 : ((angs[1] > 2.60) ? 2.60 :
                angs[1]);
131         youbot.joint_positions[2] = (angs[2] < -5.01) ? -5.01 : ((angs[2] > -0.02) ? -0.02 :
                angs[2]);
132         youbot.joint_positions[3] = (angs[3] < 0.03) ? 0.03 : ((angs[3] > 3.41) ? 3.41 :
                angs[3]);
133         youbot.joint_positions[4] = (angs[4] < 0.12) ? 0.12 : ((angs[4] > 5.63) ? 5.63 :
                angs[4]);*/
134 
135         for(int i = 0; i < 5; i++)
136         {
137             youbot.joint_positions[i] = (angs[i] < ang_min[i] + sec_ang) ?
138                 ang_min[i] + sec_ang : ((angs[i] > ang_max[i] - sec_ang) ?
139                     ang_max[i] - sec_ang : angs[i]);
140 
141             std::cout << "ang min: " << ang_min[i] << std::endl;
142             std::cout << "ang max: " << ang_max[i] << std::endl;
143         }
144 
145         // Publicamos en el topic del robot
146 
147         youbot.PublishMessage();
148 
149         // Imprimimos en pantalla los ángulos que le hemos pasado al robot
150 
151         std::cout << "Published angles: " << std::endl;
152         for(unsigned int i = 0; i < 5; ++i)
153         {
154             std::cout << "joint #" << (i+1) << ": " << youbot.joint_positions[i] << std::endl;
155         }
156         std::cout << "--------------------------------" << std::endl;
157 
158         // Frecuencia del bucle: 20 Hz
159         ros::Duration(0.05).sleep();
160         ros::spinOnce();
161     }
162 
163     return 0;
164 }
165 
166 double NormalizeAngle(double angle)
167 {
168     while(angle < -dfv::pi)
169     {
170         angle += 2*dfv::pi;
171     }
172     while(angle > dfv::pi)
173     {
174         angle -= 2*dfv::pi;
175     }
176     return angle;
177 }
178 
179 void CorrectRPY(double& roll, double& pitch, double& yaw)
180 {
```

```
181     if (roll > dfv::pi/2.0)
182     {
183         roll -= dfv::pi;
184         pitch = (pitch < 0)? - dfv::pi - pitch : dfv::pi - pitch;
185         yaw = (yaw < 0)? yaw + dfv::pi : yaw - dfv::pi;
186 
187     }
188     else if (roll < -dfv::pi/2.0)
189     {
190         roll += dfv::pi;
191         pitch = (pitch < 0)? - dfv::pi - pitch : dfv::pi - pitch;
192         yaw = (yaw < 0)? yaw + dfv::pi : yaw - dfv::pi;
193     }
194 }
```

C.3.2. youbot.h

```
/*
 * Clase Youbot. Encapsula el mecanismo
 * de publicación de los mensajes necesarios
 * para mover las articulaciones del
 * robot Youbot.
 *
 * Autor: Daniel Fernández Villanueva
 * Mayo de 2013
 *
 */

#ifndef YOUBOT_H
#define YOUBOT_H

#include <string>
#include <cstdlib>
#include <ros/ros.h>
#include <brics_actuator/JointPositions.h>

class Youbot
{
    public:
        Youbot(ros::NodeHandle& node_handle_,
               std::string arm_topic_name_ = "arm_1/arm_controller/position_command",
               std::string gripper_topic_name_ = "arm_1/gripper_controller/position_command");
        ~Youbot();

        double joint_positions[5];
        double gripper_positions[2];
        void PublishMessage(bool publish_gripper = false);

    private:
        ros::NodeHandle& node_handle;
        ros::Publisher arm_publisher;
        ros::Publisher gripper_publisher;

        std::vector<brics_actuator::JointValue> v_joint_values;
        std::vector<brics_actuator::JointValue> v_gripper_values;
};

#endif
```

C.3.3. youbot.cpp

```
#include <youbot_controller/youbot.h>

Youbot::Youbot(ros::NodeHandle& node_handle_,
               std::string arm_topic_name_,
               std::string gripper_topic_name):
    node_handle(node_handle_)
{
    this->arm_publisher =
        this->node_handle.advertise<brics_actuator::JointPositions>(arm_topic_name_, 1);
    this->gripper_publisher =
        this->node_handle.advertise<brics_actuator::JointPositions>(gripper_topic_name, 1);

        this->v_joint_values.resize(5);
        for(int i = 0; i < 5; ++i)
        {
            std::stringstream ss;
            ss << "arm_joint_" << (i+1);
            v_joint_values[i].joint_uri = ss.str();
            v_joint_values[i].unit = std::string("rad");
            v_joint_values[i].value = 0.0;
        }

        this->v_gripper_values.resize(2);
        v_gripper_values[0].joint_uri = "gripper_finger_joint_l";
        v_gripper_values[0].unit = std::string("m");
        v_gripper_values[0].value = 0.001;

        v_gripper_values[1].joint_uri = "gripper_finger_joint_r";
        v_gripper_values[1].unit = std::string("m");
        v_gripper_values[1].value = 0.001;

        this->gripper_positions[0] = 0.01;
        this->gripper_positions[1] = 0.01;

}

Youbot::~Youbot()
{

}

void Youbot::PublishMessage(bool publish_gripper)
{
    brics_actuator::JointPositions msg;
    for(int i = 0; i < 5; ++i)
    {
        v_joint_values[i].value = this->joint_positions[i];
    }
    msg.positions = v_joint_values;
    this->arm_publisher.publish(msg);

    // Publicar las posiciones del gripper resulta en una
    // bajada importante en el rendimiento del robot.
    // Se recomienda no pubicar las posiciones del gripper
    // al mismo ritmo que las del brazo
    if(publish_gripper)
    {
        brics_actuator::JointPositions gripper_msg;
        v_gripper_values[0].value = this->gripper_positions[0];
        v_gripper_values[1].value = this->gripper_positions[1];
        gripper_msg.positions = v_gripper_values;
```

```
62
63          this->gripper_publisher.publish(gripper_msg);
64      }
65  }
```

Apéndice D

SOLUCIÓN DE PROBLEMAS

D.1. Error iniciando Gazebo

```
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_gaze
/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
```

Solución: Ejecutar comando:

```
$ ps ax | grep [g]z
```

Ver si hay un proceso gzserver

```
 3118 ?        Sl    12:47
 /opt/ros/fuerte/stacks/simulator_gazebo/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/fuer
-s /opt/ros/fuerte/stacks/simulator_gazebo/gazebo/lib/libgazebo_ros_api_plugin.so
```

Si lo hay, ejecutar *System Monitor* y matar el proceso *gzserver*.

Parte III

Otros documentos

D.2. Manual del sensor MTi-G

AAAAAAAAAAAAAAAAAAAAAAa