

UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE
GIJÓN

**ÁREA DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL**

PROYECTO FIN DE CARRERA N° 3122067

**INTEGRACIÓN DEL DISPOSITIVO 'MICROSOFT KINECT'
COMO PARTE DE UNA ARQUITECTURA ROBÓTICA
GENÉRICA PARA SU USO COMO DISPOSITIVO DE ENTRADA**

DOCUMENTO N° I al VI

MEMORIA DEL PROYECTO



MIGUEL A. GARCÍA GONZÁLEZ
NOVIEMBRE - 2012

Índice de contenidos

Índice de contenidos	I
Índice de figuras	VII
Índice de tablas	IX
1. MEMORIA	4
1.1. Introducción	4
1.1.1. Identificación del proyecto	4
1.1.2. Visión general del proyecto	5
1.1.3. Visión general del documento	5
1.2. Ámbito y alcance del sistema	6
1.2.1. Ámbito del proyecto	6
1.2.2. Descripción general del proyecto	7
1.2.3. Objetivos y alcance	7
1.3. Descripción del sistema actual	9
1.3.1. Modelo actual	9
1.3.2. Lista de problemas y necesidades	10
1.4. Análisis de alternativas	12
1.4.1. Discusión previa - Decisiones de diseño	12
1.4.2. Descripción de la solución	15
1.4.3. Valoración de la solución	17

1.5. Posibles ampliaciones	20
1.6. Documentos que conforman la documentación	21
1.7. Documentación auxiliar	22
1.7.1. Robot Operating System (ROS)	22
1.7.2. Sistemas de referencia y matrices de transformación	24
1.7.3. Cámara de profundidad <i>Microsoft Kinect</i>	26
1.7.4. Características del robot modelo <i>Amigobot</i>	28
1.7.5. Algoritmo Iterativo del Punto más Cercano (ICP)	29
1.8. Glosario	31
1.9. Referencias electrónicas	35
2. PLANIFICACIÓN Y PRESUPUESTO	40
2.1. Introducción	40
2.1.1. Identificación del proyecto	40
2.1.2. Visión general del proyecto	41
2.1.3. Visión general del documento	41
2.2. Plan para las fases	42
2.3. Documentación resultante de las fases	44
2.4. Objetivos de cada iteración	45
2.5. Planificación de los costes	47
2.5.1. Estimación de recursos necesarios	47
2.6. Presupuesto	49
2.6.1. Mediciones	49
2.6.2. Cuadros de precios	49
2.6.3. Presupuestos parciales	50
2.6.4. Presupuesto final	51

3. ESPECIFICACIÓN DEL SISTEMA	56
3.1. Introducción	56
3.1.1. Identificación del proyecto	56
3.1.2. Visión general del proyecto	57
3.1.3. Visión general del documento	57
3.2. Ámbito y alcance del sistema	58
3.2.1. Ámbito del proyecto	58
3.2.2. Descripción general del proyecto	59
3.2.3. Objetivos y alcance	59
3.3. Lista de usuarios participantes	61
3.3.1. Mercado del producto	61
3.3.2. Lista de usuarios	61
3.3.3. Perfil de usuario	62
3.3.4. Necesidades de los usuarios	63
3.4. Análisis de alternativas	64
3.5. Especificación del sistema	65
3.5.1. Diagrama de subsistemas	65
3.5.2. Interfaces con otros sistemas	65
3.5.3. Vista del sistema	67
3.6. Entorno tecnológico de desarrollo	69
3.7. Arquitectura del producto	70
3.8. Calidad exigida al producto	72
3.9. Catálogo de requisitos del sistema y prioridades	73
3.9.1. Tablas de requisitos	74
3.10. Especificación de subsistemas	79
3.10.1. Subsistema Interfaz de Parámetros	79
3.10.2. Subsistema Odometría Visual	84

3.10.3. Subsistema Configuración del entorno	88
4. DISEÑO E IMPLEMENTACIÓN	93
4.1. Introducción	93
4.1.1. Identificación del proyecto	93
4.1.2. Visión general del proyecto	94
4.1.3. Visión general del documento	94
4.2. Diseño de la arquitectura del sistema	95
4.2.1. Entorno tecnológico de implantación	95
4.3. Especificación de subsistemas de diseño	98
4.3.1. Relación de subsistemas de diseño	98
4.3.2. Entorno tecnológico de desarrollo	99
4.4. Diseño detallado de las interfaces de clases	101
4.4.1. Diagrama general de clases	101
4.4.2. Interfaces de clases	101
4.5. Estructura física de la base de datos y/o de los ficheros	107
4.6. Código fuente del producto	109
4.6.1. Nodo interfaz	109
4.6.2. Nodo odometría	111
4.7. Limitaciones de desempeño de la solución	114
4.8. Especificación del plan de pruebas	117
4.8.1. Pruebas de requisitos funcionales de la interfaz	118
4.8.2. Pruebas de requisitos funcionales de la odometría	119
4.8.3. Casos de prueba del subsistema interfaz	120
4.8.4. Casos de prueba del subsistema de odometría	125
5. MANUAL DE USUARIO	149
5.1. Introducción	149

5.1.1.	Identificación del proyecto	149
5.1.2.	Visión general del proyecto	150
5.1.3.	Visión general del documento	150
5.2.	Manual de usuario de la interfaz	151
5.2.1.	Requisitos necesarios	151
5.2.2.	Instalación del entorno	152
5.2.3.	Configuración de la interfaz: fichero de lanzamiento	154
5.2.4.	Configuración de la interfaz: diccionario de parámetros	156
5.2.5.	Arranque inicio y cierre de la interfaz	157
5.2.6.	Consultar el valor de un parámetro	158
5.2.7.	Modificar parámetros del driver en tiempo de ejecución	159
5.2.8.	Consultar la dirección de un topic	159
5.2.9.	Renombrar un topic	160
5.3.	Manual de usuario de la odometría.	161
5.3.1.	Requisitos necesarios	161
5.3.2.	Instalación del entorno	162
5.3.3.	Puesta en marcha y detención de la odometría	164
5.3.4.	Configuración de al odometría	164
5.3.5.	Utilización de las funciones básicas.	166
5.4.	Creación de ficheros de lanzamiento personalizados	171

6. MANUAL TÉCNICO

6.1.	Introducción	177
6.1.1.	Identificación del proyecto	177
6.1.2.	Visión general del proyecto	178
6.1.3.	Visión general del documento	178
6.2.	Recursos necesarios	179
6.3.	Instalación del entorno de programación	182

6.4.	Ficheros fuente	183
6.4.1.	Estructura general	183
6.4.2.	Ficheros de la interfaz	184
6.4.3.	Ficheros de la odometría	185
6.5.	Estructura de clases de la interfaz	186
6.5.1.	Clases del paquete interfaz	187
6.5.2.	Clase <i>driver_manager</i>	192
6.5.3.	Clase <i>iface_translator</i>	195
6.6.	Estructura de clases de la odometría	199
6.6.1.	Clases del paquete odometría	200
6.6.2.	Clase <i>myTransf</i>	207
6.7.	Configuraciones y parámetros	211
6.7.1.	Parámetros del nodo interfaz configurables durante el lanzamiento	211
6.7.2.	Parámetros para configurar el comportamiento de la odometría	212

Índice de figuras

1.1.	Representación gráfica del modelo actual	9
1.2.	Ejemplo de múltiples sistemas de referencia que pueden intervenir en una escena.	24
1.3.	Representación de los dos convenios utilizados para los sistemas de coordenadas. A la izquierda el obtenido directamente desde la cámara y a la derecha según el estándar del entorno.	25
1.4.	Imagen tomada directamente con la cámara de infrarrojos antes de calcular las profundidades	26
1.5.	Imagen de profundidad obtenida con el Kinect. Las zonas más oscuras están más cerca. Las negras son puntos no detectados.	27
1.6.	Representación bidimensional del acercamiento entre las nubes de puntos	30
3.1.	Diagrama de los subsistemas en los que se divide el sistema para el análisis	65
3.2.	Vistas de la arquitectura	67
3.3.	Diagrama de la arquitectura conceptual de la solución	70
3.4.	Diagrama de casos de uso del nodo interfaz	79
3.5.	Diagrama de casos de uso del paquete de odometría.	85
4.1.	Diagrama de subsistemas de diseño	98
4.2.	Logotipos de las tecnologías que forman el entorno de desarrollo (apartado 4.3.2)	100
4.3.	Diagrama general de clases del sistema.	101
4.4.	Bloque UML de la clase <i>upward_interface</i> del nodo interfaz	102
4.5.	Bloque UML de la clase <i>iface_translator</i> del nodo interfaz	103
4.6.	Bloque UML de la clase <i>driver_manager</i> del nodo interfaz	103

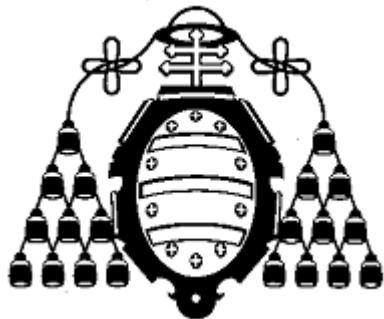
4.7.	Bloque UML de la clase odometryComm del nodo odometría (atributos)	104
4.8.	Bloque UML de la clase odometryComm del nodo odometría	105
4.9.	Bloque UML de la clase myTransf del nodo odometría	106
4.10.	Diagrama de la estructura física de datos	108
4.11.	Diagrama en árbol de la estructura del diccionario	108
4.12.	Nube de puntos a color correspondiente al escenario de las pruebas	128
4.13.	Ajuste de las nubes de puntos para las dos primeras pruebas visto desde el frente y desde arriba	130
4.14.	Pruebas de movimiento lateral. $y = +20$ cm en planta - $y = -20$ cm en perspectiva.	135
4.15.	Pruebas de movimiento diagonal en (x,y): en perspectiva a la izquierda, y a la derecha en planta.	135
4.16.	Pruebas de giros en el eje “z”	141
4.17.	Pruebas de giros con la cámara inclinada	141
5.1.	Ubuntu - Software Sources Panel	153
5.2.	Diagrama en árbol de la estructura del diccionario	156
5.3.	Pantalla inicial del configurador de parámetros dinámicos	158
5.4.	Lista de parámetros dinámicos configurables	159
5.5.	Ubuntu - Software Sources Panel	163
5.6.	Robot amigobot con el MS Kinect y sus dos sistemas de referencia	169
6.1.	Diagrama de clases que incluye las de la interfaz	186
6.2.	Diagrama de clases que incluye las del subsistema de odometría	199

Índice de tablas

2.1.	Fases e hitos contemplados en la planificación del proyecto	42
2.2.	Duraciones de las distintas fases del desarrollo	43
2.3.	Documentación a obtener tras las distintas fases del desarrollo	44
2.4.	Objetivos de las iteraciones del desarrollo	46
2.5.	Relación de recursos Hardware necesarios	47
2.6.	Relación de recursos Software necesarios	47
2.7.	Relación de recursos Humanos necesarios	48
2.8.	Cuadro de costes de los recursos hardware	49
2.9.	Cuadro de costes de los recursos software	49
2.10.	Cuadro de costes de los recursos humanos	50
2.11.	Cuadro de costes de los recursos hardware	50
2.12.	Cuadro de costes de los recursos software	50
2.13.	Cuadro de costes de los recursos humanos	51
2.14.	Presupuesto total	51
2.15.	Presupuesto final	51
3.1.	Clases de usuarios a los que se destina el sistema	62
3.2.	Grupo de requisitos 1: Características Generales	74
3.3.	Grupo de requisitos 2: Nodo Interfaz	75
3.4.	Grupo de requisitos 3: Nodo de Odometría (1)	76

3.5. Grupo de requisitos 3: Nodo de Odometría (2)	77
3.6. Grupo de requisitos 4: Requisitos No Funcionales	78
3.7. Caso de uso: “Leer valor actual de parámetro”	80
3.8. Caso de uso: “Modificar valor de parámetro”	81
3.9. Caso de uso: “Obtener ubicación de un topic”	82
3.10. Caso de uso: “Reubicar Topic”	83
3.11. Caso de uso: “Notificación de cambio en los parámetros”	84
3.12. Caso de uso: “Obtener estimación de odometría”	85
3.13. Caso de uso: “Reiniciar medidas. Puesta a cero”	86
3.14. Caso de uso: “Asignar nueva posición y orientación de la cámara”	87
3.15. Caso de uso: “Publicar las últimas estimaciones conocidas”	88
4.1. Tabla de pruebas de caja negra del subsistema Interfaz	120
4.2. Resultado de la prueba sobre <i>Leer valor actual de parámetro (opción A)</i>	121
4.3. Resultado de la prueba sobre <i>Leer valor actual de parámetro (opción B)</i>	121
4.4. Resultado de la prueba sobre <i>Modificar valor de parámetro (opción A)</i>	122
4.5. Resultado de la prueba sobre <i>Modificar valor de parámetro (opción B)</i>	122
4.6. Resultado de la prueba sobre <i>Obtener ubicación de topic</i>	123
4.7. Resultado de la prueba sobre <i>Reubicar topic</i>	123
4.8. Resultado de la prueba sobre <i>Notificación de cambio en parámetros</i>	124
4.9. Tabla de pruebas de caja negra del subsistema de odometría (funcionalidades)	125
4.10. Resultado de la prueba sobre <i>Obtener estimación de odometría</i>	126
4.11. Resultado de la prueba sobre <i>Reiniciar medidas. Puesta a cero</i>	126
4.12. Resultado de la prueba sobre <i>Asignar nueva posición y orientación de la cámara</i>	127
4.13. Resultado de la prueba sobre <i>Publicar las últimas estimaciones conocidas</i>	127

4.14. Tabla de pruebas de caja negra de la odometría visual	129
4.15. Resultado de la prueba sobre <i>Estimación de +20 cm en x</i>	131
4.16. Resultado de la prueba sobre <i>Estimación de -20 cm en x</i>	132
4.17. Resultado de la prueba sobre <i>Estimación de +20 cm en 'y'</i>	133
4.18. Resultado de la prueba sobre <i>Estimación de -20 cm en y</i>	134
4.19. Resultado de la prueba sobre <i>Estimación diagonal de 20 cm con "x" negativo e "y" positivo</i>	136
4.20. Resultado de la prueba sobre <i>Estimación diagonal de 20 cm con "x" positivo e "y" negativo</i>	137
4.21. Resultado de la prueba sobre <i>Giro de +28.5 grados en el eje "z"</i>	138
4.22. Resultado de la prueba sobre <i>Giro de -28.5 grados en el eje "z"</i>	139
4.23. Resultado de la prueba sobre <i>Giro de +28.5 grados en el eje "z" con cabeceo de 12 grados</i> .	140
4.24. Resultado de la prueba sobre <i>Giro de -28.5 grados en el eje "z" con cabeceo de 12 grados</i> .	142
4.25. Resultado de la prueba sobre <i>Cabeceo de 22.5 grados en sentido positivo</i>	143
4.26. Resultado de la prueba sobre <i>Cabeceo de 22.5 grados en sentido negativo</i>	144
5.1. Listado de topics de comunicación con el nodo de odometría.	167



UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE
GIJÓN

**ÁREA DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL**

PROYECTO FIN DE CARRERA N° 3122067

**INTEGRACIÓN DEL DISPOSITIVO 'MICROSOFT KINECT'
COMO PARTE DE UNA ARQUITECTURA ROBÓTICA
GENÉRICA PARA SU USO COMO DISPOSITIVO DE ENTRADA**

DOCUMENTO N° I

MEMORIA



MIGUEL A. GARCÍA GONZÁLEZ

NOVIEMBRE – 2012

TUTOR: LUCIANO SÁNCHEZ RAMOS

MEMORIA PRESENTADA POR

D. Miguel A. García González

**PARA OPTAR AL TÍTULO DE
INGENIERO EN INFORMÁTICA**

CAPÍTULO 1

MEMORIA

1.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

1.1.1. Identificación del proyecto

Título:	“Integración del dispositivo ‘Microsoft Kinect’ como parte de una arquitectura robótica genérica”
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

1.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo “Microsoft Kinect” comercializado por la empresa “Microsoft Corporation”. La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

1.1.3. Visión general del documento

En las siguientes páginas se pretende enmarcar el proyecto en su contexto durante lo que conforma la memoria del mismo, e introducir al lector en los conceptos, tecnologías y escenarios convenientes para comprender su motivación y desarrollo. Se realizará un breve repaso por la situación de la que surge el proyecto, las necesidades que trata de cubrir y problemas que las propician. También se explicarán las motivaciones que llevaron a tomar la solución ofrecida frente a otras posibles alternativas y se describirá brevemente el sistema resultante así como las ampliaciones propuestas sobre el mismo. La especificación y el diseño se encuentran detallados en profundidad en los documentos correspondientes sobre *Especificación y Diseño e implementación*.

Al final del documento se facilita un apartado de *Documentación Auxiliar* profundizando en las tecnologías y algoritmos usados para aclarar en qué consiste cada uno y, por tanto, qué utilidad tienen en el presente desarrollo, así como un *Glosario* de términos utilizados que pueden resultar familiares o no al lector, pero que trata de ser lo más amplio posible para cubrir todas las posibles dudas.

1.2. Ámbito y alcance del sistema

El proyecto a desarrollar surge en respuesta a unas necesidades determinadas y dentro de un contexto que es conveniente analizar en aras de buscar la mejor aproximación posible a la solución. Se definen, a continuación, el ámbito en el que se enmarca dicho proyecto, el alcance en cuanto a sus responsabilidades y las funciones que deberá realizar. Con esto se pretende tener una idea clara y completa que sirva como paso previo a la especificación de la solución.

1.2.1. Ámbito del proyecto

La robótica es una rama de la tecnología que nace con el propósito de crear dispositivos mecanismos autónomos y capaces, entre otras cosas, de realizar trabajo físico por sí mismos para liberar de éste a los seres humanos. Dicho campo lleva ya tiempo captando el interés de inventores, investigadores y profesionales; y se encuentra en pleno desarrollo por su evolución cada vez más rápida y su potencial para mejorar la vida de las personas. Sin embargo, existe hasta el momento una cierta tendencia a desarrollar soluciones excesivamente específicas (*ad-hoc*) que dificultan en parte la reutilización de recursos (tanto código como elementos físicos) y dificultan, así, el crecimiento de nuevos proyectos ralentizando la aparición de resultados.

A la vista de lo anterior, el grupo de investigación *Willow Garage*, junto con el laboratorio de inteligencia artificial de la universidad de Stanford crearon un entorno genérico para aplicaciones robóticas llamado “*Robot Operating System*” (en adelante *ROS*) que con una estructura de nodos individuales pero complementarios fuera capaz de implementar, mediante paquetes de uno o varios procesos, las funciones que cada usuario fuera necesitando para su robot. Así, los usuarios disponen de un entorno común genérico con gran cantidad de funciones y controladores ya implementados que ha ido evolucionando y mejorando rápidamente desde su creación en 2007 con personas y proyectos que se han ido uniendo para contribuir. Para más información sobre *ROS* se puede consultar el apartado 1.7.1 sobre “*Robot Operating System*” al final del documento *Memoria*.

Gracias a la existencia de un entorno genérico para robots se hace posible la creación de herramientas reutilizables que sirvan de apoyo en diferentes proyectos cada vez más complejos, como los que se llevan a cabo en el departamento de “*Sistemas Inteligentes Colaborativos*” del Centro Europeo por el progreso del Soft Computing (ECSC por su siglas en inglés), situado en Mieres (Asturias). La unidad se encarga de diversos proyectos relacionados con la interacción entre distintos robots y también de estos con humanos, y por ello han plasmado dos de sus necesidades actuales en el desarrollo actual: la creación de una interfaz genérica para comunicar más fácilmente distintas aplicaciones con diferentes dispositivos de entrada y robots (ya que si bien el entorno es genérico, los drivers no tienen asignados nombres iguales para recursos homólogos), y la incorporación del dispositivo Microsoft Kinect en particular como fuente para estimar la odometría (desplazamiento) utilizando los robots del modelo “*Amigobot*” de los que se dispone y que se describirán más ampliamente en otras secciones del documento.

1.2.2. Descripción general del proyecto

El proyecto propuesto consiste, concretamente, en el desarrollo de dos elementos software en forma de paquetes para el entorno ROS que sean capaces de ejecutarse sobre el susodicho entorno (incluso utilizados por separado) y cumplir dos cometidos:

- El primero de ellos, comunicándose directamente con los controladores de dispositivos, facilitará el acceso a sus recursos de forma genérica con nombres comunes incluso si los controladores pertenecen a periféricos distintos siempre y cuando los recursos sean comunes y estén definidos (por ejemplo: “PointCloud” puede ser una nube de puntos proveniente de una cámara de profundidad “Microsoft Kinect” o del dispositivo homólogo fabricado por la compañía Asus : “XTionPRO”), de tal forma que al cambiar el dispositivo sólo sea necesario especificar el nuevo fichero con los “nombres” de los recursos. Para ello se diseñará una “interfaz” para los drivers que conozca los distintos canales de comunicación de ROS y haga de intermediario entre las aplicaciones y los controladores siendo configurable por el usuario mediante un fichero externo.
- El segundo paquete ROS deberá ser una aplicación que, comunicándose bien con la interfaz anterior o directamente con los drivers, reciba capturas del entorno en forma de nubes de puntos con información del escenario en tres dimensiones y realice, a partir de cada par consecutivo que reciba, una estimación de la odometría del robot (el movimiento realizado) para calcular la posición y la orientación nuevas resultantes.

1.2.3. Objetivos y alcance

Se especifican ahora los objetivos que se extraen de las necesidades presentadas y de sucesivas entrevistas con un miembro del departamento. Estos determinarán el alcance del proyecto, que se ampliará de forma más concreta durante el análisis de requisitos, y se considera fuera del alcance todo lo que no esté explícitamente especificado en los últimos. Se enumerarán por separado los aspectos a cumplir para cada uno de los dos elementos software que se propone: la interfaz y el cálculo de la odometría.

El cometido de la interfaz a diseñar es actuar de intermediaria entre ciertas aplicaciones (capa superior en términos de abstracción) y diferentes controladores de dispositivos (capa inferior), y que el usuario pueda utilizar dicha interfaz sin necesidad de tener conocimiento alguno de programación para lograr que una aplicación se comunique con un dispositivo que no era conocido para ésta, siempre y cuando los recursos que requiere sean de la misma naturaleza (y tipo de datos). Planteado como un listado de objetivos se pretende:

- La interfaz será capaz de modificar los parámetros de los controladores a petición de otras aplicaciones, así como informar a estas de cambios en los primeros.
- Los parámetros y recursos podrán tener distintos “identificadores” (nombres por parte de las aplicaciones y de los drivers) que la interfaz será capaz de relacionar sirviéndose de un fichero de configuración preparado a tal efecto.
- Se hará posible manipular la configuración de la interfaz de forma simple y sin conocimientos de programación previos.

- El software deberá requerir la mínima interacción posible por parte del usuario una vez iniciada, siendo lo bastante autónoma como para no distraer su atención de forma innecesaria.
- Mediante el uso de ficheros de lanzamiento de ROS (y ficheros de configuración adicionales) se podrá configurar lo necesario para que la interfaz se adapte a cada par aplicación-driver de forma que estos se comuniquen.
- Se asegurará el correcto comportamiento del software en las condiciones esperadas y la adecuada notificación de errores cuando exista algún problema con el entorno.

En el caso de la odometría, la simplicidad es muy importante de cara a desarrollos posteriores por lo que se exige, únicamente, la posibilidad de calcular la transformación (el movimiento y rotación) entre cada dos nubes de puntos consecutivas, de forma que ese mismo comportamiento sea ampliable a un flujo de nubes de puntos y dejando como posible ampliación la implementación de información adicional en la respuesta del algoritmo como una evaluación de la fiabilidad de la medida y del estado del algoritmo para determinar si hubo algún error. La configurabilidad para adaptarlo a distintos casos es también imprescindible. Esto se traduce en las siguientes directrices:

- A partir de dos capturas (nubes de puntos) consecutivas de la cámara, debe ser posible calcular una estimación del movimiento realizado entre ellas (si éste se mantiene dentro de un margen determinado).
- Las estimaciones de odometría resultantes deben ser retransmitidas de forma tal que otra aplicación pueda recibirlas y utilizarlas.
- La estimación del movimiento se realiza a partir de la información de profundidad obtenida en forma de nubes de puntos, pero sin tener en cuenta la información adicional sobre el color debido a la posibilidad de utilizar otros dispositivos de entrada que no aporten tal información).
- La configuración del algoritmo deberá ser posible mediante el propio entorno *ROS* para máxima compatibilidad con él y simplicidad.
- La mayor cantidad de parámetros que sea posible deben ser configurables para adaptar el funcionamiento del algoritmo a los distintos escenarios y optimizarlo según los intereses del usuario.
- Se debe asegurar el correcto comportamiento del software en las condiciones esperadas y la adecuada notificación de errores cuando exista algún problema con el entorno.

Adicionalmente al comportamiento del software en tiempo de ejecución y de cara al usuario final, es imprescindible que el desarrollo esté especialmente orientado a facilitar un código fuente modular y comprensible destinado a su reutilización, pues se planea utilizarlo como base para futuros desarrollos.

1.3. Descripción del sistema actual

Con los recursos y condiciones disponibles actualmente, de forma previa al despliegue del presente desarrollo, el sistema lo forman los recursos físicos (robots, ordenadores y periféricos del robot) utilizados para las pruebas en el laboratorio del Centro *ECSC* que propuso el proyecto, y los recursos software de que se dispone para tratar de realizar las funciones que puedan ser requeridas. En adelante se detalla dicho sistema actual dejando entrever las carencias del mismo frente a las aportaciones esperadas de este proyecto.

1.3.1. Modelo actual

En el laboratorio de Sistemas Colaborativos Inteligentes del *ECSC*, en la actualidad, se dispone de un número determinado de ejemplares de robots modelo *Amigobot*, así como una serie de ordenadores de sobremesa, dos portátiles de marca *Asus eeePC* y dos ejemplares de cámaras de profundidad *Microsoft Kinect*. En cuanto al software del que se sirven dichos elementos: los ordenadores disponen tanto de Microsoft Windows como de Ubuntu Linux como sistemas operativos, y el *Amigobot* puede ser controlado de forma inalámbrica desde un sistema Windows mediante el software *MobileEyes* proporcionado por el fabricante, o bien conectándolo a un PC con el entorno ROS instalado (usando Ubuntu Linux en tal caso) y mediante el paquete ROS “generic_teleop” creado por uno de los miembros del laboratorio precisamente con la intención de poder controlar distintos robots utilizando un mismo paquete genérico.



Figura 1.1: Representación gráfica del modelo actual

Con los medios disponibles actualmente, y tal como se ilustra en la figura 1.1, un usuario puede controlar el robot manualmente mediante su software para Windows o bien con un sistema ROS y el ya mencionado paquete “generic_teleop”; o bien puede programar el comportamiento del robot mediante paquetes específicos de los disponibles para ROS que sean compatibles con el modelo *Amigobot*. Ciertos paquetes pueden estar programados incluso para incorporar el dispositivo Kinect como fuente de datos, como es el caso de “clearpath_kinect” o “robotino_kinect”, pero ninguno de los reconocidos oficialmente¹ está diseñado para ser compatible con los robots del laboratorio. En cualquier caso, en el sistema actual, la cámara de profundidad no está integrada en el robot, que depende de una serie de sónares para su orientación.

Tampoco se ofrece, por el momento, una solución que permita obtener imágenes o nubes de puntos de forma genérica desde diferentes dispositivos en distintos robots y permita generar, exclusivamente, los valores de la cada transformación o movimiento individuales a partir de las dos nubes de puntos correspondientes para ser utilizados en una segunda aplicación.

En cuanto al campo de la robótica en general, la mayoría de las aplicaciones son excesivamente específicas y no dan facilidades para ser utilizadas con nuevos periféricos o robots. El entorno ROS, si bien trata de romper esa tendencia, carece por ahora de soluciones genéricas para las funcionalidades arriba mencionadas. Los paquetes más afines están orientados a dispositivos concretos y tratan de construir un mapa mediante la alineación de todas las nubes de puntos en un solo escenario sobre el cuál realizar cálculos posteriores.

1.3.2. Lista de problemas y necesidades

Los principales **problemas** del modelo actual tienen que ver con la compatibilidad y el planteamiento excesivamente específico de las soluciones actuales, especialmente el software original de los robots que se pretenden utilizar inicialmente. Esto se traduce en los siguientes problemas:

- Utilizar software demasiado específico implicar usar controladores específicos para cada driver en cada robot así como versiones diferentes de las aplicaciones adaptadas a cada controlador: una gran limitación al elegir el hardware y enorme gasto de recursos al crear las aplicaciones.
- Al carecer de software específico para calcular movimientos entre cada par de nubes de puntos, todas las aplicaciones relacionadas con la odometría pasan por hacer una reconstrucción completa utilizando gran cantidad de memoria y procesamiento. El cálculo de dicho movimiento entre dos nubes es mucho más adecuado para las necesidades del proyecto.
- Otro problema consiste en el uso de código fuente complejo y ajeno al centro. Los paquetes ROS disponibles públicamente siguen una estructura diferente que no responde a las necesidades específicas de funcionalidad, diseño y mantenimiento que el laboratorio podría necesitar de cara a futuros desarrollos.

¹Listado de paquetes ROS oficiales: “ www.ros.org/browse/stack_list.php ”

Las **necesidades**, por otro lado, no son sólo vienen dadas por los problemas señalados, sino también optimizar en lo posible los recursos disponibles para el desarrollo de proyectos futuros por parte del centro.

- Es necesaria una aplicación capaz de realizar el cálculo de la odometría del robot utilizando cámaras de profundidad, para que ésta sirva como apoyo y complemento de la odometría de ruedas y, quizá, la estimada con los sónares.
- La estimación del movimiento debe poder realizarse individualmente para cada par de capturas con información de profundidad (nubes de puntos) siempre que éstas sean consecutivas y dentro de los márgenes permitidos.
- Para poder adaptar distintas aplicaciones a varios modelos de robots y periféricos, se necesita una herramienta o sistema capaz de “traducir” los identificadores de los recursos entre las dos partes, que no siempre son comunes para recursos iguales.
- Finalmente, es imprescindible poder utilizar el desarrollo actual como base o herramienta (según el caso) para otros nuevos en el futuro. Esto implica que, además de las funcionalidades, debe ofrecerse un código claro, modular y reutilizable y una documentación completa del mismo.

1.4. Análisis de alternativas

El presente apartado está orientado a especificar las distintas posibles alternativas que se plantearon de forma previa al desarrollo del proyecto de cara a poder analizarlas y elegir la más adecuada. Se trata de enumerar ordenadamente todas las posibles soluciones a los problemas planteados y analizarlas de manera que quede justificada la elección definitiva de una de ellas. Para este caso concreto, y dado que existía un planteamiento específico trazado desde el principio, son pocos los aspectos que requieren una decisión. Por ese motivo sólo se introducirán las alternativas a dichas cuestiones individuales y luego se describirá directamente la alternativa final seleccionada.

1.4.1. Discusión previa - Decisiones de diseño

Si bien las opciones a elegir para el desarrollo eran limitadas, sí hay algunas cuestiones de diseño que tienen su razón de ser y es apropiado que sean explicadas en esta sección. La parte física o hardware la dictan los elementos disponibles en el laboratorio como se detallará en el correspondiente subapartado, pero sí es necesario justificar el lenguaje de programación, las versiones del GNU/Linux y ROS utilizadas y el número de fuentes de datos a tener en cuenta para el cálculo, como se explicará a continuación.

Lenguaje de programación

El entorno ROS incorpora una serie de paquetes, herramientas y librerías a disposición de los usuarios para desarrollar sus propios paquetes. Estos elementos estaban escritos originalmente para C++, pero el lenguaje Python ya está completamente integrado e incluso se comienza a dar soporte a Java. Los dos primeros lenguajes son las opciones que se barajan para desarrollar el software objeto de este documento, a continuación se enumeran algunas características relevantes extraídas de sus webs correspondientes: “python.org” y “cplusplus.com” respectivamente.

Python es un lenguaje de programación dinámica que busca un alto nivel de abstracción para facilitar y agilizar el trabajo del programador, con lo que no está ideado para ser el más eficiente ni para su uso en dispositivos empotrados pero sí abarca las siguientes características:

1. Sintaxis muy clara y legible, una forma muy natural de escribir código procedural
2. Sigue el paradigma de orientación a objetos de forma intuitiva
3. Es completamente modular y soporta la inclusión de paquetes con una estructura jerárquica
4. Manejo de errores basado en excepciones
5. Tipos de datos dinámicos y de muy alto nivel de abstracción
6. Amplia gama de librerías estándar y módulos de terceros para prácticamente cualquier tarea

Es justo destacar, como se aclara en su web, que la diferencia de eficiencia respecto a otros lenguajes es muy baja como para ser perceptible (y aún menos determinante) para la mayoría de las tareas. La alternativa al primero es C++, que es comúnmente considerado muy eficiente por ser un lenguaje de bajo nivel y completamente compilado, por lo que sigue siendo utilizado en tareas intensas en procesamiento:

1. Está estandarizado según la norma ISO
2. Utiliza un tipado fuerte e inseguro
3. Permite tipado explícito e inferido
4. Soporta comprobación de tipos tanto estática como dinámica
5. Ofrece diferentes paradigmas de programación
6. Es portable gracias a la amplia variedad de compiladores disponibles
7. Gran cantidad de librerías disponibles

El primer paquete a desarrollar (la interfaz), no requiere gran eficiencia en tiempo de ejecución pero sí manipular los tipos de datos de las comunicaciones de forma lo más genérica e intuitiva posible. Además, es la primera parte a realizar y se solapa con el aprendizaje inicial del propio entorno ROS, con lo que un lenguaje simple e intuitivo supone una mayor ventaja, si cabe, durante esa primera etapa.

La odometría visual, por su parte, es un cálculo intensivo en procesamiento que, idealmente, deberá pasarse el máximo tiempo posible realizando cálculos para aprovechar la máxima cantidad de información (nubes de puntos) que sea posible. Esto es razón suficiente para que prime la eficiencia en el desarrollo del segundo paquete.

A partir del análisis anterior y considerando las características de cada lenguaje y las diferencias entre ambos paquetes a desarrollar, se decide utilizar lenguaje Python para la primera parte, terminar la interfaz lo antes posible y huir del tipado inseguro de C++ como principal precaución. La segunda parte, en cambio, se escribirá en C++ para agilizar al máximo los cálculos de odometría, siendo en este caso lo más importante optimizar el rendimiento.

Versiones del software

Se observó la necesidad de elegir las versiones tanto del sistema operativo de base, en el que se realizará la instalación de ROS, como del propio entorno ROS. La versión del intérprete Python y de los paquetes vendrá determinada por la versión disponible para cada uno en los repositorios si no se indica lo contrario.

La última versión de Ubuntu Linux disponible al inicio del desarrollo, la 11.04, es la primera alternativa frente a la opción de utilizar la última versión LTS (de mantenimiento prolongado): 10.04. A falta de un año para la siguiente versión LTS de Linux y ya que es deseable minimizar los cambios de versiones en lo posible, se decidió utilizar Ubuntu 10.04 de nombre *Lucid Lynx* como sistema operativo aunque con posibilidad de cambiarlo en las especificaciones finales si surgiera algún problema de versiones.

El entorno ROS, por su parte, tiene dos versiones estables durante el transcurso del proyecto: *Electric* y *Fuerte*. Su predecesora, llamada *Diamond*, fue ya clasificada como obsoleta. Dado que es deseable evitar

futuros cambios de versiones, se elige comenzar con la versión más reciente (*Fuerte*) y hacer uso de su alternativa sólo en caso de encontrar problemas que lo justifiquen. Las versiones de paquetes ROS serán las correspondientes a la versión y que estén disponibles en los repositorios de Ubuntu.

Fuentes de entrada de datos

En un sistema de odometría completamente robusto y completo, la presencia de información redundante o complementaria permite confirmar o mejorar la obtenida desde una fuente de datos con otra diferente. Para el proyecto se dispone de un relativamente amplio número de fuentes, a saber: cámara de color, cámara de profundidad, acelerómetros, micrófonos, sónar y codificadores de ejes (odometría de ruedas).

Por otro lado, de los objetivos contemplados en apartados anteriores se desprende que no es la robustez la mayor prioridad, sino la opción de sustituir la cámara de profundidad y la capacidad para utilizar diferentes dispositivos en general. Por ese motivo y debido al limitado alcance del proyecto, se ha decidido deliberadamente centrar el desarrollo en la obtención de un sistema adaptable y funcional que utilice únicamente la información de profundidad (es decir nubes de puntos provenientes de la cámara de profundidad) de la forma más correcta posible y de cara a introducir cambios en implementaciones futuras.

Librerías auxiliares

Una vez elegido el tipo de datos a manejar, los objetivos a satisfacer, el sistema operativo y versiones de software sobre las que podrá correr la solución y el lenguaje de programación, aún queda decidir las librerías adicionales que se utilizarán para importar funcionalidades sin implementarlas desde cero. Maximizar el número de funcionalidades “importadas” frente a las “implementadas” no es una decisión caprichosa; los proyectos más maduros y/o específicos son menos proclives a contener errores y, de haberlos, se pueden solucionar más rápidamente debido a la cantidad de usuarios y, posiblemente, desarrolladores. Reutilizar proyectos existentes también agiliza el desarrollo, reduce su mantenimiento y puede mejorar su calidad, que se traduce en menor coste y mejor producto.

La primera funcionalidad que debe externalizar es el tratamiento de ficheros de configuración en la interfaz. Se necesita un lenguaje estándar como pueden ser “XML” o “YAML” (que se introducirá en los apartados técnicos). Siguiendo las tendencias de otros desarrolladores, y muy especialmente de ROS, se opta por utilizar YAML al igual que en los ficheros de parámetros manejados por el entorno. Tratándose de una función básica, bastó con utilizar la librería más popular para el lenguaje de la interfaz (Python), es decir: “pyYAML”.

El tratamiento de las nubes de puntos de la odometría no es en absoluto trivial y, de hecho, existe sólo una solución popular y especializada que sea compatible (casi completamente) con ROS. Se trata de la librería de nubes de puntos o “Point Cloud Library”, en adelante *PCL*, desarrollada y mantenida de forma colaborativa por desarrolladores individuales, universidades, fabricantes de hardware, etc². De los recursos que ofrece la librería, el algoritmo iterativo del punto más próximo (*ICP* por sus siglas en inglés “Iterative Closest Point”) será el punto central que permitirá extraer la información de odometría de las escenas tridimensionales; su naturaleza y funcionamiento se explican en el apartado 1.7.5 de esta *Memoria*.

²La información sobre los colaboradores se encuentra en la página web del proyecto:
[“<http://pointclouds.org/about>”](http://pointclouds.org/about)

Elementos hardware disponibles

Si bien no cabe decisión alguna relativa al hardware, pues se trata de utilizar el que hay disponible, sí que tiene sentido una pequeña explicación de qué elementos hay disponibles en el laboratorio.

Obviando que podrían ser incorporados nuevos robots en un futuro cercano (de ahí el carácter genérico del desarrollo), los ejemplares disponibles actualmente en el laboratorio son los llamados “*Amigobot*” de la empresa “*Mobile Robots*”, que se describen en el apartado 1.7.4 y de los que se guardan unos 8 ejemplares para estudiar las posibles interacciones programadas entre ellos. Dos dispositivos *Kinect* de *Microsoft* fueron adquiridos antes del inicio del proyecto para ser conectados a los robots y dotarles de mayor información sobre el entorno (estos incluyen cámara de profundidad, cámara a color y un vector de micrófonos). En todo caso, y para el proyecto actual, sólo se tendrán en cuenta un robot y un *Kinect* que pueda realizar capturas para las pruebas correspondientes.

En cuanto a los ordenadores del laboratorio, el proyectante cuenta con un ordenador de sobremesa, un ordenador portátil *HP 620* propio y 2 portátiles *Asus eeePC* compartidos con otros compañeros. Si bien lo ideal es poder instalar la presente solución en uno de los últimos, se ha elegido usar el de *HP* como objetivo debido a la limitada potencia de los portátiles *Asus*. La utilización final de estos últimos dependerá del rendimiento del algoritmo en la práctica y el anterior es, al mismo tiempo, el PC utilizado para la programación.

1.4.2. Descripción de la solución

Para describir la solución elegida se expondrá, primeramente, una descripción del producto especificando las decisiones tomadas frente a las alternativas disponibles, y después se explicarán las características de hardware y de software respectivamente. Finalmente se realizará una valoración de la solución elegida y qué características podrían mejorarse o ampliarse más allá del alcance.

Descripción del producto

Expuestos los apartados anteriores y en base a los objetivos deseados, se desprende que la solución deberá estar dividida en dos partes conformando sendos paquetes del entorno ROS que resuelvan, por una parte, la comunicación entre las aplicaciones ROS y las diferentes versiones o modelos de los dispositivos correspondientes, y brinden, por otra parte, la opción de obtener estimaciones individuales para los movimientos del robot. No se puede descuidar tampoco la necesidad de que la comunicación sea configurable y el algoritmo de odometría utilizable en el mayor rango posible de casos y escenarios, que son características irrenunciables para los propósitos del laboratorio.

Concretamente se propone un primer paquete llamado *my_adaptor* que se encargue de realizar una “traducción” entre los recursos solicitados por las aplicaciones y aquellos que son ofrecidos por los controladores. De ese modo, las aplicaciones serán capaces de comunicarse con esta interfaz (asumiendo la configuración adecuada de la misma), y los controladores recibirán los cambios necesarios o enviarán información del mismo modo que si la interfaz fuera la aplicación final que los utiliza.

Por su parte, el segundo paquete (de nombre *my_odometry*), será en sí mismo una aplicación de *alto*

nivel en el sentido de que realiza sus cálculos a partir de las nubes de puntos que obtiene a través de los drivers desde los periféricos de entrada. Así se obtendrá una matriz de transformación que describa el movimiento entre las dos nubes de puntos y se podrá comunicar el resultado a otras aplicaciones que lo soliciten.

A priori cada paquete ejecutará un proceso individual (o “nodo”, si se utiliza la nomenclatura de ROS). Éstos llevarán nombres en inglés del tipo “my_funcionalidad”, en los que “my” representa, dentro del laboratorio que solicitó el proyecto, productos que no están oficialmente publicados y aún carecen de nombre final.

En cuanto a la parte física, y sin detrimiento de que el diseño vaya a ser lo más genérico posible, el hardware de destino será un ordenador portátil *HP 620* conectado a un robot modelo *Amigobot* y a una cámara de profundidad *Microsoft Kinect* que sirva como dispositivo de entrada. Los detalles sobre dichos elementos corresponden al siguiente subapartado (1.4.2) de esta documentación.

Características hardware

Se explican, a continuación, los detalles correspondientes a la parte física del proyecto entendiendo como tal ordenadores, robots y periféricos de entrada; si bien es necesario aclarar que en ningún caso se trata de partes desarrolladas durante el mismo sino de los elementos destinatarios del software sobre los que funcionarán las partes creadas. Las conexiones físicas y de comunicación entre dichos elementos también es materia a tratar en esta sección.

Si bien el proyecto trata de ser adaptable como ya se ha mencionado, el robot hacia el que se orientan las pruebas inicialmente es uno de los llamados *Amigobot*; un autómata de pequeño tamaño con forma de prisma ovalado con dos ruedas motrices y una rueda que pivota libremente (conocida como “caster”) en la parte trasera. Incluye como sensores un sistema de sónar, codificadores rotatorios en las ruedas y un giroscopio (según el modelo) que no serán utilizados en este desarrollo. Un pequeño procesador integrado permite procesar pequeños programas creados en C/C++ y una antena “Wireless Serial Ethernet” habilita la operación inalámbrica a distancia del robot. Además de la conexión inalámbrica, el robot ofrece la opción de conectarse al PC mediante un cable ethernet (RJ48).

El único dispositivo de entrada que se espera conectar al robot, y que es imprescindible para realizar las medidas de profundidad y la odometría visual, es una cámara de profundidad *Microsoft Kinect*, que es la que obtiene la información tridimensional del escenario y permite generar las nubes de puntos utilizadas para los cálculos. La cámara no requiere ningún tipo de preparación especial más allá de la conexión USB a un PC con los controladores y paquetes ROS adecuados para su funcionamiento.

Finalmente, es imprescindible la utilización de un PC que incorpore el software *ROS* (ya introducido con anterioridad) para que sirva como nexo de unión y comunicación entre el robot y la cámara, tanto a nivel físico (con sendas conexiones al PC) como de software, al ser ROS el intermediario entre los dos anteriores tal como se explica en la siguiente sección. Se detallan a continuación las características recomendadas para el ordenador que debe ejecutar el software:

- Procesador Intel Pentium T4500 (2.30 GHz) o superior
- 2 GiB de Memoria RAM

- 20 GiB de espacio en el disco duro
 - (incluyendo un pequeño margen para actualizaciones y ficheros temporales)
- Teclado y ratón para configurar y controlar el entorno

Asumiendo que se cumplan los requisitos anteriores, los paquetes desarrollados deberían funcionar correctamente y de forma estable si se utiliza adecuadamente tal como se explica en el documento de *Manuales de usuario* y dentro de dentro de las limitaciones que se exponen al final de dicho documento.

Características software

En esta sección se aclaran los detalles a nivel de software sobre la solución desarrollada ya introducida y la plataforma elegida para su adecuada ejecución, así como los requisitos no físicos adicionales que se necesitan para garantizar el correcto funcionamiento de la primera.

El entorno básico para el que se desarrolla la presente solución es el sistema *ROS* que ya se introdujo anteriormente por ser el entorno robótico genérico utilizado en el laboratorio. *ROS fuerte* es la versión estable al inicio del proyecto y la elegida para el desarrollo. Dicho sistema debe ser instalado en un PC con sistema operativo *GNU/Linux* que, como ya se discutió en las alternativas, será una distribución *Ubuntu* en su versión 10.04, de nombre *Lucid Lynx*. De todas formas, es lo esperable que los paquetes funcionen bien en versiones posteriores si se sigue manteniendo la compatibilidad en ambos productos.

Los paquetes que forman la solución en sí y sus funciones son las que se indican en la descripción del producto (1.4.2) si bien deben cumplir algunos requisitos comunes. Ambos paquetes deben ser reutilizables en distintos escenarios y fácilmente modificables y mantenibles para proyectos futuros, todo ello sin perjuicio de su facilidad de utilización y configurabilidad. Se espera que ambos paquetes sean capaces de funcionar sin la interacción del usuario en las condiciones básicas esperadas, notifiquen de forma ordenada posibles problemas en caso de haberlos y proporcionen algún medio para conservar su configuración entre ejecuciones de forma fácil y sin modificar el código fuente.

Es importante mencionar, al respecto de la odometría visual, que no es su objetivo obtener un gran rendimiento o la máxima fiabilidad y precisión (debido principalmente al ruido en las medidas), sino una base sobre la que realizar mejoras futuras en función de las necesidades. Se espera, de hecho, solucionar esos aspectos tanto con recursos físicos más avanzados (memoria, procesamiento...) como con ayuda de la información redundante entre las nubes de puntos, aceleraciones, odometría de ruedas, etc.

1.4.3. Valoración de la solución

Ya explicadas las características individuales seleccionadas para a la solución final y discutidas las elecciones tomadas en los casos con alternativas, queda todavía realizar una valoración del conjunto resultante observando las ventajas y desventajas que conlleva. Es necesario tener en cuenta que, tratándose de una solución robótica y haciendo uso de un entorno genérico específico como es ROS, las limitaciones son, por un lado, los recursos físicos disponibles a los que el proyecto está orientado; y por otro lado, las capacidades que ofrece el entorno ROS para la comunicación y el manejo de los dispositivos físicos, así como el tiempo disponible para el desarrollo del proyecto.

Se expone a continuación un listado con las ventajas y desventajas que ofrece la solución elegida frente a la situación actual como resultado de las alternativas elegidas y teniendo en cuenta las necesidades originales del proyecto.

Ventajas

La solución propuesta es una solución a medida para los problemas y objetivos descritos al principio del documento y con los que se cumple, por tanto, en la medida de lo posible. Más concretamente, se siguen los objetivos principales buscados en el diseño que son la traducción de nombres de recursos en la interfaz (para la compatibilidad entre aplicaciones y dispositivos) y la obtención de estimaciones del movimiento con el algoritmo de odometría. Los objetivos no funcionales también siguen presentes: configurabilidad a nivel de usuario (mediante parámetros y ficheros del entorno ROS) y de ser mantenible y reutilizable a nivel de código.

A continuación se expone una lista de las implicaciones prácticas de la solución:

- Un usuario puede escribir fácilmente su fichero de configuración (o de “traducción”) que relacione los nombres de los recursos por parte de la aplicación y del controlador que utilizará.
- Una misma aplicación puede comunicarse con diferentes dispositivos y viceversa siempre que la interfaz use el fichero de configuración correcto para ellos.
- Con la interfaz será posible asignar una configuración por defecto para los controladores distinta de la que incorporen originalmente.
- Se pueden realizar estimaciones de movimiento (odometría) para dos nubes de puntos individuales sin necesidad de almacenar en memoria todo un mapa completo del entorno.
- La configuración de ambos nodos (o procesos, que designan lo mismo en entornos ROS) se puede completar mediante ficheros de lanzamiento y parámetros del entorno ROS para adaptarlas rápidamente a diferentes escenarios.
- Un código fuente ya orientado a futuras adaptaciones permitirá realizar mejoras en el comportamiento de los nodos cuando resulte necesario. Si se descubre un algoritmo nuevo y mejorado para la parte de odometría o cambian las necesidades o el contexto en que se usa, basta modificar las líneas indicadas en el documento de *Manual Técnico* para adaptar y mejorar el software.

Desventajas

A pesar del esfuerzo realizado para obtener una solución que aproveche al máximo los recursos, existen una serie de limitaciones de tiempo y funcionalidad, así como prioridades ya comentadas en las discusiones de las decisiones de diseño (1.4.1). Eso implica que algunas funcionalidades “esperables” o alternativas atractivas, tienen que ser desestimadas para cumplir con las restricciones.

Las desventajas más triviales tienen que ver con cosas que podrían ser mejoradas o agilizadas, aunque funcionen correctamente:

- Resulta tedioso escribir un fichero de configuración, ya que se plantea escribir uno a uno los nombres de los recursos y su identificador en el driver correspondiente. Esto no implica que no resulte fácil pero sería menos tedioso si fuera automático.
- El uso de la odometría visual conlleva un tiempo de procesamiento alto debido a su complejidad. Pueden existir optimizaciones pero requieren tiempo de desarrollo y pueden depender de cada escenario.
- Al restringir el uso de recursos (cámara a color o memoria para los mapas), la odometría sólo es una herramienta de apoyo que dependerá de otros módulos o nodos, pues la precisión y fiabilidad son bajas con información limitada.

También hay limitaciones del entorno y del hardware utilizado como las siguientes:

- ROS todavía no permite la reubicación real de los canales de comunicación. Simularlo conllevaría sobrecarga en el procesamiento o esfuerzo para el usuario respectivamente, por lo que esto limita la utilidad de la interfaz.
- La precisión de la cámara de profundidad limita la cantidad de capturas que se pueden realizar (a menor movimiento más error relativo). Afortunadamente es posible sustituirla fácilmente o complementarla con otros sensores en el futuro.
- El algoritmo (*ICP*) utilizado, además de consumir tiempo, tiene limitaciones inherentes que limitarán la velocidad del robot. La facilidad para implementar un algoritmo más adecuado o mejorar el actual restan importancia a este aspecto.

Estimación de los costes

La validez de la solución y el posible interés en llevarla a cabo, están sujetos también a la relación coste/beneficio que se espera obtener enfrentando los gastos generados a lo largo del desarrollo (que se puede estimar de forma cuantitativa) con el valor del mismo tras la creación del producto final. Al ser una solución dirigida a un centro de investigación, sin embargo, no se planea obtener un beneficio económico directo sino una herramienta que facilite y agilice el trabajo y cuyo valor sólo se puede observar como una ventaja cualitativa.

Para realizar una estimación previa aproximada del coste de la solución propuesta, éste puede desglosarse, a grandes rasgos, en tres puntos diferentes que se valorarán por separado:

- El coste de desarrollo de la aplicación que se valorará teniendo en cuenta la planificación del proyecto que se detalla en el siguiente documento: *Planificación y Presupuesto* .
- El importe del equipo en términos de recursos Hardware y Software.
- Los costes de implantación del sistema que, al no ser requisito en este caso, no serán considerados.

Los detalles de este estudio sobre los costes, pueden encontrarse en el apartado específico sobre *Planificación de los costes* (2.5) que contiene todo el desglose y los cálculos para el valor estimado del proyecto.

1.5. Posibles ampliaciones

La cantidad de ampliaciones que se pueden plantear a partir del desarrollo es muy alta y potencialmente creciente a medida que se utilicen más recursos y que el entorno de base (ROS) ofrezca nuevas y mejores posibilidades. Por ello se enumeran a continuación, en una lista, sólo las principales opciones que surgieron durante el desarrollo y no llegaron a materializarse.

- **Reubicación de *topics* (canales de comunicación).** Actualmente implicaría una sobrecarga del procesador pero ampliaría la utilidad real de la interfaz si pudiera realizarse de forma interna.
- **Creación automatizada de ficheros de configuración.** Dado que la configuración de la interfaz será la parte más tediosa (a pesar de requerir pocos conocimientos), un generador de estos agilizaría el trabajo notablemente.
- **Redundancia de sensores de movimiento.** Ya en otros desarrollos se utilizan datos redundantes para mejorar los resultados de la odometría. Las ruedas no son precisas pero sí muy fiables al determinar si se realizó un giro o un avance y en qué dirección, reduciendo enormemente el campo de búsqueda del algoritmo. Lo mismo ocurre con acelerómetros, sónar, triangulación de señales inalámbricas, etc.
- **Parámetros más amigables en la odometría.** Poder configurar el algoritmo según la velocidad física del robot o el ángulo de visión de la cámara en vez de conocer los parámetros internos en que se traducen haría el software mucho más amigable.
- **Optimizaciones de la odometría.** Aunque no procede entrar en detalle, los algoritmos de odometría visual disponibles son aún susceptibles de ser mejorados, especialmente con información previa del entorno (como la certeza de que el suelo será visible, por ejemplo) y mediante el uso de una tarjeta gráfica.

1.6. Documentos que conforman la documentación

Tal como se indica en el reglamento, se ofrece a continuación una relación de los documentos que se pueden encontrar incluidos en la presente documentación.

- 1. Memoria** Incluirá la descripción inicial del sistema desarrollado e información adicional sobre el producto, así como información adicional de interés para comprender el proyecto.
- 2. Planificación y presupuesto** Una estimación previa del tiempo que debe durar el proyecto con cada una de sus etapas y un análisis del coste económico que supone.
- 3. Especificación del sistema** Estudio anterior al diseño desde un punto de vista no técnico de las características que tendrá el producto y las funcionalidades que deberá implementar.
- 4. Diseño e implementación** Especificación definitiva del diseño que se implementará y que, a su vez, será plasmado en la segunda parte del mismo documento incluyendo las pruebas de aceptación sobre la solución resultante.
- 5. Manual de usuario** Manual de utilización del producto explicando el proceso de instalación y las funcionalidades disponibles de cara a los usuarios del mismo.
- 6. Manual técnico** Documento orientado a cualquier desarrollador que se disponga a reutilizar el código fuente del producto y desee comprender mejor su funcionamiento interno y el modo de realizar cambios en su comportamiento.

1.7. Documentación auxiliar

1.7.1. Robot Operating System (ROS)

En el campo de la robótica existe una cierta tendencia, por parte de los fabricantes, a la comercialización de dispositivos que no siguen convenios comunes ni responden a la misma programación, especialmente entre marcas diferentes. Como resultado, es común la creación de soluciones específicas (*ad hoc*) y sin tener en cuenta una posible reutilización del código para futuros desarrollos.

El entorno “Robot Operating System”, en adelante “ROS”, es un framework para aplicaciones robóticas que trata de romper con la dinámica de las soluciones *ad hoc* ofreciendo un marco de desarrollo y ejecución común en la medida de lo posible y con una estructura adaptable y modular, evitando que partes innecesarias del entorno tengan que ser cargadas en memoria y/o ejecutadas provocando sobre carga. Este producto, mantenido actualmente por el laboratorio de investigación *Willow Garage* pero abierto a colaboraciones, presenta una estructura de nodos o procesos individuales con funcionalidades simples y separadas. Éstos pueden comunicarse a través del entorno para colaborar en tareas comunes y se distribuyen agrupados en *paquetes* de nodos que trabajan juntos y estos a su vez en *pilas*.

La definición de ROS por parte de sus creadores en la página web oficial³ amplía la provista en las líneas superiores:

«Un “meta sistema operativo” de código libre que proporciona los servicios que se pueden esperar de un sistema operativo: abstracción de hardware, control de dispositivos a bajo nivel, implementación de funcionalidades de uso común, comunicación entre procesos mediante el uso de mensajes y gestión de paquetes. Además incluye herramientas y librerías para obtener, escribir, generar y ejecutar código repartido entre varios equipos u ordenadores.»

Se crea ROS con la intención de favorecer la reutilización e intercambio de código y, de ese modo, que el trabajo de distintas personas sea aprovechado para agilizar los desarrollos de otras. Dicho en las palabras de los propios desarrolladores:

«El objetivo principal de ROS es apoyar la reutilización de código en el desarrollo y la investigación sobre robótica. ROS es un framework distribuido de procesos que (...) pueden ser agrupados en *paquetes* y *pilas* que se pueden compartir y distribuir fácilmente.»

³La definición de ROS se encuentra bajo la dirección: “www.ros.org/wiki/ROS/Introduction”

En cuanto al funcionamiento de ROS se destacan los siguientes aspectos:

- **A nivel de framework:** ofrece un conjunto de librerías y herramientas que permiten crear los nodos y que estos se comuniquen siempre por medios comunes, se ejecuten mediante lanzadores estándar y utilicen ficheros de configuración con el mismo formato si se desea.
- **Para las comunicaciones:** implementa de forma nativa tres clases de comunicación:
 - **Servicios.**- una implementación similar a las “Llamadas a procesos remotos” de otros entornos (“RPC” por sus siglas en inglés) permite a los nodos proveer de funciones a otros nodos y, por lo tanto, una interfaz intuitiva para que estos puedan comunicarse con los primeros. Gracias a los servicios es posible realizar solicitudes de reubicación de topics en la interfaz o solicitar una nueva lectura a la odometría visual.
 - **Parámetros.**- un servidor global de parámetros en el que cada nodo tiene su espacio privado pero puede acceder a los de los otros nodos es el medio más simple de conocer el estado de otros nodos y dar a conocer el propio. En las últimas versiones se ha introducido el “Servidor de Reconfiguración Dinámica” (Dynamic Reconfigure Server) que permite desencadenar un aviso automático cuando se observan cambios en los parámetros.
 - **Topics.**- los llamados “topics” son canales de envío y recepción que se mantienen abiertos constantemente permitiendo flujos continuos (o no) de mensajes. Tienen asignada una “dirección” bajo la cual trasmitir los datos que pertenecen al mismo flujo. Es la comunicación más directa dado que cualquier nodo puede publicar datos en cualquier topic y todos pueden leerlos instantáneamente, así que se utiliza para los flujos continuos de datos.

La utilización de estos tres canales está disponible por defecto en las librerías ROS para cualquier nodo que sea instanciado en el entorno y no existen permisos ni restricciones, por lo que todos los nodos los utilizan.

- **En tiempo de ejecución:** depende de un núcleo llamado “roscore” que gestiona la publicación de los nodos en forma de recursos del entorno (con un nombre único dentro de un espacio de nombres y con las características internas que sean necesarias) y todas las comunicaciones entre los procesos ejecutados. Nada funciona si el núcleo no está ejecutado.
- **Desde el punto de vista del usuario:** tanto el núcleo como las herramientas de usuario (nodos proporcionados como parte del entorno), están programados en C++ para correr sobre un sistema operativo GNU/Linux con posibilidad de funcionar en Mac OS X y está en desarrollo una versión para Microsoft Windows. En ningún caso se espera que los nodos sean dependientes del sistema operativo de base.

El sistema ROS es la base de todo el desarrollo de la presente documentación. Si se desea más información general sobre él, ésta puede ser ampliada en su web “www.ros.org”. Los tipos de comunicaciones, estructura y conceptos se explican más a fondo en: “www.ros.org/wiki/ROS/Concepts”.

1.7.2. Sistemas de referencia y matrices de transformación

Existen dos conceptos ineludibles al tratar con movimientos en espacios tridimensionales: las matrices de transformación que los describen y el sistema de referencia con respecto al cual se interpretan. Aunque son conceptos relativamente básicos es necesario aclarar algunos aspectos relevantes para este contexto en particular.

- Durante la presente documentación se denomina *sistema de referencia* a un sistema de coordenadas cartesianas formado por un punto de origen y tres ejes de coordenadas ortogonales entre sí que definan un espacio euclídeo. Así, toda posición y movimiento vienen asociados a un sistema de referencia con respecto al cual realizar las medidas. En el sistema propuesto existen, al menos, tres: el de la cámara, el del robot y el considerado para el “*mundo*”.
- **Los movimientos de la cámara** son estimados respecto a su propio sistema de coordenadas (como se aprecia en el ejemplo de la figura 1.2) por lo que sólo son extrapolables al del robot si se conocen la posición y orientación de la cámara respecto a él. En cualquier caso basta con almacenar la información de posición/orientación de la cámara respecto al robot en una matriz de transformación para poder relacionar ambos sistemas de referencia y conocer el movimiento del conjunto. El sistema final implementa ya dicha operación como parte de su funcionalidad.

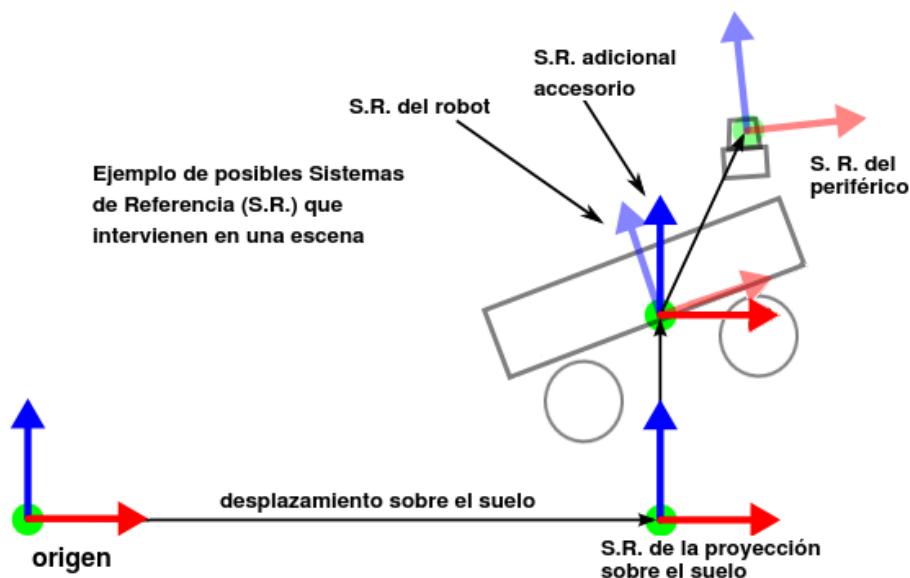


Figura 1.2: Ejemplo de múltiples sistemas de referencia que pueden intervenir en una escena.

- **El convenio para las coordenadas** varía entre la representación de nubes de puntos (junto con la estimación del algoritmo) y el estándar de ROS⁴. El primero parte de una idea conceptual distinta: los puntos forman un mapa de bits con valor de profundidad, es decir, que 'x' crece hacia la derecha, 'y' hacia arriba y 'z' se aleja hacia el frente. El convenio de *Robot Operating System* difiere al considerar que el eje 'x' es ortogonal a 'y' sobre el plano del suelo y 'z' completa, con la altura, la tridimensionalidad del espacio. Según el convenio de la mano derecha: 'x' crece hacia el frente, 'y' hacia la izquierda y 'z' paralela a la altura real.

⁴Los convenios utilizados por ROS se encuentran definidos en la documentación bajo la dirección:

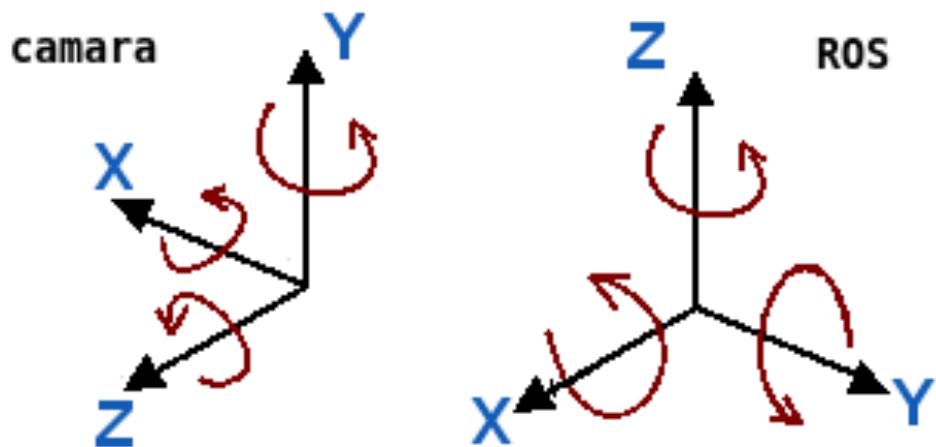


Figura 1.3: Representación de los dos convenios utilizados para los sistemas de coordenadas. A la izquierda el obtenido directamente desde la cámara y a la derecha según el estándar del entorno.

En ambos casos se utiliza la regla de la mano derecha para el sentido de los giros: se alinea el pulgar con el eje señalando el sentido en el que éste crece y el resto de los dedos, formando un puño, señalan el giro considerado positivo. Las publicaciones se realizarán siempre contemplando el convenio de ROS.

- Las *matrices de transformación*, por su parte, están formadas por los valores necesarios para describir el movimiento completo de un elemento (desplazamiento y rotación) respecto a un sistema de coordenadas previamente conocido. El producto de dos matrices de transformación es una tercera con la combinación de sendos movimientos, por lo que son acumulables; e interpretadas respecto al origen de coordenadas coinciden con una posición y orientación finales.

Dado que existen múltiples representaciones para los movimientos y posiciones relativas en el espacio, se utiliza una estructura nativa de ROS, independiente de la implementación, llamada “transform frame” (marco de transformación), y que se abrevia como “**TF**”. Las *TFs* representan la posición y orientación de un sistema de coordenadas con respecto a otro en el espacio y, por extensión, con respecto a sí mismo en un instante anterior. Las relaciones entre los elementos de la ilustración 1.2 se pueden definir mediante estos elementos.

^{“<http://ros.org/wiki/geometry/CoordinateFrameConventions>”}

1.7.3. Cámara de profundidad *Microsoft Kinect*

El periférico *Microsoft Kinect*, en adelante *Kinect*, es un dispositivo de juego desarrollado por *Microsoft Corporation* para interactuar con la consola *XBOX 360* mediante una interfaz natural de usuario: gestos, comandos de voz, objetos e imágenes, etc. Para ello cuenta con un tandem de cuatro micrófonos, acelerómetro, una cámara RGB y una cámara de profundidad; además de un pequeño motor para modificar la inclinación (en el eje 'y': cabeceo). En lo que a este proyecto respecta, se alude al Kinect como si de una cámara de profundidad se tratara pues la utilización del resto de sensores se encuentra fuera del alcance y no intervienen en la adquisición de información tridimensional del entorno ni entran en conflicto con dicho proceso.

Las características de la cámara de profundidad vienen definidas por su ángulo de visión: 43º en vertical y 57º en horizontal; su resolución: 640x480; y el rango de profundidades que percibe: desde 0.8 hasta 3.5 metros según algunas⁵ especificaciones. En cuanto al funcionamiento, se introduce brevemente a continuación ya que puede tener implicaciones durante su uso como occlusiones inexplicadas, reflejos imprevistos o materiales no detectados:



Figura 1.4: Imagen tomada directamente con la cámara de infrarrojos antes de calcular las profundidades

Físicamente se utilizan un proyector de luz infrarroja y una cámara de infrarrojos corriente capaz de percibir la proyección del primero. El proyector “emite” una matriz de puntos ordenados según una serie de patrones determinados y conocidos de antemano. La cámara captura imágenes del escenario en las que aparecen las proyecciones anteriores distorsionadas en función de los objetos sobre los que se proyectan. Esto se ilustra en la figura 1.4.

Dado que las distorsiones del patrón de puntos dependen de las distancias de los obstáculos alcanzados, es posible reconstruir dichas distancias generando, así, una mapa de profundidades como el de la figura 1.5. Los valores resultantes son, a su vez, susceptibles de ser convertidos en una nube de puntos que se aproxima a una reconstrucción tridimensional de la escena.

⁵Las especificaciones son variables en función de los controladores: los drivers oficiales más actuales (en este caso se utiliza software no oficial) prometen una distancia máxima de 4 metros y un modo “near” capaz de distinguir distancias por encima de 40 centímetros: “<http://www.microsoft.com/en-us/kinectforwindows/discover/features.aspx>”

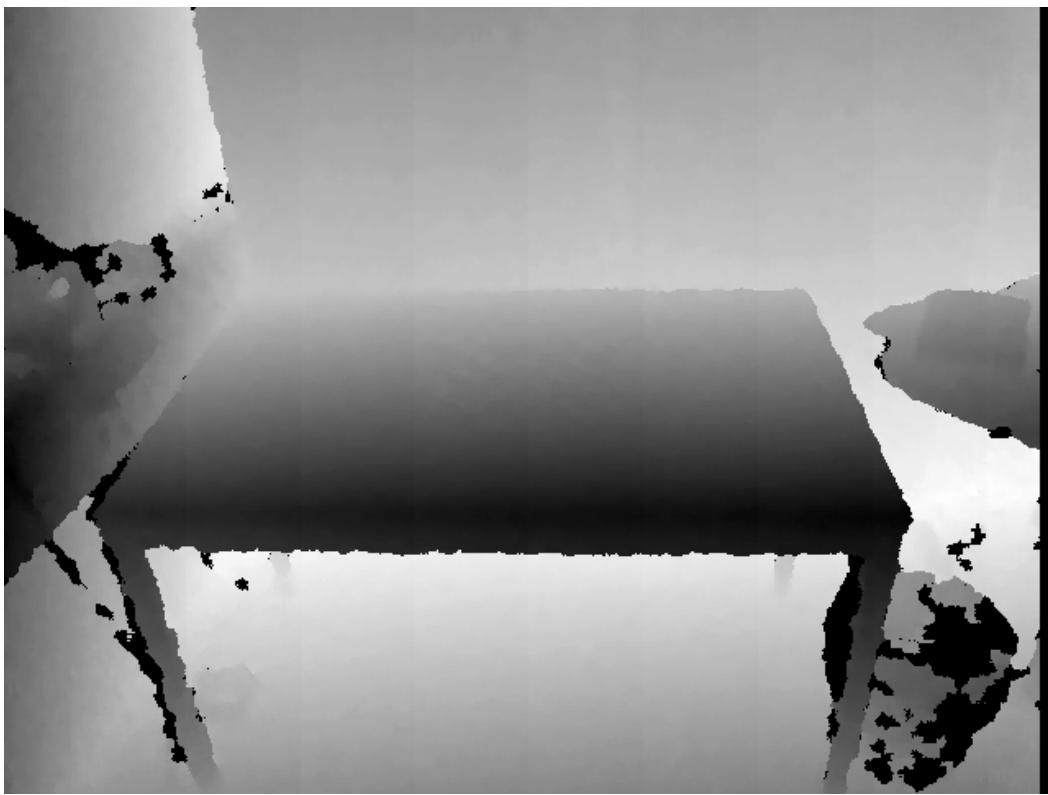


Figura 1.5: Imagen de profundidad obtenida con el Kinect. Las zonas más oscuras están más cerca. Las negras son puntos no detectados.

En ambas imágenes de ejemplo se pueden visualizar occlusiones en las zonas a donde no llega la proyección, y en la segunda se observa cómo el extremo de la papelera tampoco es detectado, posiblemente por su inclinación respecto a la proyección. Conocer las técnicas utilizadas permite detectar y prever las limitaciones del dispositivo.

1.7.4. Características del robot modelo *Amigobot*

En el laboratorio que propone el proyecto se cuenta ya con un grupo de unos 8 robots que fueron adquiridos para pruebas y demostraciones de otros desarrollos. Se trata de pequeños robots de modelo *Amigobot* fabricados por la empresa *Mobile Robots* que incorporan ruedas para su desplazamiento y pueden ser dirigidos por control remoto desde un PC. Aunque no es imprescindible para la comprensión del proyecto, se introducen en esta sección algunas de las características del dispositivo.

Físicamente, el *Amigobot* es un autómata de unos 15 cm de altura con forma de prisma ovalado de 33x28cm con dos ruedas motrices y una tercera que pivota libremente (conocida como “caster”) en la parte trasera. Incluye como sensores un sistema de sónares (cuatro frontales, dos traseros y uno más en cada uno de sus laterales), codificadores rotatorios en las ruedas y un giroscopio (según el modelo) que no serán utilizados en este desarrollo. Un pequeño procesador integrado permite ejecutar pequeños programas creados en C/C++ y una antena “Wireless Serial Ethernet” habilita la operación inalámbrica a distancia del robot. Además de la conexión inalámbrica, el robot ofrece la opción de conectarse a un PC bien mediante un cable ethernet (RJ48) o puerto serie (RS-232). Otras de sus principales características no mencionadas son su peso (3.6 Kg), capacidad de carga (1 Kg) y velocidad de avance, retroceso y giro: que son de 1 metro por segundo en línea recta y 100 grados por segundo al girar.

Gracias a sus posibilidades de conexión, el robot puede ser controlado de forma inalámbrica desde un sistema Windows mediante el software *MobileEyes* proporcionado por el fabricante, pero también conectándolo a un PC con el entorno ROS instalado (usando Ubuntu Linux en tal caso) y mediante el paquete ROS “generic_teleop” creado por un miembro del laboratorio con la intención de poder controlar distintos robots utilizando un mismo paquete genérico.

En cuanto al presente desarrollo, el único dispositivo de entrada que se espera conectar al robot, y que es imprescindible para realizar las medidas de profundidad y la odometría visual, es la cámara de profundidad *Microsoft Kinect*, que es la que obtiene la información tridimensional del escenario y permite generar las nubes de puntos utilizadas para los cálculos. La cámara no requiere preparación especial, pero sí conexión USB a un PC, que será el mismo que gobierne el robot. Existe más información disponible sobre el robot y sus características en la página web del fabricante: “www.mobilerobots.com”.

1.7.5. Algoritmo Iterativo del Punto más Cercano (ICP)

El “algoritmo iterativo del punto más cercano” (en adelante *ICP* por sus siglas del inglés *Iterative Closest Point*) es una técnica iterativa para aproximar nubes de puntos entre sí hasta alinearlas (superponerlas en el espacio) tal como se ejemplifica en la figura. El resultado de los cálculos es una matriz de transformación que relaciona la posición y orientación de las dos nubes por lo que, indirectamente, se obtiene la transformación sufrida por el visor (es decir, la cámara, dado que su posición es fija en relación a cada nube de puntos porque se miden con respecto a ella).

Dado que el objetivo de la segunda parte del proyecto es calcular los movimientos (odometría) de un visor a partir de dos nubes de puntos capturadas, se hacen frecuentes alusiones a este algoritmo que se utiliza para los cálculos. Es importante aclarar que su implementación no es parte del alcance del desarrollo, sino que se utiliza una implementación externa. Por ello, y con especial interés para los manuales de usuario y desarrollador, se tratará de ofrecer un pequeño esbozo de en qué se basa ICP y qué parámetros permiten modificar su comportamiento.

Funcionamiento del algoritmo.- ICP trata de asociar puntos de la primera nube (patrón) a los más similares dentro de la segunda (objetivo) para luego calcular la distancia entre cada par y obtener una matriz de transformación tal que, aplicada sobre los puntos del patrón, minimice la suma de las distancias (elevadas al cuadrado). Este proceso se ilustra simplificado a 2 dimensiones en la figura 1.6:

1. **Asociación de puntos.** Se buscan pares de puntos correspondientes (aparentemente) a la misma zona de la escena en cada una de las nubes. Estos serán los utilizados para la estimación.
2. **Descartar puntos.** Mediante el algoritmo RANSAC se determinan y eliminan las correspondencias más atípicas (que se alejen de las tendencias generales). Así se tendrán en cuenta sólo los valores más fiables.
3. **Estimación de la transformación.** Se optimizan los parámetros de la matriz de transformación mediante la minimización de la función coste, calculada según el error cuadrático medio. El error es la distancia entre los puntos asociados.
4. **Aplicar la transformación.** El patrón se actualiza mediante la matriz de transformación para obtener los valores nuevos de cada punto de la nube.
5. **Iterar.** Hasta que se cumplan los criterios de parada, se repiten los pasos anteriores con el nuevo patrón acumulando los resultados de las transformaciones.

Parámetros de configuración del algoritmo.- Existen una serie de parámetros que permiten configurar el comportamiento del algoritmo, bien determinando hasta cuándo debe seguir iterando (criterios de parada), o el modo en que calcula el resultado.

- **Nº máximo de iteraciones** Al alcanzar este número de iteraciones, el algoritmo retorna el mejor resultado obtenido.



Figura 1.6: Representación bidimensional del acercamiento entre las nubes de puntos

- **Convergencia** Detiene el cálculo cuando la suma de las diferencias entre la estimación anterior y la nueva es inferior a un margen “epsilon” determinado llamado.
- **Diferencia con la solución** Finaliza cuando la distancia entre los puntos asociados es lo suficientemente baja. Calcula la suma de las distancias al cuadrado y comprueba que sea menor que el margen determinado.
- **Distancia máxima entre correspondencias** También denominado “distancia euclíadiana” en las referencias, es el desplazamiento máximo que se espera de un mismo punto entre las dos nubes. Si dos puntos “similares” están demasiado lejos en el espacio se descarta que sean el mismo.
- **Máximo de iteraciones de RANSAC** Indica la cantidad veces que itera el algoritmo contenido (RANSAC) que detecta los puntos atípicos.
- **Mínima distancia entre vecinos (RANSAC)** La distancia, en metros, dentro de la cual dos puntos son considerados vecinos.

1.8. Glosario

Conceptos generales de computación

C++ Lenguaje de programación creado por Bjarne Stroustrup para agregar orientación a objetos al ya existente lenguaje “C”. Se trata de un lenguaje con tipado fuerte, inseguro y nominativo, considerado de bajo nivel con respecto a otros lenguajes más modernos.

Controladores (o drivers) Ambos términos se usarán de forma indistinta para denominar el software de bajo nivel que controla los dispositivos físicos externos. Así las aplicaciones se comunican con los drivers y no, directamente, con cada dispositivo.

Callback Función a la que se llama en respuesta a un evento para que actúe en consecuencia. En el caso actual es llamada automáticamente al recibir un mensaje nuevo o cambios en las configuraciones. También se pueden denominar “retrollamadas” o, más descriptivamente: “manejadores de eventos”.

European Centre for Soft Computing Centro Europeo por el progreso del Soft Computing, sito en Mieres (Asturias). Está orientado a la investigación de soluciones a problemas con información incompleta mediante algoritmos computacionales e inteligencia artificial.

Marca de tiempo (timestamp) Campo de datos de algunas estructuras y de muchos mensajes del entorno ROS en general que sitúa dicha información en el tiempo con distintos propósitos. Se trata generalmente de un valor en segundos contabilizados desde un instante de partida común en función del convenio utilizado.

Microsoft Corporation Empresa de origen estadounidense dedicada al sector de la informática y fundada en 1975 por Bill Gates y Paul Allen. Es la empresa que comercializa la cámara de profundidad utilizada en el proyecto (“Microsoft Kinect”) aunque se ideó inicialmente como un dispositivo de juego.

Microsoft Kinect Periférico de consola comercializado por *Microsoft* pero compatible con PC que incorpora una cámara a color, una de profundidad y varios micrófonos para la interacción con los jugadores. Durante el presente se utilizará como periférico de adquisición de mapas de profundidad (cámara de profundidad) de los que se obtienen las nubes de puntos en 3D usadas en el desarrollo.

Mobile Eyes Software para control remoto y monitorización del robot *Amigobot* usado en el desarrollo. Consiste en una interfaz gráfica de usuario diseñada y distribuida por la empresa creadora del robot: “Mobile Robots”.

Periférico Dispositivo físico capaz de acoplarse externamente a un sistema para extender su funcionalidad (con nuevas entradas o salidas de datos o modos de interactuar con el entorno, por ejemplo).

Python Lenguaje de programación interpretado, de alto nivel y multiparadigma. Utiliza tipado fuerte y dinámico y es mantenido por la Fundación de Software Python (*Python Software Foundation*) desde su publicación en 1991.

Retrocompatibilidad Es la característica del software que, de ser poseída, determina que el software es compatible con los recursos de versiones anteriores. Los recursos pueden ser ficheros (por ejemplo documentos de 1998 que son legibles con aplicaciones de 2010) o librerías y código fuente (Un compilador que comprende código fuente antiguo o librerías que incorporan funciones anticuadas con el objetivo de mantener esta característica).

Robot Entidad mecánica con una autonomía física determinada en sus movimientos y acciones de manipulación del entorno. Aunque puede haber otras interpretaciones, en este caso se trata de elementos capaces de ser programados para realizar ciertas tareas físicas.

Robot Amigobot Modelo de robot de que se dispone para el proyecto. Es un dispositivo de pequeño tamaño, sin extremidades y comercializado por la empresa “Mobile Robots”. Se amplía ligeramente su descripción en el apartado “Características del robot modelo *Amigobot*” de los anexos (1.7.4).

RJ45 Interfaz física de conexión con un conector de 8 pines y que es estándar para las conexiones “Ethernet” entre equipos. Es la conexión utilizada para conectar el PC al robot en el presente proyecto.

Wiki Neologismo que designa una suerte de sitio web de consulta en cuya edición pueden participar múltiples usuarios de forma colaborativa. En este caso se utiliza para la documentación del sistema ROS (entre otros desarrollos) incluyendo tutoriales, referencias y listados de recursos.

XML Lenguaje de marcado desarrollado por el W3C (World Wide Web Consortium) para favorecer el intercambio estandarizado de información estructurada entre diferentes plataformas.

YAML Lenguaje de marcado ligero inspirado en XML entre otros pero orientado a ser utilizado como formato estándar para la serialización de datos. Se basa en la idea de que todos los datos pueden ser representados como combinaciones de listas, estructuras y datos básicos y permite representar éstos de un modo estructurado y amigable para la lectura.

Conceptos de visión por computador

ICP (Iterative Closest Point) Algoritmo iterativo del punto más cercano, ideado para estimar la posición relativa entre dos colecciones de puntos de dimensionalidad variable. Se utiliza en el desarrollo para averiguar el movimiento que relaciona dos nubes de puntos capturadas (o sus escenas asociadas) y se explica en los anexos de esta documentación bajo el nombre: “Algoritmo Iterativo del Punto más Cercano” (apartado 1.7.5 de la documentación).

Cámara de profundidad (depth camera) Dispositivo de adquisición de datos que produce información relativa a la distancia entre la lente y diferentes puntos del entorno. Se pueden representar dichos valores como un “mapa de profundidades” o “imagen de disparidad” o se pueden convertir a nubes de puntos.

Cámara RGB (RGB camera) Dispositivo de adquisición de datos que produce información relativa al color que muestran los distintos puntos del entorno. Da lugar a una “imagen” en el sentido común de la palabra e intuitivamente perceptible por las personas.

Comportamiento planificado Modelo de comportamiento que designa, en robótica, la programación de un autómata que requiere toda la información disponible sobre el entorno para buscar soluciones completas al problema que se le proponen.

Comportamiento reactivo Modelo de comportamiento que designa, en robótica, la programación de un autómata cuando éste reacciona de forma directa e inmediata a estímulos, es decir, reaccionando exclusivamente a la información puntual disponible en el momento desencadenando uno o varios comportamientos en respuesta.

Imagen de profundidad (depth image) Estructura de datos que almacena información de una escena sobre la distancia entre el dispositivo de adquisición y cada punto visible de la misma. También llamado “mapa de profundidades”, se representa como una sucesión ordenada de valores que se asocian a cada sección en que se divide la escena desde el punto de vista de la cámara.

Imagen RGB (RGB image) Estructura de datos que almacena información de una escena sobre el color asociado a cada sección visible desde el punto de vista de la cámara. Se representa como una sucesión ordenada de puntos con tres valores cada uno para definir la mezcla de colores rojo, verde y azul.

Nube de puntos Estructura de datos que conforma una colección de puntos que puede estar ordenada según diferentes criterios o desordenada. Los puntos pueden tener diferentes dimensionalidades y campos como: 'x', 'y', 'z' y los colores 'r', 'g', 'b'. En el desarrollo actual se utilizan nubes de puntos no ordenadas (no explícitamente, aunque pueden estarlo internamente para optimizar su manejo) y sin información sobre color.

Odometría Estudio de los movimientos realizados por un elemento mecánico para estimar su posición durante la navegación. El concepto puede designar la estimación obtenida en sí misma.

Odometría Visual Estudio de los movimientos de un elemento mecánico para estimar su posición haciendo uso de cámaras como dispositivos de adquisición e imágenes como datos de partida.

Point Cloud Library Librería especializada en el tratamiento de nubes de puntos y la obtención de información a partir de ellas. El proyecto, también conocido por sus siglas *PCL*, es llevado a cabo por un grupo de marcas individuales que colaboran y entre los que se pueden encontrar universidades, fabricantes de procesadores y nombres reconocidos en el sector de la automoción.⁶

Visión por computador Disciplina de la computación impulsada por la pretensión de procesar imágenes y extraer información de forma no asistida (sin ayuda humana) mediante un equipo informático e inteligencia artificial.

Conceptos específicos del sistema ROS

Robot Operating System (ROS) Se define como un meta-sistema operativo para aplicaciones robóticas. Hace las veces de framework de desarrollo, entorno de ejecución y *suite* de herramientas. Se amplía su definición en el primer anexo de la documentación bajo el título “Robot Operating System (ROS)”.

Nodo ROS Cada uno de los procesos individuales ejecutados dentro del entorno ROS y que conforman su estructura modular. Cada nodo tiene responsabilidad limitada y se relaciona con otros mediante los canales ofrecidos por ROS (topics, servicios y servidor de parámetros) para colaborar en tareas conjuntas.

⁶Información extraída de su página web: “<http://pointclouds.org/about>”

Paquete ROS (ROS package) Conjunto de elementos del entorno ROS (ficheros de lanzamiento y de configuración, ejecutables, etc). Representa una aplicación o software individual del sistema aunque puede englobar varios nodos o procesos. Cada paquete permite extender el entorno con una funcionalidad o grupo de funcionalidades.

Parámetros Dinámicos (Dynamic Parameters) Subconjunto de los parámetros del *servidor de parámetros de ROS* que son actualizados y monitorizados en tiempo real por un módulo llamado “dynamic_reconfigure”. Éste se encarga de informar a los nodos interesados de cambios en determinados parámetros, de forma que se puedan desencadenar efectos en tiempo real.

Pilas ROS (ROS stack) Conjunto de paquetes ROS relacionados que son agrupados por motivos organizativos y de mantenimiento.

Servicios ROS Canal de comunicación ofrecido de forma nativa por el entorno ROS y con un funcionamiento similar a llamadas a procedimientos remotos (RPC) de otros sistemas. Es decir: cada proceso ofrece sus servicios en forma de funciones públicas de modo que los otros procesos puedan realizar llamadas a ellos y obtener una respuesta.

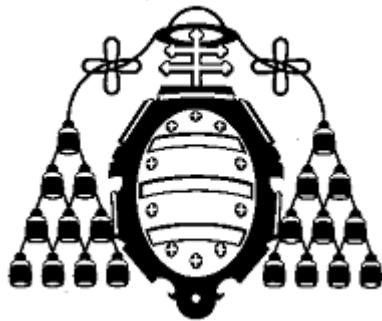
Servidor de Parámetros de ROS (ROS Parameter Server) Espacio de memoria compartida habilitado nativamente dentro del entorno ROS que permite almacenar información de forma pública y accesible por todos los nodos para lectura y escritura. El entorno ofrece los medios para interactuar con el servidor de parámetros en el que cada nodo tiene su “espacio privado” para uso propio pero no “restringido” en caso de que otro intente modificar sus valores.

Topic de ROS Canal de comunicación orientado al intercambio de flujos continuos de mensajes asíncronos. Cada topic se distingue por una “dirección” dentro del entorno y un tipo de mensaje (o dato) asociado y se mantiene abierto mientras haya nodos a la escucha o pendientes de publicar. Cualquier nodo puede transmitir o recibir en cualquier dirección siempre y cuando utilice el tipo de mensaje apropiado.

1.9. Referencias electrónicas

A continuación se presenta una relación de las principales referencias electrónicas consultadas durante el desarrollo del proyecto. Se trata de fuentes consideradas fiables y consultadas de forma frecuente en busca de documentación, especificaciones técnicas, etc.

- Página web del proyecto *ROS* que incluye la documentación de referencia, relación de paquetes y herramientas disponibles y un foro para posibles dudas y problemas: “www.ros.org”.
- Documentación de referencia y sobre modo de uso en la página web de la librería *PCL*:
“<http://pointclouds.org/documentation>” y
“<http://docs.pointclouds.org/trunk>”.
- Foros de consulta para usuarios y desarrolladores de *PCL*:
“wwwpcl-users.org” y “<http://dev.pointclouds.org>”.
- Web del proyecto *openni* que desarrolla los controladores para *Kinect*:
“<http://openni.org>”.
- Blog de control y seguimiento del proyecto con los resultados preliminares de cada avance:
“<http://cogiesmieres2012.blogspot.com.es>”.
- Documentación de referencia de la librería *pyYAML*: “<http://pyyaml.org>”.
- Documentación de referencia del lenguaje *Python*: “www.python.org”.
- Documentación de referencia del lenguaje *C++*: “www.cplusplus.com”.
- Especificaciones técnicas oficiales de *Microsoft Kinect* en su web:
“<http://kinectforwindows.org>”.
- Página web del fabricante del robot *Amigobot*: “www.mobilerobots.com”.
- Diversos artículos de “<http://en.wikipedia.org>” y sus fuentes referenciadas.



UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE
GIJÓN

**ÁREA DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL**

PROYECTO FIN DE CARRERA N° 3122067

**INTEGRACIÓN DEL DISPOSITIVO 'MICROSOFT KINECT'
COMO PARTE DE UNA ARQUITECTURA ROBÓTICA
GENÉRICA PARA SU USO COMO DISPOSITIVO DE ENTRADA**

DOCUMENTO N° II

PLANIFICACIÓN Y PRESUPUESTO



MIGUEL A. GARCÍA GONZÁLEZ

NOVIEMBRE – 2012

TUTOR: LUCIANO SÁNCHEZ RAMOS

CAPÍTULO 2

PLANIFICACIÓN Y PRESUPUESTO

2.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

2.1.1. Identificación del proyecto

Título:	“Integración del dispositivo ‘Microsoft Kinect’ como parte de una arquitectura robótica genérica”
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

2.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo “Microsoft Kinect” comercializado por la empresa “Microsoft Corporation”. La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

2.1.3. Visión general del documento

A continuación se ofrece el informe general detallado con la planificación temporal y el presupuesto económico preliminares a los que se somete el presente proyecto aunque sujetos a posibles cambios, adaptaciones y mejoras que se realicen durante el transcurso del mismo.

Deberán aparecer clasificadas las distintas fases del proyecto especificando su naturaleza, duración y objetivos (hitos), así como los distintos elementos y recursos a presupuestar y el desglose de los gastos que se espera asumir.

2.2. Plan para las fases

El desarrollo del presente software se realizará usando un enfoque en varias fases en el que varias iteraciones pueden ocurrir dentro de cada fase. De igual modo cada iteración puede sufrir varias actualizaciones. En la tabla siguiente (2.1) se describen todas las frases y el hito que marca la finalización de cada una:

fase	Descripción	Hito
Aprendizaje (entorno ROS)	El aprendizaje y asimilación de las numerosas tecnologías y conceptos necesarios para el desarrollo forman una fase preliminar tener en cuenta solamente a efectos de planificación.	La fase finaliza tras adquirir los conocimientos imprescindibles para comenzar.
Concepción	Se deciden la naturaleza y alcance del proyecto. Hay que describir las principales funcionalidades a desarrollar y realizar la planificación preliminar del proyecto. Tras la fase de concepción debe decidirse si el proyecto es técnicamente viable.	Conjunto preliminar de requisitos. Al final se decide: continuar o no continuar.
Elaboración (interfaz)	Se analizan los requisitos, se diseñan los casos de uso necesarios y se comienza a implementar una versión preliminar del sistema. El objetivo de esta implementación será comprobar la viabilidad técnica del desarrollo.	La elaboración de un prototipo de la interfaz marca el final de la fase y posibilita verificar las principales características propuestas.
Construcción (interfaz)	Las características y funcionalidades serán refinadas según las condiciones. El prototipo evoluciona hacia una versión preliminar, sobre la que realizar una nueva batería de pruebas.	El test de funcionalidad de la versión preliminar (beta) marca el final de la fase de construcción.
Aprendizaje (odometría)	Serán necesarios conocimientos adicionales sobre odometría, nubes de puntos y herramientas relacionadas para completar la siguiente fase.	El hito lo marca la obtención de los conocimientos necesarios para continuar.
Elaboración (odometría)	Esta etapa es análoga a su homóloga de la interfaz, se trata de alcanzar una versión preliminar (conocida como <i>beta</i>) que confirme la viabilidad.	La elaboración del prototipo marca, de nuevo, el final.
Construcción (odometría)	Se refinará el diseño y se completará el ejecutable hasta alcanzar una fase beta, del mismo modo que se hizo con la interfaz.	El final de la fase se alcanza tras el test de funcionalidad correspondiente.
Integración y Transición	Se tratará de obtener una versión final del conjunto, un producto acabado. Incluirá el funcionamiento conjunto de ambas partes, los manuales correspondientes y la memoria para futuras referencias.	La obtención del material de soporte y una versión del producto que cumpla todos los requisitos indicará el fin de esta fase y del desarrollo.

Tabla 2.1: Fases e hitos contemplados en la planificación del proyecto

Se presentan también las duraciones correspondientes a cada fase de las ya listadas y qué semanas ocupará cada una. Cabe destacar que la última fase no incluye la confección de la memoria y los manuales por lo que la suma total es de 28 semanas (7 meses) en total. Todo ello está recogido en la tabla 2.2.

Fase	Nº Iteraciones	Comienzo	Fin
Aprendizaje	1 - 4 semanas	Semana 1	Semana 6
Concepción	1 - 2 semanas	Semana 2	Semana 3
Elaboración (interfaz)	1 - 3 semanas	Semana 4	Semana 7
Construcción (interfaz)	1 - 5 semanas	Semana 8	Semana 12
Elaboración (odometría)	1 - 2 semanas	Semana 13	Semana 14
Construcción (odometría)	1 - 6 semanas	Semana 15	Semana 20
Transición	1 - 8 semanas	Semana 21	Semana 28

Tabla 2.2: Duraciones de las distintas fases del desarrollo

2.3. Documentación resultante de las fases

Al término de cada fase se espera ir obteniendo y completando diferentes documentos que reflejan el trabajo realizado y, aún más importante, las directrices a seguir durante la realización de fases posteriores. El cuadro (2.3) contiene el listado de la documentación correspondiente asociada a dichas fases.

	Concepción	Elaboración	Construcción	Transición
Análisis y requisitos	Visión del proyecto - Ámbito y alcance - Análisis de alternativas	Catálogo de requisitos - Espec. de Casos de Uso. - Espec. Suplementaria	Espec. de Casos de Uso (final) - Espec. Suplementaria	
Diseño		Realización de Casos de Uso. Diseño de la arquitectura del sistema	Realización de Casos de Uso. Documento “Arquitectura del sistema”	
Implementación			Generación de versiones beta ¹	Generación de versiones beta]
Test			Plan de pruebas - Resultado del plan de pruebas	Plan de pruebas - Resultado del plan de pruebas
Distribución				Manual de usuario - Memoria - Notas de la versión.
Organización	Planificación del proyecto			

Tabla 2.3: Documentación a obtener tras las distintas fases del desarrollo

¹Se denomina beta a una versión preliminar del producto que aún no ha pasado las pruebas de aceptación.

2.4. Objetivos de cada iteración

Cada fase consiste en una serie de iteraciones de desarrollo en las cuales se diseña una parte del sistema. En general, estas iteraciones:

- Proporcionan versiones preliminares del sistema que sirvan como base a las siguientes.
- Flexibilizan la planificación de características para cada versión preliminar al proporcionar un punto de partida para las estimaciones.
- Permiten realizar cambios en los objetivos para mejorar el resultado final a la vista de los resultados preliminares

La tabla que se muestra a continuación, 2.4 describe las iteraciones junto con sus hitos asociados sin incluir los documentos que ya se destacaron en el título anterior:

Fase	Iteración	Descripción	Hito
Aprendizaje	Preliminar	Adquisición de conocimientos mínimos necesarios para continuar.	-
Concepción	Preliminar	Definir los requisitos del producto y la planificación del proyecto, así como su viabilidad técnica.	Revisión de requisitos del sistema
Elaboración (interfaz)	Desarrollar un prototipo	Ánalisis y diseño para los casos de uso más importantes propuestos para el prototipo. Desarrollar un prototipo de la arquitectura para la primera versión.	Prototipo de la interfaz
Construcción (interfaz)	C.1.0. – Desarrollar la primera versión beta	Implementar y testear los casos de uso incorporados a la primera versión para poder lanzar la primera versión beta.	versión Beta 1
	C.2.0. – Desarrollar la versión RC ²	Implementar y testear los casos de uso restantes. Solucionar defectos de la primera versión beta. Completar la versión RC	versión RC
Elaboración (odometría)	Desarrollar un prototipo	Ánalisis y diseño para los casos de uso más importantes propuestos para el prototipo. Desarrollar un prototipo de la arquitectura para la primera versión.	Prototipo de la odometría
Construcción (odometría)	C.1.0. – Desarrollar la primera versión beta	Implementar y testear los casos de uso incorporados a la primera versión para poder lanzar la primera versión beta.	versión Beta 1
	C.2.0. – Desarrollar la versión RC ²	Implementar y testear los casos de uso restantes. Solucionar defectos de la primera versión beta. Completar la versión RC	versión RC
Transición	T.1.0 - Desarrollo de la Release 1.0	Unir ambas partes del desarrollo. Distribuir e instalar la Release 1.0 (primera versión para publicar)	Release 1.0

Tabla 2.4: Objetivos de las iteraciones del desarrollo

²Siglas de “Release Candidate”, voz inglesa para “Candidata para publicar”.

2.5. Planificación de los costes

En la siguiente sección se realiza un estudio de los costes que, se espera, conlleve el proyecto a modo de planificación previa para llegar a confeccionar, como resultado, un presupuesto lo más coherente posible, de los costes del proyecto.

2.5.1. Estimación de recursos necesarios

Se realiza primeramente un análisis de los recursos necesarios para llevar a cabo el proyecto, teniendo en cuenta los elementos físicos y productos software que serán utilizados durante el mismo, así como las personas y horas de trabajo que serán necesarias para completar los objetivos.

Recursos hardware

ID UNIDAD	DESCRIPCIÓN	U. DE MEDIDA	Nº DE UD.
HW01	Ordenador portátil tipo PC	Unidades	1
HW02	Robot Amigobot	Unidades	1
HW03	Cámara MS Kinect	Unidades	1

Tabla 2.5: Relación de recursos Hardware necesarios

Recursos software

ID UNIDAD	DESCRIPCIÓN	U. DE MEDIDA	Nº DE UD.
SW01	Ubuntu GNU/LINUX	Unidades	1
SW02	Robot Operating System (ROS)	Unidades	1
SW03	Eclipse IDE	Unidades	1
SW04	Kile LaTeX environment	Unidades	1

Tabla 2.6: Relación de recursos Software necesarios

Recursos humanos

Las labores de dirección han sido llevadas a cabo por **Enrique H. Ruspini**, director del departamento de *Sistemas Inteligentes Colaborativos* en el Centro Europeo de Soft Computing (ECSC) de Mieres y **Luciano Sánchez Ramos**, profesor catedrático en el departamento de informática de la universidad de Oviedo. Se estima su dedicación al proyecto en dos horas mensuales de puesta en común con el supervisor y el desarrollador.

La supervisión del proyecto corre a cargo de **Kevin LeBlanc**, investigador miembro departamento de *Sistemas Inteligentes Colaborativos* en el Centro Europeo de Soft Computing (ECSC) de Mieres, cuya dedicación, en horas semanales es de unas dos para reuniones sobre los avances del desarrollo.

El desarrollo del presente proyecto se ha dividido en las siguientes fases:

- Estudio de la documentación correspondiente a las tecnologías y conceptos necesarios para el desarrollo del proyecto.
 - Estudio del entorno de base ROS.
 - Familiarización con el dispositivo Kinect y el robot Amigobot a utilizar.
 - Lenguaje de programación Python (se partía de C++ ya conocido).
 - Conocimientos sobre el funcionamiento de la librería “Point Cloud Library” (PCL).
- Planificación del desarrollo a realizar y especificación técnica de las decisiones tomadas.
- Implementación del software según la especificación obtenida. Incluyendo las correcciones necesarias sobre el mismo.
- Integración de las dos partes planificadas para obtener un producto final.
- Entrega del código y publicación de la versión final en el sistema de control de versiones utilizado.

Para el peritaje de los costes de los recursos humanos se ha elaborado la siguiente estimación:

ID UNIDAD	DESCRIPCIÓN	U. DE MEDIDA	Nº DE UD.
HU01	Análisis del catálogo de requisitos y planificación	Horas	80
HU02	Desarrollo	Horas	610
HU03	Supervisión	Horas	70
HU04	Dirección	Horas	21

Tabla 2.7: Relación de recursos Humanos necesarios

2.6. Presupuesto

2.6.1. Mediciones

Los inmovilizados inmateriales (Ubuntu GNU/Linux 12.04, Eclipse IDE 4.2 Juno y Robot Operating System) son todos ellos productos software de uso libre con licencias GNU GPL, Eclipse Public License y BSD License respectivamente, por lo que no conllevan gasto económico. Los inmovilizados materiales, por su parte (ordenador, cámara de profundidad y robot) se deberán amortizar a lo largo de futuros proyectos durante la vida del laboratorio. Tratándose de un proyecto de 7 meses de duración, a los presupuestos se les imputará la amortización correspondiente de los inmovilizados utilizados durante ese tiempo.

2.6.2. Cuadros de precios

Recursos hardware

ID UNIDAD	DESCRIPCIÓN	U. DE MEDIDA	PRECIO/U.
HW01	Ordenador portátil tipo PC	Unidades	365
HW02	Robot Amigobot	Unidades	1750
HW03	Cámara MS Kinect	Unidades	90

Tabla 2.8: Cuadro de costes de los recursos hardware

Recursos software

ID UNIDAD	DESCRIPCIÓN	U. DE MEDIDA	PRECIO/U.
SW01	Ubuntu GNU/LINUX	Unidades	0
SW02	Robot Operating System (ROS) (and packages)	Unidades	0
SW03	Eclipse IDE	Unidades	0
SW04	Kile LaTeX environment	Unidades	0

Tabla 2.9: Cuadro de costes de los recursos software

Recursos humanos

ID UNIDAD	DESCRIPCIÓN	U. DE MEDIDA	PRECIO/U.
HU01	Análisis del catálogo de requisitos y planificación	€ /hora	30
HU02	Desarrollo	€ /hora	25
HU03	Supervisión	€ /hora	50
HU04	Dirección	€ /hora	55

Tabla 2.10: Cuadro de costes de los recursos humanos

2.6.3. Presupuestos parciales

Recursos hardware

ID UNIDAD	DESCRIPCIÓN	IMPORTE
HW01	Ordenador portátil tipo PC ³	61
HW02	Robot Amigobot	1750
HW03	Cámara MS Kinect	90
	TOTAL PRESUPUESTO HARDWARE	1.901

Tabla 2.11: Cuadro de costes de los recursos hardware

Recursos software

ID UNIDAD	DESCRIPCIÓN	IMPORTE
SW01	Ubuntu GNU/LINUX	0
SW02	Robot Operating System (ROS) (and packages)	0
SW03	Eclipse IDE	0
SW04	Kile LaTeX environment	0
	TOTAL PRESUPUESTO SOFTWARE	0

Tabla 2.12: Cuadro de costes de los recursos software

³Plazo de amortización de 3 años, de los cuales 7 meses corresponden a este proyecto

Recursos humanos

ID UNIDAD	DESCRIPCIÓN	IMPORTE
HU01	Análisis del catálogo de requisitos y planificación	2.400
HU02	Desarrollo	15.250
HU03	Supervisión	3.500
HU04	Dirección	1.155
	TOTAL PRESUPUESTO RRHH	22.305

Tabla 2.13: Cuadro de costes de los recursos humanos

2.6.4. Presupuesto final

IMPORTE TOTAL (€)	IMPORTE
Recursos Hardware	1.901
Recursos Software	0
Recursos Humanos	22.305
TOTAL	24.206

Tabla 2.14: Presupuesto total

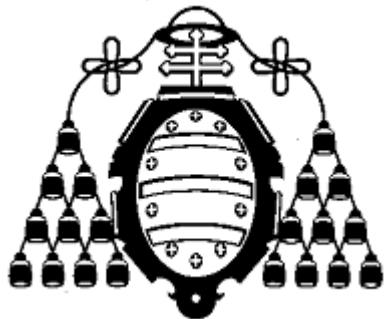
CONCEPTO	IMPORTE
Presupuesto de ejecución de material	24.206,00
Beneficio Industrial (6 %)	1.452,36
Costes Generales (15 %)	3.630,90
Suma de gastos y beneficios	29.289,26
I.V.A. (21 %)	6.150,74
PRESUPUESTO DE EJECUCIÓN POR CONTRATA	35.440,00

Tabla 2.15: Presupuesto final

Asciende el presupuesto de ejecución por contrata a la expresada cantidad de **treinta y cinco mil cuatrocientos cuarenta euros**.

Gijón, a 5 de noviembre de 2012

Firmado: **Miguel A. García González (53.553.235-N)**



UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE
GIJÓN

**ÁREA DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL**

PROYECTO FIN DE CARRERA Nº 3122067

**INTEGRACIÓN DEL DISPOSITIVO 'MICROSOFT KINECT'
COMO PARTE DE UNA ARQUITECTURA ROBÓTICA
GENÉRICA PARA SU USO COMO DISPOSITIVO DE ENTRADA**

DOCUMENTO N° III

ESPECIFICACIÓN DEL SISTEMA



MIGUEL A. GARCÍA GONZÁLEZ

NOVIEMBRE – 2012

TUTOR: LUCIANO SÁNCHEZ RAMOS

CAPÍTULO 3

ESPECIFICACIÓN DEL SISTEMA

3.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

3.1.1. Identificación del proyecto

Título:	“Integración del dispositivo ‘Microsoft Kinect’ como parte de una arquitectura robótica genérica”
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

3.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo “Microsoft Kinect” comercializado por la empresa “Microsoft Corporation”. La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

3.1.3. Visión general del documento

El documento de *especificación* tiene como objeto responder a la pregunta de qué debe hacer el sistema y describir su funcionalidades de forma clara, unívoca y completa desde el punto de vista de las necesidades del usuario. No deben formar parte de este, por tanto, los detalles técnicos de cómo se alcanzarán los objetivos, que se pueden encontrar en el documento siguiente (Parte 4) sobre el *Diseño*.

En aras de cumplir las normas de presentación, que aconsejan el uso de una metodología de desarrollo reconocida, se utilizarán actividades correspondientes a dos procesos de la metodología “Métrica V3”: el “Estudio de Viabilidad del Sistema” y el “Análisis del Sistema de Información”. Al final del documento se habrá desarrollado la ingeniería previa que permitirá crear un sistema sólido, coherente y estructurado una vez se tienen las distintas partes y una idea clara del producto que se desea alcanzar.

3.2. Ámbito y alcance del sistema

El proyecto a desarrollar surge en respuesta a unas necesidades determinadas y dentro de un contexto que es conveniente analizar en aras de buscar la mejor aproximación posible a la solución. Se definen, a continuación, el ámbito en el que se enmarca dicho proyecto, el alcance en cuanto a sus responsabilidades y las funciones que deberá realizar. Con esto se pretende tener una idea clara y completa que sirva como paso previo a la especificación de la solución.

3.2.1. Ámbito del proyecto

La robótica es una rama de la tecnología que nace con el propósito de crear dispositivos mecanismos autónomos y capaces, entre otras cosas, de realizar trabajo físico por sí mismos para liberar de éste a los seres humanos. Dicho campo lleva ya tiempo captando el interés de inventores, investigadores y profesionales; y se encuentra en pleno desarrollo por su evolución cada vez más rápida y su potencial para mejorar la vida de las personas. Sin embargo, existe hasta el momento una cierta tendencia a desarrollar soluciones excesivamente específicas (*ad-hoc*) que dificultan en parte la reutilización de recursos (tanto código como elementos físicos) y dificultan, así, el crecimiento de nuevos proyectos ralentizando la aparición de resultados.

A la vista de lo anterior, el grupo de investigación *Willow Garage*, junto con el laboratorio de inteligencia artificial de la universidad de Stanford crearon un entorno genérico para aplicaciones robóticas llamado “*Robot Operating System*” (en adelante *ROS*) que con una estructura de nodos individuales pero complementarios fuera capaz de implementar, mediante paquetes de uno o varios procesos, las funciones que cada usuario fuera necesitando para su robot. Así, los usuarios disponen de un entorno común genérico con gran cantidad de funciones y controladores ya implementados que ha ido evolucionando y mejorando rápidamente desde su creación en 2007 con personas y proyectos que se han ido uniendo para contribuir. Para más información sobre *ROS* se puede consultar el apartado 1.7.1 sobre “*Robot Operating System*” al final del documento *Memoria*.

Gracias a la existencia de un entorno genérico para robots se hace posible la creación de herramientas reutilizables que sirvan de apoyo en diferentes proyectos cada vez más complejos, como los que se llevan a cabo en el departamento de “*Sistemas Inteligentes Colaborativos*” del Centro Europeo por el progreso del Soft Computing (ECSC por su siglas en inglés), situado en Mieres (Asturias). La unidad se encarga de diversos proyectos relacionados con la interacción entre distintos robots y también de estos con humanos, y por ello han plasmado dos de sus necesidades actuales en el desarrollo actual: la creación de una interfaz genérica para comunicar más fácilmente distintas aplicaciones con diferentes dispositivos de entrada y robots (ya que si bien el entorno es genérico, los drivers no tienen asignados nombres iguales para recursos homólogos), y la incorporación del dispositivo Microsoft Kinect en particular como fuente para estimar la odometría (desplazamiento) utilizando los robots del modelo “*Amigobot*” de los que se dispone y que se describirán más ampliamente en otras secciones del documento.

3.2.2. Descripción general del proyecto

El proyecto propuesto consiste, concretamente, en el desarrollo de dos elementos software en forma de paquetes para el entorno ROS que sean capaces de ejecutarse sobre el susodicho entorno (incluso utilizados por separado) y cumplir dos cometidos:

- El primero de ellos, comunicándose directamente con los controladores de dispositivos, facilitará el acceso a sus recursos de forma genérica con nombres comunes incluso si los controladores pertenecen a periféricos distintos siempre y cuando los recursos sean comunes y estén definidos (por ejemplo: “PointCloud” puede ser una nube de puntos proveniente de una cámara de profundidad “Microsoft Kinect” o del dispositivo homólogo fabricado por la compañía Asus : “XTionPRO”), de tal forma que al cambiar el dispositivo sólo sea necesario especificar el nuevo fichero con los “nombres” de los recursos. Para ello se diseñará una “interfaz” para los drivers que conozca los distintos canales de comunicación de ROS y haga de intermediario entre las aplicaciones y los controladores siendo configurable por el usuario mediante un fichero externo.
- El segundo paquete ROS deberá ser una aplicación que, comunicándose bien con la interfaz anterior o directamente con los drivers, reciba capturas del entorno en forma de nubes de puntos con información del escenario en tres dimensiones y realice, a partir de cada par consecutivo que reciba, una estimación de la odometría del robot (el movimiento realizado) para calcular la posición y la orientación nuevas resultantes.

3.2.3. Objetivos y alcance

Se especifican ahora los objetivos que se extraen de las necesidades presentadas y de sucesivas entrevistas con un miembro del departamento. Estos determinarán el alcance del proyecto, que se ampliará de forma más concreta durante el análisis de requisitos, y se considera fuera del alcance todo lo que no esté explícitamente especificado en los últimos. Se enumerarán por separado los aspectos a cumplir para cada uno de los dos elementos software que se propone: la interfaz y el cálculo de la odometría.

El cometido de la interfaz a diseñar es actuar de intermediaria entre ciertas aplicaciones (capa superior en términos de abstracción) y diferentes controladores de dispositivos (capa inferior), y que el usuario pueda utilizar dicha interfaz sin necesidad de tener conocimiento alguno de programación para lograr que una aplicación se comunique con un dispositivo que no era conocido para ésta, siempre y cuando los recursos que requiere sean de la misma naturaleza (y tipo de datos). Planteado como un listado de objetivos se pretende:

- La interfaz será capaz de modificar los parámetros de los controladores a petición de otras aplicaciones, así como informar a estas de cambios en los primeros.
- Los parámetros y recursos podrán tener distintos “identificadores” (nombres por parte de las aplicaciones y de los drivers) que la interfaz será capaz de relacionar sirviéndose de un fichero de configuración preparado a tal efecto.
- Se hará posible manipular la configuración de la interfaz de forma simple y sin conocimientos de programación previos.

- El software deberá requerir la mínima interacción posible por parte del usuario una vez iniciada, siendo lo bastante autónoma como para no distraer su atención de forma innecesaria.
- Mediante el uso de ficheros de lanzamiento de ROS (y ficheros de configuración adicionales) se podrá configurar lo necesario para que la interfaz se adapte a cada par aplicación-driver de forma que estos se comuniquen.
- Se asegurará el correcto comportamiento del software en las condiciones esperadas y la adecuada notificación de errores cuando exista algún problema con el entorno.

En el caso de la odometría, la simplicidad es muy importante de cara a desarrollos posteriores por lo que se exige, únicamente, la posibilidad de calcular la transformación (el movimiento y rotación) entre cada dos nubes de puntos consecutivas, de forma que ese mismo comportamiento sea ampliable a un flujo de nubes de puntos y dejando como posible ampliación la implementación de información adicional en la respuesta del algoritmo como una evaluación de la fiabilidad de la medida y del estado del algoritmo para determinar si hubo algún error. La configurabilidad para adaptarlo a distintos casos es también imprescindible. Esto se traduce en las siguientes directrices:

- A partir de dos capturas (nubes de puntos) consecutivas de la cámara, debe ser posible calcular una estimación del movimiento realizado entre ellas (si éste se mantiene dentro de un margen determinado).
- Las estimaciones de odometría resultantes deben ser retransmitidas de forma tal que otra aplicación pueda recibirlas y utilizarlas.
- La estimación del movimiento se realiza a partir de la información de profundidad obtenida en forma de nubes de puntos, pero sin tener en cuenta la información adicional sobre el color debido a la posibilidad de utilizar otros dispositivos de entrada que no aporten tal información).
- La configuración del algoritmo deberá ser posible mediante el propio entorno *ROS* para máxima compatibilidad con él y simplicidad.
- La mayor cantidad de parámetros que sea posible deben ser configurables para adaptar el funcionamiento del algoritmo a los distintos escenarios y optimizarlo según los intereses del usuario.
- Se debe asegurar el correcto comportamiento del software en las condiciones esperadas y la adecuada notificación de errores cuando exista algún problema con el entorno.

Adicionalmente al comportamiento del software en tiempo de ejecución y de cara al usuario final, es imprescindible que el desarrollo esté especialmente orientado a facilitar un código fuente modular y comprensible destinado a su reutilización, pues se planea utilizarlo como base para futuros desarrollos.

3.3. Lista de usuarios participantes

En los puntos posteriores se analiza cuál es el perfil esperado de los usuarios finales de la herramienta que se ofrece, que se consideran todos aquellos que interactúen con la aplicación en cualquier momento de su vida útil. Siendo una aplicación diseñada para su uso en un sistema robótico más amplio y que no distingue (internamente) entre distintos tipos de usuarios, se puede considerar que todos los usuarios serán iguales de cara a la aplicación, es decir, “usuarios finales” sin más. Por ese motivo, se estudiará simplemente qué clase de profesionales podrán ser dichos usuarios.

3.3.1. Mercado del producto

Este software nace de dos necesidades distintas pero complementarias que son la conexión y control de diferentes dispositivos a distintos robots, y el uso del dispositivo “MS Kinect” en particular como dispositivo de entrada de datos para la estimación de la odometría en los robots de modelo “Amigobot” en particular. Dichas funcionalidades están orientadas especialmente a facilitar el trabajo de los miembros del laboratorio de “Sistemas Inteligentes Colaborativos” del “Centro Europeo de Soft Computing” (ECSC) de Mieres: la interfaz les permitirá utilizar con mayor facilidad diferentes modelos de hardware sin tener que rediseñar cada aplicación, y la estimación de odometría es necesaria para que los robots sean capaces de corregir sus trayectorias en los desplazamientos y ser lo más autónomos posible.

Cabe destacar, en cuanto a la *situación del producto en el mercado*, que éste es lanzado como una solución desarrollada bajo demanda; sin contemplar su comercialización como objetivo al considerarse una mera herramienta creada con el propósito de cubrir las necesidades del solicitante para las que no existía una solución concreta hasta la fecha. Paquetes ya existentes como “clearpath_kinect” o “robotino_kinect” no cubren ciertos fines como la compatibilidad y la adquisición de medidas individuales que fueron ya analizadas en los apartados correspondientes dentro del documento *Memoria*.

Así, aunque la aceptación generalizada no es un objetivo principal, el software se hará público como parte del repertorio de paquetes disponibles para el entorno “ROS” y cabe esperar cierto calado entre desarrolladores de proyectos similares a los del laboratorio solicitante y, sobre todo, que el código fuente sea utilizado en desarrollos posteriores que posean mayor o menor relación con la idea inicial.

3.3.2. Lista de usuarios

Se adjunta, a continuación, una tabla en la que se enumeran los tipos de usuarios identificados a priori para el producto, así como una breve descripción de cada uno de ellos. Como ya se adelantó, los usuarios no se corresponden con usuarios del sistema propiamente dichos sino usuarios finales con distintas aptitudes y propósitos y que, aunque se puedan calificar de “roles” en algún momento, no son identificables como tales por parte del software. La descripción más detallada corresponde al siguiente apartado.

Nombre	Descripción
Usuario Básico	Aquel profesional que sólo necesita el resultado final del software y que se limita a ejecutarlo y obtener los resultados en cada escenario y con la configuración ya determinada o modificando sólo los parámetros más obvios.
Usuario Administrador	Un usuario con conocimientos adicionales, que son necesarios para realizar la configuración del entorno, imprescindible si se desea obtener buenos resultados en distintas situaciones y condiciones.
Desarrollador	Se contempla la existencia de una persona capaz de reutilizar el código para nuevos proyectos que, si bien es discutible su inclusión en esta lista, debe tenerse en cuenta sin duda durante el proyecto, pues uno de los objetivos a los que está orientado es permitir mejoras y adaptaciones para nuevos desarrollos.

Tabla 3.1: Clases de usuarios a los que se destina el sistema

3.3.3. Perfil de usuario

1. Usuario básico

Se trata de una persona que, sin conocimientos específicos del funcionamiento del entorno en general ni condiciones específicas particulares, necesita obtener resultados de forma rápida y simple. Este tipo de usuario utilizará las características básicas del producto y disfrutará de un manejo sencillo y del funcionamiento autónomo del software para realizar tareas básicas que no requieran su intervención.

En resumen, priman en este caso la simplicidad y agilidad frente a la configurabilidad y adaptabilidad que corresponden al usuario administrador; pero sin dejar de lado la robustez y fiabilidad, que son exigibles en general.

2. Usuario avanzado

El producto que se desarrolla debe ofrecer lo necesario para un uso más avanzado que se adapte a las distintas condiciones en las que posiblemente se encuentre: escenario físico, dispositivos de entrada variados, capacidad de cómputo, etc. Como contrapunto, un uso más avanzado y configurable requiere más conocimientos y renunciar a la simplicidad inicial.

El usuario avanzado es una persona capaz de realizar dichas configuraciones y adaptaciones a nivel de parámetros para que la interfaz se conecte a nuevos dispositivos (además de los ya conocidos y configurados) y realice estimaciones en entornos más o menos complejos intensificando o acelerando los cálculos del algoritmo.

3. Desarrollador

El último perfil que se contempla no representa realmente un usuario del software pero sí de su código fuente cuando se necesite modificarlo o adaptarlo a nuevos desarrollos, tal como se exige en los objetivos del proyecto. Por ese motivo el diseño debe ser realizado sin olvidar su figura. El desarrollador debe tener conocimientos técnicos además de comprender la mecánica del proyecto y las necesidades de los otros usuarios, pues deben ser tenidas en cuenta al realizar las modificaciones.

3.3.4. Necesidades de los usuarios

Tras una sucesión de reuniones con un miembro representante del laboratorio de “CIS” y tras estudiar otras herramientas disponibles para el entorno “ROS”, surgen una serie de conclusiones sobre las necesidades que han de ser tenidas en cuenta en aras de facilitar el uso de estas nuevas herramientas al personal investigador del centro:

- **Desarrollo de una interfaz genérica** que permita el uso de diferentes aplicaciones y controladores de dispositivos conjuntamente cambiando solamente la configuración de dicha interfaz. Ésta debería simular la compatibilidad entre dichos elementos haciendo las veces de “traductora” de identificadores (nombres) de los recursos (parámetros, canales de comunicación, etc).
- **Autonomía** por parte de los elementos desarrollados, que una vez configurados y ejecutados deberán requerir la mínima interacción posible por parte de los usuarios.
- **Creación de una aplicación de odometría visual** capaz de estimar el movimiento del robot (y en consecuencia su posición y orientación) a partir de nubes de puntos obtenidas desde una cámara de profundidad.
- **La compatibilidad con MS Kinect** es particularmente imprescindible. Si bien se necesita una interfaz genérica para permitir el uso de otros dispositivos, las pruebas del laboratorio se realizarán con la aplicación de odometría desarrollada y sobre ese modelo en particular.
- **Ofrecer la máxima configurabilidad posible** de ambos elementos principales, no sólo para favorecer la compatibilidad con distintos dispositivos sino de cara a optimizar su uso en distintos escenarios y condiciones.
- **Un desarrollo ampliable** ya que se espera modificar y reutilizar el código. Esto incluye modularidad, claridad, buena documentación del código y la documentación técnica adicional.

3.4. Análisis de alternativas

Las secciones desarrolladas en adelante toman como base el análisis de posibles soluciones alternativas y la explicación en detalle de la alternativa seleccionada realizados en el apartado 1.4 correspondiente dentro del documento *Memoria*.

3.5. Especificación del sistema

En cualquier proyecto de ingeniería es importante prever con antelación a qué dificultades y necesidades se va a enfrentar el desarrollador (o grupo de desarrolladores) y, sobre todo, qué suerte de producto final es posible ofrecerle al usuario y qué características y necesidades cubrirá el mismo en las condiciones disponibles. Por ese motivo es necesario realizar un análisis previo de distintos aspectos del sistema para obtener la especificación de la arquitectura del mismo.

Así al final de esta sección deberá quedar clara la división en subsistemas que se tendrá en cuenta en adelante, los sistemas externos que colaboran con el estudiado y, finalmente, una explicación de las diferentes vistas del mismo con el fin de detallar su funcionalidad, organización y topología.

3.5.1. Diagrama de subsistemas

Un paso importante del análisis previo de un sistema software es su división en distintos subsistemas con el objetivo de discernir con mayor facilidad los objetivos y responsabilidades de cada parte. En este caso, y debido a la estructura de nodos, basta con considerar dichos nodos como los dos subsistemas principales, ya que se puede aceptar que ambos agrupan ya funcionalidades conceptualmente relacionadas entre sí. Es necesario, sin embargo, un tercer subsistema encargado de la configuración de los nodos, que se realiza de forma similar pero independiente haciendo uso del entorno de base como se explica en el siguiente título. En la figura 3.1 es posible apreciar el diagrama que representa los subsistemas del desarrollo actual que se acaban de justificar.

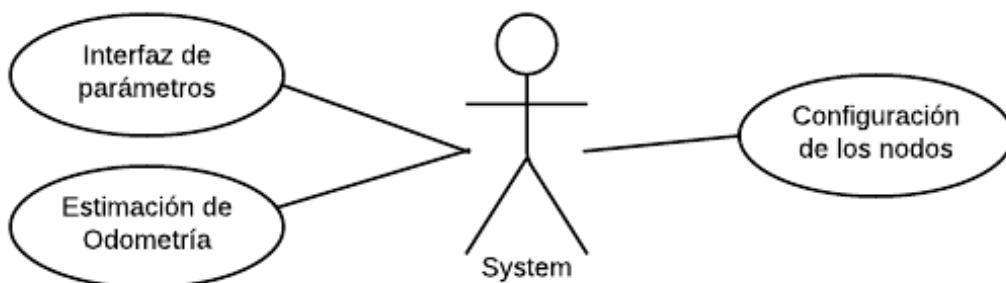


Figura 3.1: Diagrama de los subsistemas en los que se divide el sistema para el análisis

3.5.2. Interfaces con otros sistemas

Corresponde a este apartado explicar la posible integración y colaboración con otros sistemas externos que, sin ser parte del desarrollo actual, puedan ser parte complementaria imprescindible del producto final. Esto adquiere mayor importancia en este caso pues el software se presenta en forma de dos paquetes de utilidades específicamente enmarcadas como parte del entorno ROS para el que están diseñados. De ese modo, y aunque las funciones principales de cada paquete son parte de su propia implementación, las

comunicaciones y los parámetros del entorno han sido deliberadamente delegadas en ROS para externalizar su mantenimiento y optimizar el tiempo de desarrollo.

Lanzamiento

Desde el instante inicial en que se desea ejecutar un nodo del entorno ROS, es posible hacer uso de la bondades de los llamados “ficheros de lanzamiento” (*launchfiles*) del sistema. Éstos ficheros son interpretados por *ROS* y son responsables de diversos aspectos complementarios a la ejecución de los nodos.

Las principales responsabilidades de los ficheros de lanzamiento son la carga de parámetros en el servidor de parámetros, el lanzamiento del núcleo ROS (“*roscore*”) si no existe una instancia anterior y, sobre todo, la coordinación de múltiples nodos para garantizar la colaboración entre ellos en los casos que sea necesario.

En el paquete interfaz, los drivers se podrían ejecutar durante el lanzamiento gracias a esta herramienta que, al mismo tiempo informaría al nodo de ciertos valores de configuración como el nombre del fichero de traducción entre el controlador y la aplicación. Se puede ampliar la información sobre estos ficheros desde la documentación de ROS dentro de la siguiente dirección: “<http://ros.org/wiki/roslaunch>” y su uso básico se explica en el apartado 5.4 de los manuales de usuario.

Configuración

Adicionalmente a los ficheros específicos que se implementen para cada nodo, se utilizarán para la configuración básica, los parámetros del servidor de parámetros de ROS que son accesibles para lectura y modificación en todo momento desde cualquier nodo ejecutado en el entorno. Dichos parámetros pueden ser modificados desde el código del propio nodo o bien gracias a la herramienta específica de ros llamada “*rosparam*” que permite tanto el acceso a los parámetros individuales como la carga de ficheros de configuración en lenguaje YAML. Aún más importante es destacar que esta herramienta se integra con los ficheros de lanzamiento previamente introducidos, con lo que la carga automática de configuraciones a petición del usuario es perfectamente factible de forma simple y cómoda.

Comunicación

En cuanto a los canales de que disponen los nodos para comunicarse, se trata precisamente de los canales que el entorno les ofrece y se encarga de gestionar. Existen tres medios de comunicación en ROS:

- **Servicios.**- una implementación similar a las “Llamadas a procesos remotos” de otros entornos (“RPC” por sus siglas en inglés) permite a los nodos proveer de funciones a otros nodos y, por lo tanto, una interfaz intuitiva para que estos puedan comunicarse con los primeros. Gracias a los servicios es posible realizar solicitudes de reubicación de topics en la interfaz o solicitar una nueva lectura a la odometría visual.
- **Parámetros.**- un servidor global de parámetros en el que cada nodo tiene su espacio privado pero puede acceder a los de los otros nodos es el medio más simple de conocer el estado de otros nodos y dar a conocer el propio. En las últimas versiones se ha introducido el “Servidor de Reconfiguración Dinámica” (Dynamic Reconfigure Server) que permite desencadenar un aviso automático cuando se observan cambios en los parámetros.
- **Topics.**- los llamados “topics” son canales de envío y recepción que se mantienen abiertos constantemente permitiendo flujos continuos (o no) de mensajes. Tienen asignada una “dirección” bajo la cual trasmítir los datos que pertenecen al mismo flujo. Es la comunicación más directa dado que

cualquier nodo puede publicar datos en cualquier topic y todos pueden leerlos instantáneamente, así que se utiliza para los flujos continuos de datos.

La utilización de estos tres canales está disponible por defecto en las librerías ROS para cualquier nodo que sea instanciado en el entorno y no existen permisos ni restricciones, por lo que todos los nodos los utilizan.

3.5.3. Vista del sistema

La arquitectura del software no es un concepto unidimensional sino que está formado por múltiples perspectivas conceptuales. A modo de introducción para los próximos análisis, se propone en la ilustración 3.2 un conjunto de posibles vistas del sistema que representan las distintas aproximaciones posibles.

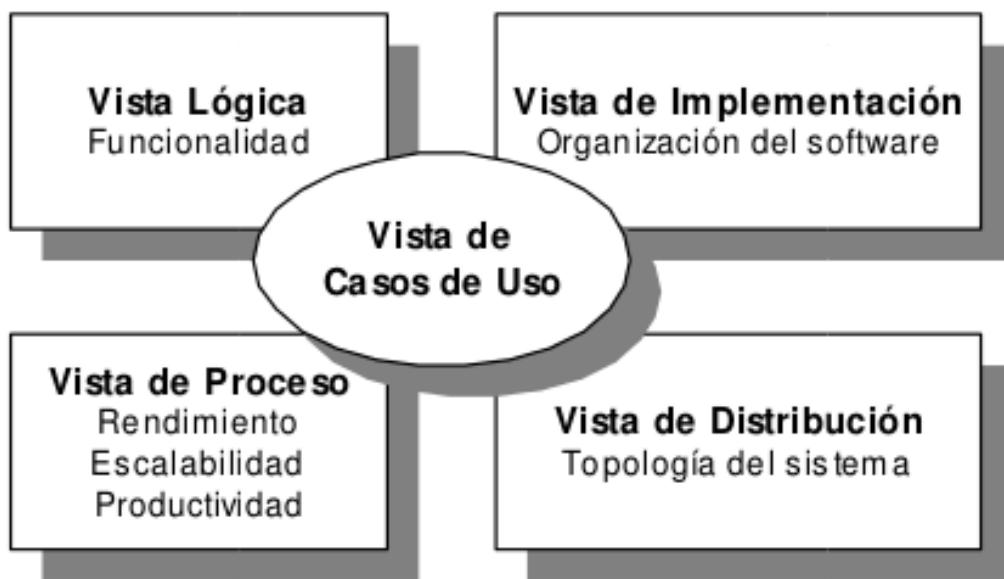


Figura 3.2: Vistas de la arquitectura

Vista Lógica

Desde la lógica de la arquitectura se muestran los requisitos funcionales del sistema, es decir, qué debería hacer el sistema en términos de servicio a los usuarios. La arquitectura lógica se captura en diagramas de clases. Se trata de un diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de entre clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargarán del funcionamiento y la relación entre uno y otro.

Vista de proceso

Centrándose en la estructura de la implementación en tiempo real del sistema, se presenta la vista de proceso, que tiene en cuenta requisitos como el rendimiento, la fiabilidad, escalabilidad, integridad, organización del sistema y sincronización. En esta vista también se usan los componentes. Los diagramas de componentes se usan para ver los diferentes componentes ejecutables creados para el sistema y también para ver qué componentes pueden compartirse entre sistemas o entre diferentes partes de un sistema.

Vista de implementación

Esta vista de la arquitectura se refiere a la organización de los distintos módulos de software dentro del entorno de desarrollo. La vista de implementación de la arquitectura debe tener en cuenta aspectos como la organización del software, la reutilización, como modularidad, facilidad de desarrollo y restricciones impuestas por los lenguajes de programación y las herramientas usadas en el desarrollo. Los elementos con los que se modela esta vista son los paquetes (*packages*) y los componentes junto con sus conexiones.

Vista de distribución

La vista de distribución de la arquitectura permite al equipo de desarrollo entender la topología final del sistema de cara a obtener una relación entre los componentes y los ejecutables del producto definitivo.

Vista de Casos de Uso

Finalmente, los casos de uso deben ser lo que ayude al desarrollador a convertir la vista lógica de lo que desea el cliente en una especificación coherente dentro de la vista de proceso y, a partir de ella, una guía para la implementación de un producto que cumpla con dichos casos de uso y, por tanto, con todos los planteamientos anteriores.

3.6. Entorno tecnológico de desarrollo

Los lenguajes en los que se basará el desarrollo son Python y C++ por su claridad y rendimiento respectivamente. En ambos casos se realizará la implementación sobre el framework ofrecido por el propio entorno ROS al que se orientan ambas piezas de la solución. El producto es, por lo tanto, compatible exclusivamente con Ubuntu GNU/Linux como sistema operativo.

Adicionalmente se utilizó el lenguaje YAML¹ mediante la librería “PyYaml”² como base para los ficheros de configuración de la interfaz y todas las herramientas de ROS que fue posible como los ficheros de lanzamiento (roslaunch) y el servidor de parámetros (rosparam) para delegar parte de la responsabilidad en las herramientas ya existentes (y probadas) y favorecer al mismo tiempo la máxima compatibilidad con futuras versiones del entorno (al ser herramientas mantenidas por sus creadores).

La librería de mayor importancia, sin embargo, es la llamada “Point Cloud Library (PCL)”, que es la única disponible para ROS capaz de interpretar y manejar las nubes de puntos en general. Con ella se realizarán las estimaciones de odometría a partir de las capturas de profundidad así como algunas operaciones auxiliares pero imprescindibles de conversión y filtrado de puntos.

En cuanto a la parte física, ya se insistió en la importancia de que ésta pueda variar con el tiempo y las necesidades, pero se enumerarán en este caso los elementos tenidos en cuenta inicialmente. Se utilizará un ordenador portátil de mediana potencia durante el desarrollo y las pruebas, así como la ejecución inicial del producto. Un periférico de consola *Microsoft Kinect* será utilizado como cámara de profundidad para la entrada de datos hacia el sistema, que a su vez estará conectado a un robot modelo *Amigobot* de la empresa *Mobile Robots*. Los detalles sobre dichos elementos hardware se pueden encontrar en la subsección 1.4.2 bajo el apartado *Descripción de la solución*.

¹Página web de YAML: “www.yaml.org”

²Página web de la librería “PyYaml”: “<http://pyyaml.org>”

3.7. Arquitectura del producto

La solución final se compondrá de dos paquetes software individuales que representen sendos nodos (procesos principales) obedeciendo al modelo de los paquetes del entorno ROS para el que se diseñan. Cada una de estas piezas software cumplirá una función principal que puede utilizarse de forma individual o complementándose con el otro paquete y comunicándose a través del entorno.

En ambos casos se tratará de aplicaciones de línea de comandos (sin interfaz gráfica de usuario) configurables mediante ficheros y parámetros previos a la ejecución y controlables mediante servicios ROS (llamadas remotas a las funciones disponibles). El primer nodo conforma una interfaz desde la cual un agente externo (usuario o nodo) puede solicitar cambios en el comportamiento de los controladores de dispositivos o la reubicación de canales de comunicación (topics de ROS). El segundo se encarga de realizar y publicar estimaciones de odometría de los movimientos del robot a partir de las nubes de puntos recibidas desde un dispositivo de entrada apropiado.

El resultado conceptual de unir dichos elementos con el entorno robótico de base y la parte física comentada en el entorno tecnológico (sección 3.6), se ilustra en el diagrama (no estándar) de la figura 3.3. En la representación se parte de la cámara de profundidad (un *Kinect* en el proyecto actual) cuyo controlador asociado adquiere y publica las capturas tridimensionales del entorno. Es el nodo interfaz el que conoce los recursos y parámetros de la cámara permitiendo así al de odometría averiguar la ubicación (topic) del flujo de entrada de datos o, en caso de necesidad, aplicar o detectar cambios en la configuración. Por ese motivo, existe una relación doble entre la interfaz y el controlador de la cámara pero una flecha directa del segundo al nodo de odometría visual, pues las capturas en tiempo real son escuchadas directamente a través de los topics que indique la interfaz.

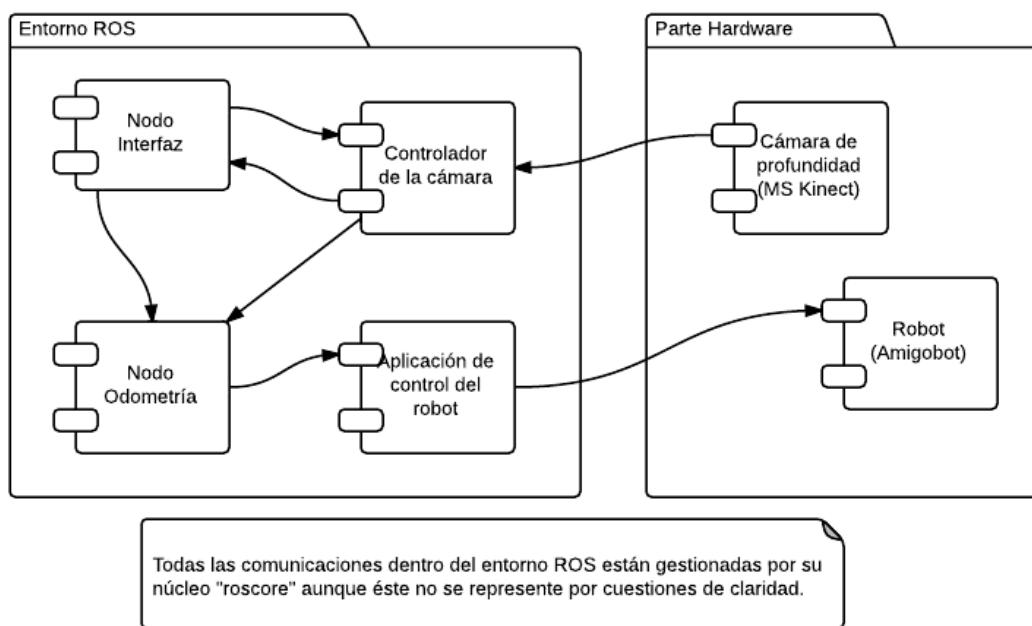


Figura 3.3: Diagrama de la arquitectura conceptual de la solución

De este modo, y una vez obtenidas las nubes de puntos, la odometría es capaz de publicar estimaciones que habiliten a tomar decisiones a una posible aplicación (externa) que esté controlando el robot. Aunque no se representa como parte de la arquitectura, el lazo de control queda cerrado cuando la cámara vuelve a capturar imágenes del escenario confirmando así cada reacción física del autómata.

Con el diseño actual, la interfaz soportará el manejo de varios drivers para un solo dispositivo, pero no permitirá (explícitamente) controlar varios dispositivos a la vez, que escapa al alcance acordado. El paquete de odometría, por su parte, también estará orientado a un solo periférico de entrada, aunque se podrían ejecutar varias instancias del mismo simultáneamente mediante el uso de ficheros de lanzamiento que varíen los nombres de los nodos.

3.8. Calidad exigida al producto

Este apartado está destinado a definir los rangos de calidad del presente software en términos de eficiencia, robustez, tolerancia a fallos y facilidad de manejo.

- Disponibilidad: el producto no supondrá restricciones de tiempo de ningún tipo, pudiendo ser utilizado las veinticuatro horas del día, todos los días de la semana (24/7).
- Facilidad de manejo: el sistema deberá ser intuitivo y fácil de usar amoldándose a las características del tipo de usuarios definidos en el apartado 3.3.3, “Perfil de usuario”. En este aspecto se contempla su usabilidad, especialmente para un uso básico.
- Configurabilidad: debe existir un balance entre la facilidad descrita en el punto anterior y la configurabilidad de ambas herramientas para que sea posible modificar y adaptar el comportamiento del software a distintas condiciones de uso.
- Mantenimiento: una vez el producto haya sido instalado no será *necesario* mantenimiento alguno para su uso básico, sin detrimento de que el código esté orientado a ser intuitivo y fácilmente adaptable en caso de necesidad.

3.9. Catálogo de requisitos del sistema y prioridades

Con la propuesta de este nuevo sistema se planteaban una serie de problemas y necesidades a solucionar (ya introducidos en el apartado 1.3.2). Estas cuestiones a abordar serán la base que dé lugar a los requisitos del sistema que deben cumplirse durante el presente desarrollo con el objetivo de alcanzar un software definitivo que dé solución a los problemas de la situación anterior y satisfaga las necesidades que se planteaban.

A continuación se presenta un listado de requisitos agrupados por su naturaleza y atendiendo a las consideraciones expuestas hasta ahora. Estos requisitos deberán servir para concretar las posibles soluciones y alternativas disponibles y valorar su validez tratando de cumplir siempre la mayor parte de los requisitos y de la forma más completa posible. Tal como se explica en el *Análisis de Alternativas* (1.4), las alternativas de diseño son limitadas por lo que se trata de estudiar qué requisitos podrán cumplirse con los recursos disponibles para confeccionar una nueva lista precisando sus prioridades y valorar cuáles de ellos podrían ser desestimados por ser prescindibles y/o costosos.

3.9.1. Tablas de requisitos

Características generales (CG)

Identificador	Descripción	Prioridad
CG 1.1	El sistema atenderá a órdenes de control del usuario provenientes de la línea de comando del sistema operativo.	Alta
CG 1.1.1	Ambos nodos implementados proveerán servicios externos para su control utilizando el entorno ROS como soporte.	Alta
CG 1.2	Durante la ejecución, se utilizarán para las configuraciones los parámetros nativos del sistema ROS.	Alta
CG 1.2.1	Se publicará la configuración actualizada en el servidor de parámetros en todo momento.	Alta
CG 1.3	El sistema, como conjunto, permitirá el uso del dispositivo <i>MS Kinect</i> en particular como periférico de entrada.	Alta
CG 1.4	Una configuración inicial, si existe, será leída y cargada al arrancar bien desde un fichero personalizado o desde el servidor de parámetros.	Media
CG 1.4.1	Se reconocerán en cada caso uno o varios ficheros de configuración (documentados en el <i>Manual de usuario</i> que permitan almacenar configuraciones para usos futuros.	Media
CG 1.4.2	En caso de no encontrarse una configuración externa, se cargarán valores por defecto que permitan el funcionamiento básico.	Alta
CG 1.5	La solución podrá trabajar en diferentes espacios de nombres del entorno según su configuración.	Alta
CG 1.5.1	Se cargará el proceso en un espacio de nombres anteriormente definido por el usuario.	Alta
CG 1.5.2	Se permitirán rutas relativas en todas las entradas de datos y configuraciones, de modo que sean independientes del espacio de nombres actual.	Alta
CG 1.6	El proceso adoptará como nombre público en el entorno ROS el nombre del fichero ejecutable sin la extensión si la hubiera.	Baja

Tabla 3.2: Grupo de requisitos 1: Características Generales

Nodo Interfaz (NI)

Identificador	Descripción	Prioridad
NI 2.1	La interfaz hará de intermediaria en la comunicación de aplicaciones y drivers no diseñados específicamente para comunicarse.	Alta
NI 2.1.1	Se traducirán los parámetros solicitados por las aplicaciones en los recursos de los drivers y viceversa.	Alta
NI 2.1.2	Se cargarán durante el arranque del nodo, los datos necesarios para la “traducción” entre las aplicaciones y los controladores.	Alta
NI 2.1.2.1	El sistema ofrecerá algún medio para conservar la configuración de cada driver y poder reutilizarla en futuras ocasiones.	Alta
NI 2.1.3	La interfaz deberá ser, en lo posible, transparente durante la comunicación.	Alta
NI 2.2	El nodo publicará un servicio que permita solicitar la realización de las tareas especificadas en los casos de uso.	Alta
NI 2.2.1	Los topics ROS serán reubicados ³ en tiempo de ejecución a petición de un posible agente externo.	Media
NI 2.2.2	Se publicarán servicios que permitan acceder a los valores de los parámetros del controlador para su lectura y modificación.	Media
NI 2.3	El sistema atenderá a modificaciones externas de sus parámetros mediante las herramientas de ROS disponibles a tal efecto.	Alta
NI 2.3.1	Se publicará un aviso a nivel de entorno cuando se realicen cambios en la configuración local.	Alta
NI 2.3.2	La interfaz conservará una caché con los valores conocidos de los parámetros del controlador.	Alta
NI 2.3.3	Se transmitirán al driver las modificaciones de los parámetros realizadas en tiempo de ejecución.	Media
NI 2.3.4	Los parámetros de nivel inferior (del driver) serán observados en busca de cambios para actualizar los valores propios.	Media

Tabla 3.3: Grupo de requisitos 2: Nodo Interfaz

³Se entiende por “reubicación” que se podrá encontrar en un nuevo emplazamiento o dirección del entorno. No implica, en este caso, que cese la existencia del topic original.

Nodo Odometría Visual (NO)

Identificador	Descripción	Prioridad
NO 3.1	El sistema procesará cada pareja de nubes de puntos recibidas (datos de entrada) para estimar la matriz de transformación que las relaciona.	Alta
NO 3.1.1	Se deberán reconocer correctamente movimientos “simples” con determinada amplitud máxima ⁴ , es decir: avanzar, retroceder o girar sin desplazarse.	Alta
NO 3.1.2	Se deberán reconocer correctamente movimientos “combinados” de dos movimientos simples.	Baja
NO 3.2	Se calculará, a partir de cada estimación, el movimiento correspondiente realizado por la cámara.	Alta
NO 3.2.1	Se calculará, a partir de cada movimiento de la cámara, el movimiento correspondiente realizado por el robot.	Alta
NO 3.2.1.1	Serán actualizadas, tras cada movimiento, la posición y orientación finales resultantes del robot respecto al punto de partida.	Alta
NO 3.2.2	Se relacionarán la posición y orientación de la cámara y del robot mediante la configuración al inicio.	Alta
NO 3.2.3	Se atenderá a un topic ROS durante la ejecución mediante el cual se podrán recibir nuevas matrices de transformación que definan la posición/orientación de la cámara respecto al robot.	Alta
NO 3.3	El nodo comprobará al arranque la posible configuración almacenada en los parámetros ROS.	Alta
NO 3.3.1	De no encontrar parámetros de configuración, los creará y exportará a partir de valores por defecto.	Alta
NO 3.3.2	Serán parámetros del nodo los parámetros del algoritmo para definir su comportamiento.	Alta
NO 3.3.3	Serán parámetros configurables los nombres de los topics que reciben los datos de entrada (como las nubes de puntos).	Media

Tabla 3.4: Grupo de requisitos 3: Nodo de Odometría (1)

⁴La amplitud máxima de los movimientos tendrá que ser determinada para cada caso en función del escenario, velocidad de procesamiento y optimizaciones o pistas del algoritmo.

Identificador	Descripción	Prioridad
NO 3.4	Se publicarán la estimación obtenida y datos relevantes a través del entorno ROS.	Alta
NO 3.4.1	Las matrices de transformación (tanto movimiento como posiciones fijas relativas) serán tratadas como TFs del sistema ROS ⁵ .	Alta
NO 3.4.2	El sistema publicará cada matriz de transformación correspondiente a las dos últimas nubes recibidas.	Alta
NO 3.4.3	Será publicada la matriz de transformación “acumulada” de la posición actual respecto al punto inicial.	Alta
NO 3.4.4	Las estimaciones de movimiento seguirá el convenio de ROS especificado bajo el apartado 1.7.2.	Alta
NO 3.4.5	Se transmitirá un valor o código que determine el estado actual del algoritmo.	Baja
NO 3.4.5.1	El código de estado del algoritmo será un valor entero según un convenio interno a determinar.	Baja
NO 3.4.5.2	El código de estado deberá reflejar por lo menos los estados de “error”, “en espera” e “iniciado” (o estados análogos).	Baja
NO 3.4.6	Se almacenarán y transmitirán estimaciones de odometría mediante los tipos de mensajes estándar ⁶ de ROS.	Alta
NO 3.4.6.1	Se usarán los tipos “tf::Transform” y “TFMessage” de ROS para almacenar y transmitir estimaciones.	Media
NO 3.4.6.2	Se usarán los tipos “nav_msgs::Odometry” y “Odometry” de ROS para almacenar y transmitir estimaciones.	Media
NO 3.4.7	Se publicará un mensaje propio que incluya todos los tipos de mensajes ⁷ implementados para notificar el estado completo en cada momento (odometría, último movimiento y estado)	Baja

Tabla 3.5: Grupo de requisitos 3: Nodo de Odometría (2)

A consecuencia del Requisito “NO 3.4.6.1”, las matrices de transformación manejadas por el sistema se denominarán en adelante “**Transform Frames**” o, abreviadamente, “**TF**” a lo largo de la presente documentación.

⁵El concepto de *TF* o “Transform Frame” se encuentra bajo el apartado 1.7.2 del document *Memoria*.

⁶Existen dos estructuras conceptualmente distintas que almacenan estimaciones en forma de matrices de transformación. En este caso confluyen en una utilidad común y se propone, simplemente, publicar ambas versiones para mayor compatibilidad

⁷ROS permite de forma nativa acceder a los campos de los mensajes publicados aún si el tipo completo del mensaje no es conocido.

Requisitos No Funcionales

La tabla siguiente expone los distintos requisitos no funcionales que se han encontrado:

Identificador	Descripción	Prioridad
RNF 4.1	Deberá mantenerse la estabilidad del sistema sin que se produzcan errores incontrolados o comportamientos anómalos.	Alta
RNF 4.1.1	Los errores controlados deben ser notificados al usuario de forma comprensible.	Alta
RNF 4.1.2	El sistema debe estar preparado para reaccionar ante los distintos escenarios posibles durante el manejo del programa.	Alta
RNF 4.2	El diseño debe favorecer un rendimiento homogéneo y acorde a las condiciones de uso.	Alta
RNF 4.2.1	El uso de la memoria debe ser razonable, optimizado para no ocupar más de lo necesario.	Alta
RNF 4.2.2	Los tiempos de espera durante el uso del nodo interfaz deben ser imperceptibles al no conllevar funciones de cálculo.	Alta
RNF 4.2.3	Es deseable que la velocidad de estimación de la odometría se optimice en lo posible dentro de las restricciones existentes de tiempo y recursos.	Baja
RNF 4.3	Ambas piezas software serán ampliamente configurables de forma que se permita variar al máximo su comportamiento.	Alta
RNF 4.3.1	La carga de configuraciones debe ser una tarea automatizable, fiable y completa (poder cargar toda la configuración de forma determinista y confiable).	Alta
RNF 4.4	El sistema funcionará como un conjunto en el que las dos partes se comuniquen correctamente entre sí y con el driver asignado.	Alta
RNF 4.5	El manejo del sistema debe estar adecuadamente orientado a los usuarios estudiados en el apartado 3.3.3 de la <i>Especificación (Lista de usuarios participantes)</i> .	Alta
RNF 4.5.1	Deberá existir la opción de un manejo básico para usuarios sin conocimientos, que maximice la simplicidad y claridad en el uso del sistema.	Alta
RNF 4.5.2	La configurabilidad del sistema debe poder realizarse de forma razonablemente fácil por parte de un usuario avanzado.	Alta

Tabla 3.6: Grupo de requisitos 4: Requisitos No Funcionales

3.10. Especificación de subsistemas

Corresponde a esta sección entrar en detalle sobre las especificaciones concretas para cada subsistema de los estudiados. Para ello se realizará una división de cada uno en diferentes casos de uso que se indicarán mediante un diagrama en cada caso y se describirá adecuadamente cada uno de los casos de uso resultantes.

3.10.1. Subsistema Interfaz de Parámetros

El primer subsistema, que se corresponde con el nodo interfaz, se encarga de compatibilizar aplicaciones y controladores y permitir la configuración de los segundos en tiempo de ejecución. Las funcionalidades que se presentan en los siguientes casos de uso responden precisamente a estos objetivos y necesidades.

Modelo de casos de uso

El diagrama con todos los casos de uso asociados a la interfaz se muestra en la figura 3.4 y atiende, como ya se adelantó, a las distintas funcionalidades de dicho subsistema. Dado que el usuario avanzado no interactúa como tal con el sistema (se caracteriza por crear ficheros de configuración como se aclara en los *Manuales de usuario*) existen dos agentes que pueden intervenir: el usuario (ya sea básico o avanzado) y las aplicaciones externas. Ambos roles pueden realizar todas las acciones y comenzar todos los casos de uso ya sea mediante el uso de servicios o del servidor de parámetros.

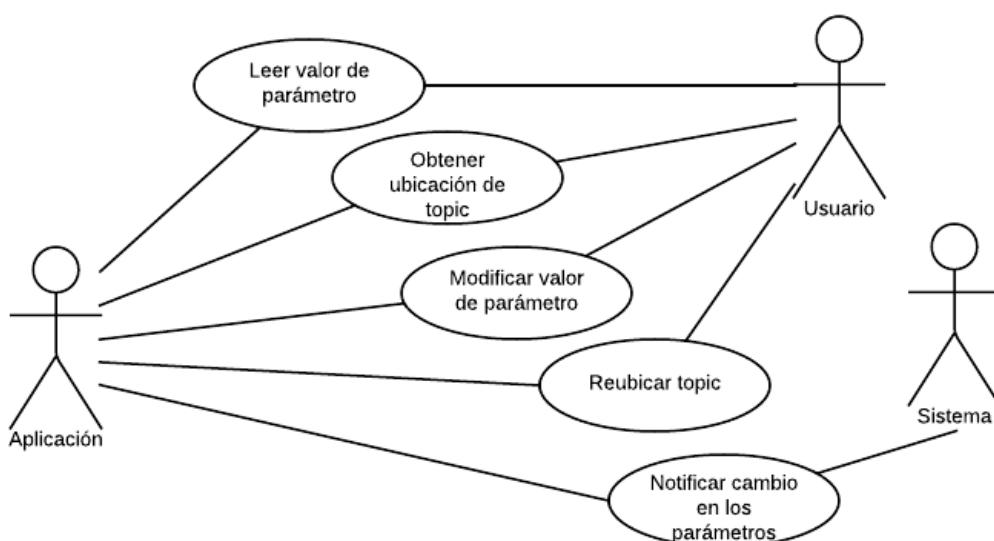


Figura 3.4: Diagrama de casos de uso del nodo interfaz

Un tercer rol es el del propio sistema, que se obvió en todos los casos por ser el receptor de las acciones, excepto en “Notificar cambio en los parámetros”, que desencadena la acción para que esta pueda ser

percibida desde otro nodo que esté a la escucha. A continuación se detallan todos los casos de uso en sus tablas correspondientes.

Descripción de casos de uso

En las siguientes tablas se presentan los casos de uso disponibles para la interfaz. En Los casos que se indique el uso de servicios ROS, estos se escribirán en forma “relativa”, es decir, que su dirección completa dependerá de la ubicación del nodo, que a su vez viene dada por el modo en que se lanza.

Caso de uso “Leer valor actual de parámetro”

Caso de uso: “Leer valor actual de parámetro”	
Numeración:	1.1
Precondición:	La interfaz está correctamente ejecutada y conectada a los drivers y conoce las “Traducciones” de los parámetros a leer.
Postcondiciones:	-
Excepciones:	-
Descripción:	<ul style="list-style-type: none">■ Opción A - Con el servicio correspondiente desde la línea de comando:<ol style="list-style-type: none">1. El usuario solicita la lectura de un parámetro mediante llamada al servicio correspondiente: <code><< rosservice call get/<tipo>Param <nombreParametro> ><</code>2. El valor resultante puede ser leído en la respuesta del comando.■ Opción B - Utilizando la herramienta de reconfiguración dinámica de ROS:<ol style="list-style-type: none">1. Se lanza la herramienta de reconfiguración de parámetros de ROS mediante el comando <code><< rosrun dynamic_reconfigure reconfigure_gui ><</code>2. Ha de elegirse el nodo correspondiente a la interfaz dentro de la herramienta externa.3. El usuario puede buscar la línea con el nombre del parámetro y leer el valor correspondiente.

Tabla 3.7: Caso de uso: “Leer valor actual de parámetro”

Caso de uso “Modificar valor de parámetro”**Caso de uso: “Modificar valor de parámetro”**

Numeración: 1.2

Precondición: La interfaz está correctamente ejecutada y conectada a los drivers y conoce las “Traducciones” de los parámetros a leer.

Postcondiciones: El parámetro modificado adopta el nuevo valor indicado por el usuario en el controlador.

Excepciones: Si valor solicitado no es válido según las restricciones del driver, la interfaz informará de esto y mantendrá el valor anterior para el parámetro.

Descripción:

- Opción A - Con el servicio correspondiente desde la línea de comando:
 1. El usuario solicita la modificación de un parámetro mediante llamada al servicio correspondiente con el valor deseado:
`<< rosservice call set/<tipo>Param <nombreParametro> <valor> >>`.
 2. El sistema responderá con una confirmación de la modificación.
- Opción B - Utilizando la herramienta de reconfiguración dinámica de ROS:
 1. Se lanza la herramienta de reconfiguración de parámetros de ROS mediante el comando
`<< rosrun dynamic_reconfigure reconfigure_gui >>`.
 2. Ha de elegirse el nodo correspondiente a la interfaz dentro de la herramienta externa.
 3. El usuario puede buscar la línea con el nombre del parámetro y escribir el valor deseado.

Tabla 3.8: Caso de uso: “Modificar valor de parámetro”

Caso de uso “Obtener ubicación de un topic”**Caso de uso: “Obtener ubicación de un topic”**

Numeración: 1.3

Precondición: La interfaz está correctamente ejecutada y conectada a los drivers y el topic pertenece a un parámetro conocido.

Postcondiciones: -

Excepciones: -

Descripción:

1. El usuario solicita la ubicación actual de un topic mediante llamada al servicio correspondiente con el valor deseado:
`«rosservice call get/TopicLocation <nombreParametro>`.
2. El sistema enviará el valor deseado como respuesta.

Tabla 3.9: Caso de uso: “Obtener ubicación de un topic”

Caso de uso “Reubicar Topic”**Caso de uso: “Reubicar Topic”**

Numeración: 1.4

Precondición: La interfaz está correctamente ejecutada y conectada a los drivers y el topic pertenece a un parámetro conocido.

Postcondiciones: El topic queda reubicado en la nueva dirección especificada desde la que puede ser leído en adelante.

Excepciones: Si el nombre introducido para el topic incluye caracteres no válidos según las restricciones de ROS, la interfaz se limitará a avisar de cuáles son dichas limitaciones.

Descripción:

1. El usuario solicita la reubicación de un topic escribiendo el parámetro al que se refiere y el destino deseado junto con la llamada al servicio correspondiente:
`«rosservice call set/TopicLocation <nombreParam><direccionFinal>`».
2. El sistema responderá con una confirmación de la modificación.

Tabla 3.10: Caso de uso: “Reubicar Topic”

Caso de uso “Notificación de cambio en los parámetros”

Caso de uso: “Notificación de cambio en los parámetros”	
Numeración:	1.5
Precondición:	La interfaz está correctamente ejecutada y conectada a los drivers. Este caso de uso lo ejecuta un nodo ROS externo previamente programado para ello.
Postcondiciones:	-
Excepciones:	-
Descripción:	<ol style="list-style-type: none">1. El nodo externo debe suscribirse a las actualizaciones del nodo interfaz. El modo de realizar esto técnicamente no corresponde a la parte de especificación por lo que no se entrará en detalle aquí.2. El subscriptor recibirá una notificación de la interfaz tras cada cambio, desencadenando una función de reacción (o <i>callback</i> que deberá estar previamente implementada).

Tabla 3.11: Caso de uso: “Notificación de cambio en los parámetros”

3.10.2. Subsistema Odometría Visual

La segunda parte del proyecto, la estimación de odometría tiene un solo cometido principal (el que indica su nombre), pero una serie de responsabilidades y funcionalidades secundarias en cuanto a publicar los resultados, resetearlos, o informar de su estado.

Modelo de casos de uso

Es en la figura 3.4 se muestra el diagrama con los casos de uso de la odometría. Al igual que en el caso anterior, intervienen dos roles principales que son el rol y la aplicación y que comparten las funcionalidades que no son iniciadas por el sistema. Vuelve a encontrarse éste último como iniciador, esta vez, de la publicación de las estimaciones resultantes tras los cálculos. En adelante se exponen las especificaciones para cada uno de los casos de uso representados.

Descripción de casos de uso

En las siguientes tablas se presentan los casos de uso disponibles para el subsistema de odometría. En Los casos que se indique el uso de servicios ROS, estos se escribirán en forma “relativa”, es decir, que su

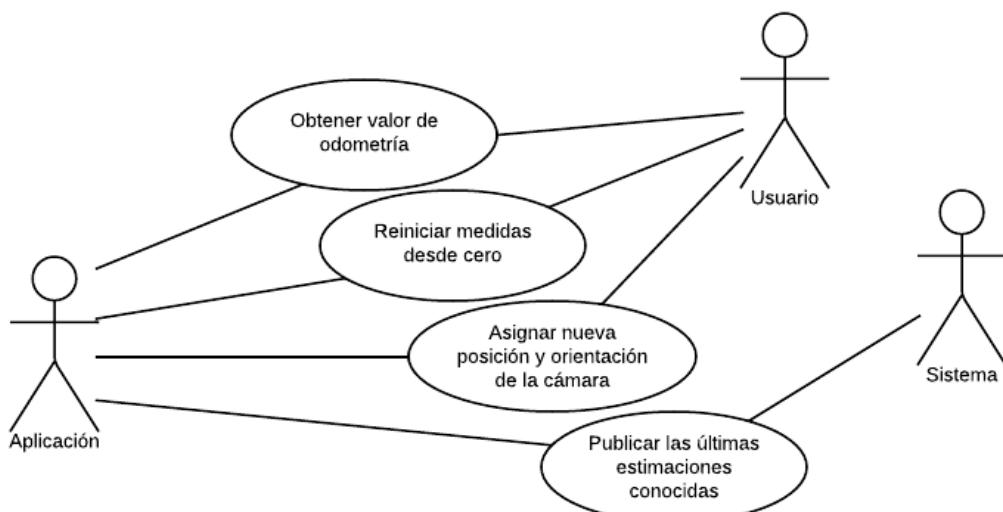


Figura 3.5: Diagrama de casos de uso del paquete de odometría.

dirección completa dependerá de la ubicación del nodo, que a su vez viene dada por el modo en que se lanza.

Caso de uso “Obtener estimación de odometría”

Caso de uso: “Obtener estimación de odometría”	
Numeración:	2.1
Precondición:	El nodo de odometría está en marcha con la configuración adecuada para recibir las nubes de puntos.
Postcondiciones:	-
Excepciones:	Si no existe una captura anterior, dado que es imprescindible comparar dos nubes de puntos, el sistema capturará la escena pero informará de la necesidad de solicitar nuevamente la estimación.
Descripción:	<ol style="list-style-type: none"> El usuario solicita la estimación nueva mediante llamada al servicio “updateOdometry”: <code>« rosservice call updateOdometry ».</code> El sistema enviará en respuesta toda la información conocida sobre la odometría en el instante actual: última estimación de movimiento, posición acumulada y estado del algoritmo.

Tabla 3.12: Caso de uso: “Obtener estimación de odometría”

Caso de uso “Reiniciar medidas. Puesta a cero”**Caso de uso: “Reiniciar medidas. Puesta a cero”**

Numeración: 2.2

Precondición: El nodo de odometría está en marcha con la configuración adecuada para recibir las nubes de puntos.

Postcondiciones: La estimaciones (posición, orientación y último movimiento realizado) se reinician a cero y se borra la nube de puntos anterior para comenzar desde una nueva ubicación.

Excepciones: -

Descripción:

1. El usuario ejecuta el servicio “resetGlobals” para reiniciar todos los cálculos:
`« rosservice call resetGlobals ».`
2. El sistema, tras realizar la tarea, responderá con los datos resultantes tras el reinicio; posición inicial, sin movimiento previo y estado a la espera.

Tabla 3.13: Caso de uso: “Reiniciar medidas. Puesta a cero”

Caso de uso “Asignar nueva posición y orientación de la cámara”**Caso de uso: “Asignar nueva posición y orientación de la cámara”**

Numeración: 2.3

Precondición: El nodo de odometría está en marcha con la configuración adecuada para recibir la TF⁸ de la cámara. La TF que se envía debe ser válida.

Postcondiciones: La TF queda almacenada de manera que, en adelante, las transformaciones se calcularán teniendo en cuenta la nueva posición y orientación de la cámara.

Excepciones: -

Descripción:

1. Primeramente el parámetro “topic_camera_tf” permite averiguar qué topic tiene asignada la función de recibir la TF de la cámara:
`«rosparam get topic_camera_tf ».`
2. Seguidamente se puede solicitar a un nodo externo que envíe, a través del mismo, una TF con la posición y orientación de la cámara.⁹
3. La recepción de la TF desencadena un evento que se encarga de almacenarla adecuadamente.

Tabla 3.14: Caso de uso: “Asignar nueva posición y orientación de la cámara”

⁸TF significa *Transform Frame*. El concepto se explica en el apartado de requisitos de la odometría y en el apartado apartado 1.7.2 de la *Documentación auxiliar*.

⁹Se propone un modo de realizar la publicación en el *Manual de Usuario* de la odometría.

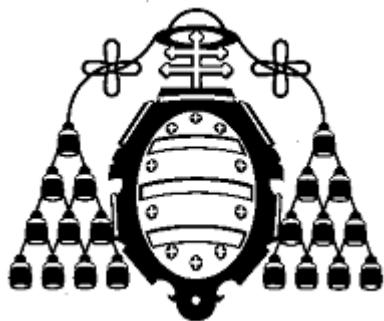
Caso de uso “Publicar las últimas estimaciones conocidas”

Caso de uso: “Publicar las últimas estimaciones conocidas”	
Numeración:	2.4
Precondición:	El nodo de odometría está en marcha a la espera de nubes de puntos en modo automático
Postcondiciones:	-
Excepciones:	-
Descripción:	<ol style="list-style-type: none">1. El nodo de odometría se encuentra a la escucha en modo automático.2. Un nodo externo que hace las veces de “fuente” publica una nube de puntos.3. La llegada de una nueva nube desencadena el cálculo de la estimación de odometría.4. Una vez calculada la matriz de transformación entre las dos últimas nubes, ésta es publicada mediante el topic ROS correspondiente.

Tabla 3.15: Caso de uso: “Publicar las últimas estimaciones conocidas”

3.10.3. Subsistema Configuración del entorno

Si bien el subsistema de configuración es tan importante como los otros dos y responde a varios requisitos funcionales, su funcionamiento viene previamente determinado por un sistema externo (el entorno ROS) y sólo es menester implementar la “interpretación” de las configuraciones, pero no la interfaz de manejo para el usuario. Por ese motivo no se estudiarán los casos de uso (que servirían para implementar las correspondientes funcionalidades) sino que sólo se tendrá en cuenta este subsistema durante el apartado final de pruebas y con el objetivo de comprobar la carga correcta de dichas configuraciones por parte de los nodos.



UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE
GIJÓN

**ÁREA DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL**

PROYECTO FIN DE CARRERA Nº 3122067

**INTEGRACIÓN DEL DISPOSITIVO 'MICROSOFT KINECT'
COMO PARTE DE UNA ARQUITECTURA ROBÓTICA
GENÉRICA PARA SU USO COMO DISPOSITIVO DE ENTRADA**

DOCUMENTO N° IV

DISEÑO E IMPLEMENTACIÓN



MIGUEL A. GARCÍA GONZÁLEZ

NOVIEMBRE – 2012

TUTOR: LUCIANO SÁNCHEZ RAMOS

CAPÍTULO 4

DISEÑO E IMPLEMENTACIÓN

4.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

4.1.1. Identificación del proyecto

Título:	“Integración del dispositivo ‘Microsoft Kinect’ como parte de una arquitectura robótica genérica”
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

4.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo “Microsoft Kinect” comercializado por la empresa “Microsoft Corporation”. La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

4.1.3. Visión general del documento

A lo largo de las siguientes páginas se exponen los pasos del diseño a nivel de software que se corresponderían principalmente con algunas actividades de la metodología “Métrica V3” en su documento de “Diseño del Sistema de Información”. El objetivo es definir el entorno tecnológico y la arquitectura final de la solución, ya no desde un punto de vista preliminar sino de cara a la implementación.

Al término del diseño, y una vez impelmentada la solución resultante, se dedicará una sección al código fuente y se finalizará con el apartado de pruebas de aceptación para comprobar el funcionamiento adecuado del producto.

4.2. Diseño de la arquitectura del sistema

4.2.1. Entorno tecnológico de implantación

El contexto para el que surge la presente solución, se puede encontrar en el laboratorio de *Sistemas Inteligentes Colaborativos* del *Centro Europeo de Soft Computing* sito en la ciudad de Mieres en Asturias. En dicho departamento, se ocupan de la programación de diferentes agentes físicos o software que deben interactuar entre sí y trabajar con un objetivo común aprovechando diferentes recursos y capacidades de cada uno, es decir: se pretende que dichos entes colaboren.

El proyecto “Absynthe” es sólo un ejemplo del trabajo realizado en el laboratorio y en el cual se podría utilizar la herramienta que se desarrolla. “Absynthe” pretende aplicar los últimos avances tecnológicos en este campo y, al mismo tiempo, demostrar su utilidad práctica en un entorno controlado pero realista. Algunos objetivos para lograrlo son: el uso de órdenes y descripciones cualitativas en la comunicación (obedeciendo a una lógica borrosa frente a la lógica clásica utilizada comúnmente hasta ahora), órdenes que impliquen cooperación con otros entes y la navegación autónoma a partir de la información de los sensores.

En tal contexto es útil contar con una herramienta que permita utilizar de forma genérica los dispositivos similares (como cámaras de profundidad de diferentes marcas y modelos) para que estos sean compatibles con aplicaciones software en común. Esto resume la primera parte del proyecto, una interfaz que permita utilizar diferentes periféricos sin modificar el software de la capa superior.

Dado que la navegación del robot debe ser autónoma, ahí entra la segunda parte del desarrollo: una estimación de la odometría del robot que permita conocer su localización aproximada tras una serie de movimientos. Es conveniente tener cierta redundancia entre la odometría de ruedas (u otras fuentes) y la posición calculada visualmente para complementar las medidas y reforzar su fiabilidad.

Si bien se ha introducido el actual trabajo realizado en el centro, es importante subrayar que el objetivo de esta sección es exclusivamente emmarcar la solución en su entorno de aplicación, sin que ésta pertenezca realmente a ningún desarrollo específico en particular. El proyecto “Absynthe” es sólo un ejemplo que sirve de orientación para la creación de esta herramienta que espera ser útil para ese y otros proyectos futuros, así como agilizar y facilitar el trabajo de los miembros de la unidad en adelante.

Equipo lógico

Los requisitos a nivel de Software para ejecutar la solución son, esencialmente, los entornos de base: un Sistema Operativo *Ubuntu GNU/Linux (versión 10.04 en este caso)* y el entorno ROS (Robot Operating System) al que se orienta, en su versión “Fuerte” concretamente. Adicionalmente se utiliza un intérprete de lenguaje python en la ejecución del primer nodo, si bien ya viene incluido con la versión de Linux elegida, al igual que las herramientas de intérprete de comandos y de editor de texto necesarias para el control del software durante su ejecución y para modificar los ficheros de configuración.

Adicionalmente será necesario descargar ciertos paquetes de los disponibles en el repertorio de ROS como son el controlador del *Kinect* (llamado *openni* y que engloba controladores para otras cámaras) y la librería de nubes de puntos (PCL, del inglés “Point Cloud Library (PCL)”), que es la única disponible para ROS capaz de interpretar y manejar las nubes de puntos provenientes del *Kinect*. Con ella se realizan las estimaciones de odometría a partir de las capturas de profundidad así como algunas operaciones auxiliares pero imprescindibles de conversión y filtrado de puntos.

Equipo físico

En lo referente al hardware, se implantará la solución utilizando una cámara de profundidad para la adquisición de datos, un robot que haga las veces de portador y el equipo mínimo contemplado en las características hardware durante el análisis preliminar en la *Descripción de la solución* (apartado 1.4.2).

El periférico de entrada será el dispositivo de juego *Microsoft Kinect* haciendo uso, exclusivamente, de la cámara de profundidad que incorpora, ya que el resto de sensores escapan al alcance del proyecto actual. El portador del anterior será un robot modelo Amigobot de los ya disponibles en el laboratorio, fabricado por la empresa *Mobile Robots* y que se conectará con el primero sólo a través del PC que incorpora el software para operarlos. Los detalles sobre estos dispositivos se encuentran en el apartado 1.7 de la *Memoria*, sobre *Documentación Auxiliar*.

La implantación inicial se realiza sobre el ordenador personal utilizado para el desarrollo, cuyas características coinciden con los requisitos mínimos de funcionamiento de la solución. Se trata de un PC con 2 GBytes de memoria de tipo RAM, suficientes para satisfacer el consumo de memoria del algoritmo; una procesador *Intel Pentium T4500* de 2.3 GHz, que permite realizar las estimaciones en un tiempo razonable (aunque mejorable) y un amplio disco duro con más de 50 GBytes que permite realizar la instalación completa del entorno: unos 4 GBytes sin contabilizar el sistema operativo que se asume ya instalado.

Comunicaciones

Con el diseño actual del sistema, y según las necesidades contempladas, sólo serán necesarias comunicaciones de dos tipos: externas entre el PC y los dispositivos, e internas al sistema ROS entre los nodos que participan. En ningún caso se contempla la comunicación entre terminales ni de forma directa entre el robot y la cámara.

En el caso de las comunicaciones dispositivo-PC, la conexión de cada dispositivo (USB por parte de la cámara y cable Ethernet desde el robot) determina los conectores a utilizar (USB tipo A y RJ45 respectivamente) y los controladores incluidos en ROS gestionan el intercambio de datos y comandos con ambos. Es importante mencionar que no se contempla ningún controlador en particular para el robot pues ello queda fuera del alcance y sólo corresponde aportar los datos de odometría que deberían ser utilizados en su caso, mientras que los datos de la cámara llegan en este caso a través del driver “*openni*” implementado en forma de paquetes ROS.

Los nodos, por su parte, deben comunicarse a través de los tres canales provistos por ROS y que ya fueron introducidos anteriormente:

- **Servicios.-** una implementación similar a las “Llamadas a procesos remotos” de otros entornos (“RPC” por sus siglas en inglés) permite a los nodos proveer de funciones a otros nodos y, por lo tanto, una interfaz intuitiva para que estos puedan comunicarse con los primeros. Gracias a los servicios es posible realizar solicitudes de reubicación de topics en la interfaz o solicitar una nueva lectura a la odometría visual.
- **Parámetros.-** un servidor global de parámetros en el que cada nodo tiene su espacio privado pero puede acceder a los de los otros nodos es el medio más simple de conocer el estado de otros nodos y dar a conocer el propio. En las últimas versiones se ha introducido el “Servidor de Reconfiguración Dinámica” (Dynamic Reconfigure Server) que permite desencadenar un aviso automático cuando se observan cambios en los parámetros.
- **Topics.-** los llamados “topics” son canales de envío y recepción que se mantienen abiertos constantemente permitiendo flujos continuos (o no) de mensajes. Tienen asignada una “dirección” bajo la cual trasmisitir los datos que pertenecen al mismo flujo. Es la comunicación más directa dado que cualquier nodo puede publicar datos en cualquier topic y todos pueden leerlos instantáneamente, así que se utiliza para los flujos continuos de datos.

La utilización de estos tres canales está disponible por defecto en las librerías ROS para cualquier nodo que sea instanciado en el entorno y no existen permisos ni restricciones, por lo que todos los nodos los utilizan.

4.3. Especificación de subsistemas de diseño

Para el diseño se ha optado por dividir el sistema en dos partes que coinciden con los dos nodos que componen la solución. La decisión se basa en la estrecha relación que guardan las funcionalidades principales abarcadas en cada nodo, puesto que comparten el mismo objetivo final en cada caso. Los módulos a implementar, por su parte, serán subconjuntos de código siguiendo la misma lógica: agrupar las funcionalidades que persiguen un mismo subobjetivo (como son la traducción o la comunicación). La primera ilustración de la sección (figura 4.1) está dedicada a representar los subsistemas de diseño mencionados junto con sus clases, mientras que las relaciones entre sus módulos se basan en las dependencias de funcionamiento durante su uso (en tiempo de ejecución).

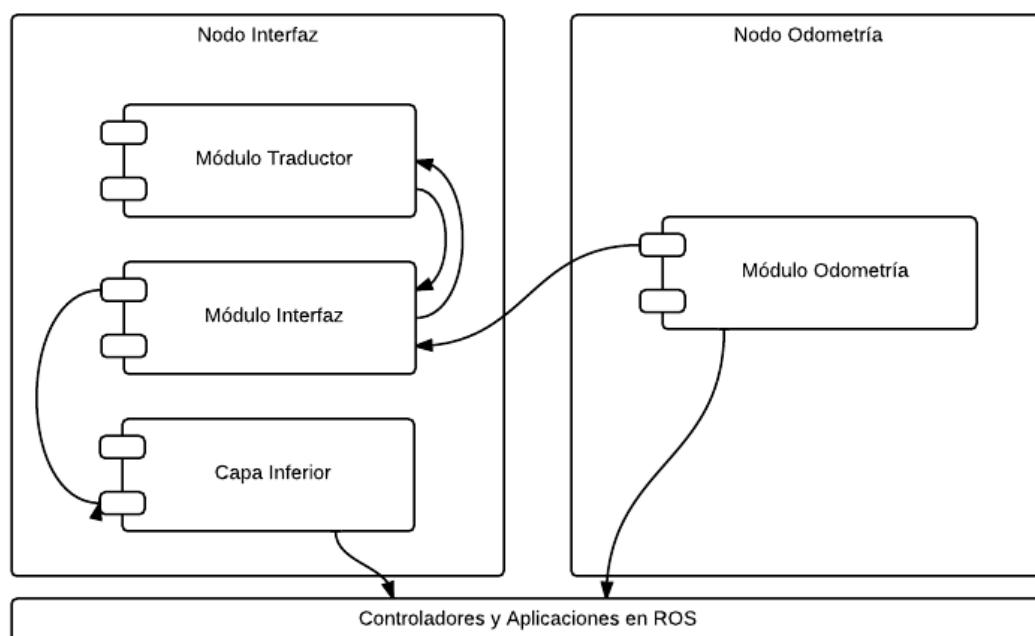


Figura 4.1: Diagrama de subsistemas de diseño

En particular se incluyen cuatro módulos principales. La interfaz contiene una parte que se denomina “capa inferior” y que representa el nivel de abstracción más bajo para comunicarse con el driver; una “interfaz” que representa la comunicación con las aplicaciones de la clase superior (como la odometría, en este caso); y el “traductor” que engloba todas las funciones e información para las traducciones. La parte de odometría, por su parte, sólo realiza la función de filtrado de nubes de puntos y ejecutar el algoritmo (externo) para obtener los cálculos, por lo que todo ello se puede englobar en un módulo común.

4.3.1. Relación de subsistemas de diseño

En cuanto a las relaciones entre los subsistemas y dentro de cada uno de ellos, es conveniente pararse a analizarlo antes de diseñar definitivamente la estructura de clases. El nodo de odometría, que conforma un único módulo, hace las veces de “aplicación” y, por lo tanto, necesita a la interfaz para comunicarse

con el controlador del *Kinect*. Una vez que obtiene los datos necesarios o asigna la configuración deseada (si la hubiera), podrá comunicarse directamente con el controlador al conocer el identificador del recurso deseado. Por lo tanto el subsistema de odometría se comunica con la interfaz y con el controlador y, del mismo modo, la interfaz se comunica con ambos para permitir que se reconozcan.

Las relaciones de la figura 4.1 se refieren también a las interrelaciones de los módulos en la interfaz: el módulo interfaz representa la capa superior que realiza la comunicación con las aplicaciones, como se acaba de introducir con la odometría. Para ello debe conocer al traductor de los identificadores de recursos y también el módulo de capa inferior que será quien se comunique con los drivers del entorno. Así, la comunicación entre la interfaz y la capa inferior representa la misma entre las aplicaciones y los controladores, con la salvedad de que se aplicará la traducción entre ellos.

4.3.2. Entorno tecnológico de desarrollo

La herramienta de desarrollo elegida ha sido *Eclipse IDE 4.2 (Juno)*. Esta herramienta programada en Java y con licencia de uso gratuito *Eclipse Public License*, ofrece un entorno integrado de desarrollo muy configurable y válido tanto para lenguaje Java como C++ y Python mediante la adición de extensiones desde el propio menú del entorno.

Este entorno multiplataforma tiene múltiples versiones específicas (según necesidades del desarrollador) todas ellas compatibles tanto con *GNU/Linux* como *MS Windows* y *Mac OS X* en sus respectivas versiones tanto de 32 como de 64 bits.

En cualquier caso, es importante destacar que la creación de los ejecutables corre siempre a cargo de las herramientas del sistema ROS, que es en realidad el responsable de la compilación de los paquetes creados como nodos del entorno. Dicho sistema de nombre “Robot Operating System” (versión *Fuerte*) es, como ya se introdujo con anterioridad, el entorno de base para la ejecución de la solución, así como el responsable de la comunicación entre ellos y un conjunto completo de herramientas para la creación, manejo y testeо de los mismos. ROS está programado especialmente para funcionar en sistemas Linux, aunque sus creadores ofrecen ya una versión “*win_ros*” para Windows que podría ampliar la compatibilidad de la solución sin necesidad de modificar el código fuente.



Figura 4.2: Logotipos de las tecnologías que forman el entorno de desarrollo (apartado 4.3.2)

Finalmente, se han elegido dos lenguajes de programación para las dos partes del proyecto. La justificación detallada se encuentra en el apartado de Análisis de Alternativas (1.4). Corresponde ahora indicar que se utiliza Python para el nodo interfaz por su claridad, rapidez al desarrollar y facilidad para la conversión o coerción de tipos; y que se prefirió C++ para la odometría por compatibilidad con la librería PCL que maneja las nubes de puntos y por cuestiones de eficiencia de los cálculos.

4.4. Diseño detallado de las interfaces de clases

4.4.1. Diagrama general de clases

La estructura de clases diseñada para la implementación de la solución ha sido plasmada en el diagrama de la figura 4.3. Por un lado se ha mantenido la división en dos subsistemas correspondientes a los dos paquetes planteados ya anteriormente. Por otro lado se mantienen los módulos de la interfaz en forma de clases, pero se desdobra el módulo de odometría en dos clases para separar el concepto de “Transform Frame” (dato) de la odometría en sí (algoritmo). El recorrido que se realizará posteriormente por las distintas clases propuestas profundizará en el contenido de estas para que el lector pueda conocerlas más a fondo.

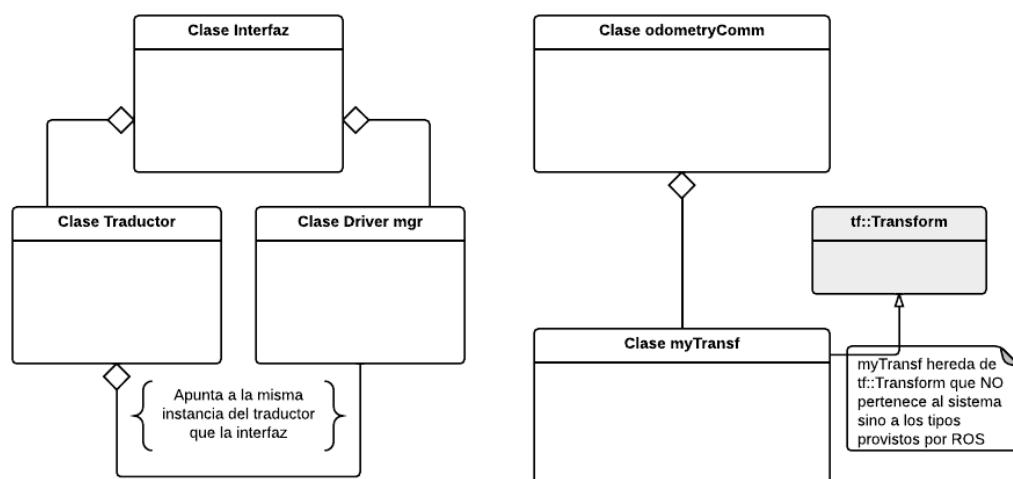


Figura 4.3: Diagrama general de clases del sistema.

4.4.2. Interfaces de clases

En aras de conocer mejor las posibilidades, responsabilidades y características de cada clase del diseño, y de obtener una documentación más clara y completa, se dedicarán los siguientes apartados a enumerar una a una dichas clases junto con su especificación a nivel de diseño.

Subsistema Interfaz - Clase *upward_interface*

La clase “*upward_interface*”, aunque ideada a priori para hacer las veces de capa superior de abstracción, destaca por incluir la responsabilidad de contener instancias de las dos clases restantes y ser el punto de partida de las acciones principales. Así, ésta publica los servicios y parámetros que son accesibles desde agentes externos y desencadena los cambios en el driver en dos pasos: primero traduce el identificador mediante la clase traductora y después se lo pasa a la capa inferior junto con el valor deseado. Cuando los

cambios comienzan en el controlador es la clase *driver_manager* la que utiliza *iface_translator* y solicita los cambios a la capa superior.



Figura 4.4: Bloque UML de la clase *upward_interface* del nodo interfaz

Subsistema Interfaz - Clase *iface_translator*

Esta clase responde a la necesidad de una herramienta simple de traducción, por lo que reúne solamente tres funcionalidades principales (a parte de las auxiliares que puedan surgir): realiza la carga de su propio fichero de configuración (que podría considerarse un diccionario de identificadores), responde a solicitudes de “traducción” de identificadores desde la capa superior para obtener el parámetro asociado y, finalmente, permite obtener los primeros a partir de los nombres de parámetros conocidos por la capa inferior.

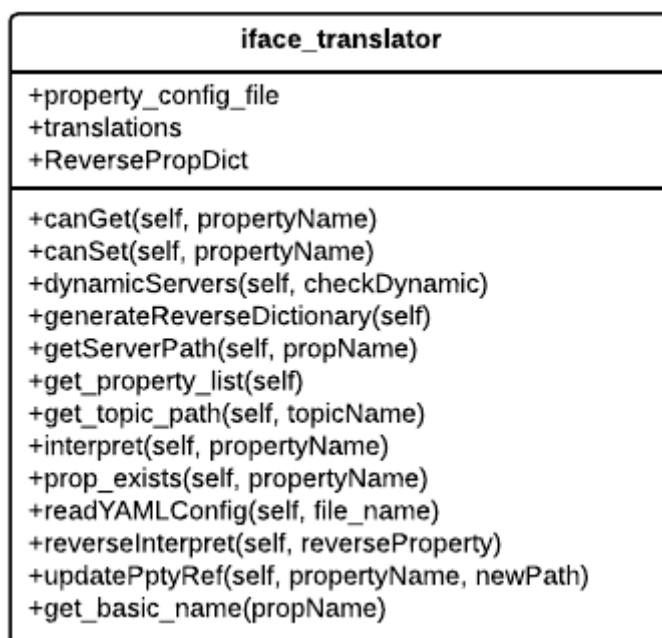


Figura 4.5: Bloque UML de la clase *iface_translator* del nodo interfaz

Subsistema Interfaz - Clase *interfaz-driver_manager*

La capa inferior de los subsistemas iniciales pasa a ser la clase *interfaz-driver_manager*, encargada de la comunicación con los controladores de dispositivos. Esto conlleva atender a los cambios en los controladores, ya sean en el servidor de parámetros o mediante topics y también modificar los mismos a petición de las capas superiores.

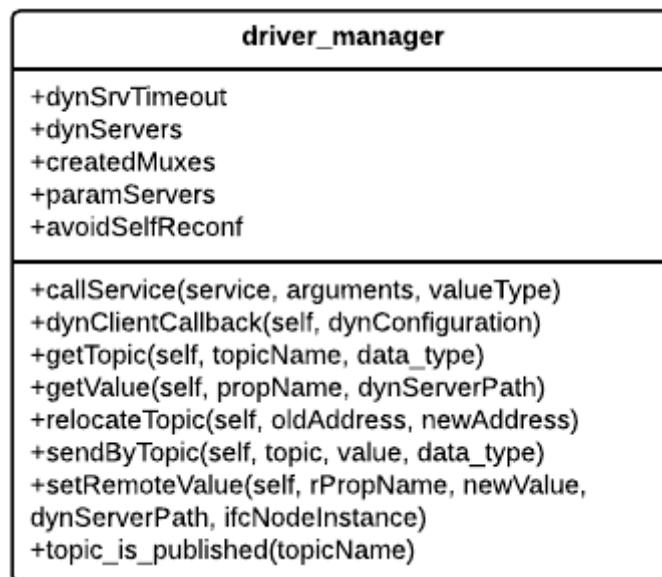


Figura 4.6: Bloque UML de la clase *driver_manager* del nodo interfaz

Subsistema Odometría - Clase *odometryComm*

En el caso de la odometría, dado que todas sus responsabilidades giran en torno a “obtener la siguiente TF”, se decidió agrupar todas las funcionalidades en una sola clase. Con esa aproximación, su objetivo se resume esencialmente en aplicar filtrados previos a la estimación, lanzar la estimación, y realizar el tratamiento necesario para que los datos obtenidos sean útiles.

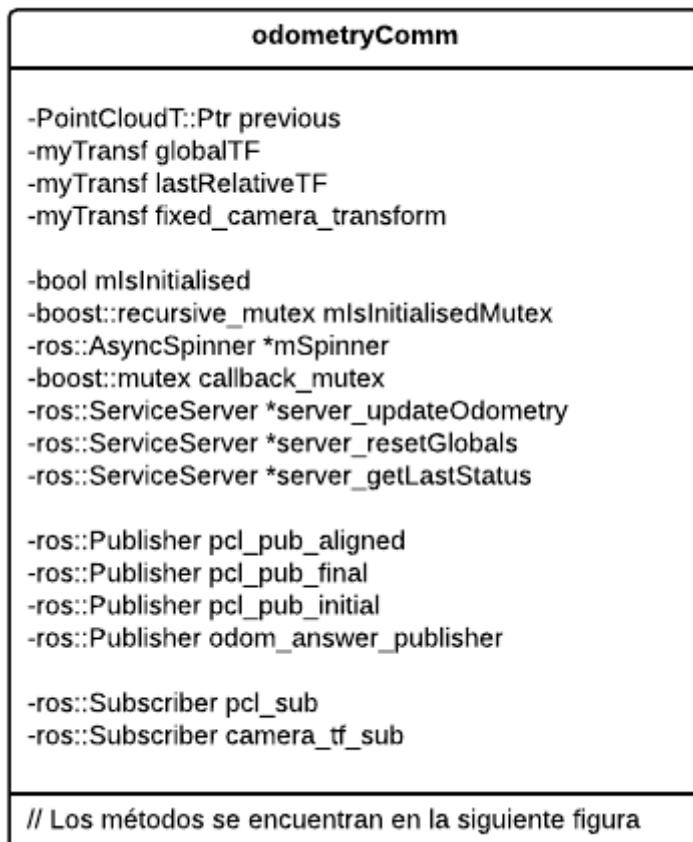


Figura 4.7: Bloque UML de la clase *odometryComm* del nodo odometría (atributos)

```

odometryComm

// Los atributos se encuentran en la figura anterior

+bool init(int argc, char** argv, std::string nodeName, uint32_t rosInitOptions)
+bool shutdown()

-void setDownsampleFiltering(float newLeafSize)
-void setDistanceFiltering(float minimum, float maximum)
-void setNoiseFiltering(float radDist, int noOfNeighbours)

-void printConfiguration()
-void apply_filters(PointCloudT::ConstPtr pointCloud1_in, PointCloudT::Ptr &pointCloud1_out)
-void trimPreviousCloud(PointCloudT::Ptr &pointCloud1_out, double x, double y, double z)
-double round(double value, int noDecimals)
-void printMatrix4f(Eigen::Matrix4f &tMatrix)
-bool generate_tf(Eigen::Matrix4f &mat, double roll, double pitch, double yaw)
-void setOdomStatus(int status)
-double matToDist(Eigen::Matrix4f t)
-void matToXYZ(Eigen::Matrix4f t, double& x, double& y, double& z)
-void matToRPY(Eigen::Matrix4f t, double& roll, double& pitch, double& yaw)
-void filter_resulting_TF(myTransf *targetTF)
-myTransf eigenToTransform(Eigen::Matrix4f tMatrix)
-bool fill_in_answer(my_odometry::odom_answer &res)

-Eigen::Matrix4f process2CloudsICP(PointCloudT::Ptr &cloud_initial, PointCloudT::Ptr
&cloud_final, double *final_score_out)

-bool update_odometry(my_odometry::odom_update_srv::Request &req,
my_odometry::odom_update_srv::Response &res )
-bool get_last_status(my_odometry::statusMsg::Request &req,
my_odometry::statusMsg::Response &res )
-bool reset_globals(my_odometry::emptyRequest::Request &req,
my_odometry::emptyRequest::Response &res )

-void cameraTF_callback(const tf::tfMessageConstPtr& newTF)
-void PCL_callback(const PointCloudT::ConstPtr & cloud_msg)
-void STR_callback(const std_msgs::String::ConstPtr & nextTopic)
-void common_callback_routine(PointCloudT::ConstPtr & pointCloud1_aux, std::string
nextTopic)

```

Figura 4.8: Bloque UML de la clase odometryComm del nodo odometría

Subsistema Odometría - Clase *myTransf*

La clase *myTransf* surge de la conveniencia de agrupar la amplia variedad de métodos que se necesitarán para el tratamiento de la TF, es decir, el conjunto de valores que describe la estimación de odometría. El entorno ROS incluye por defecto una clase llamada *tf::Transform* que permite manejar un movimiento en 6 dimensiones (desplazamiento y orientación) o, del mismo modo, una posición orientada en el espacio.

La presente heredará de `tf::Transform` manteniendo el tipo de dato pero ampliando en buena medida el abanico de métodos que ofrece.

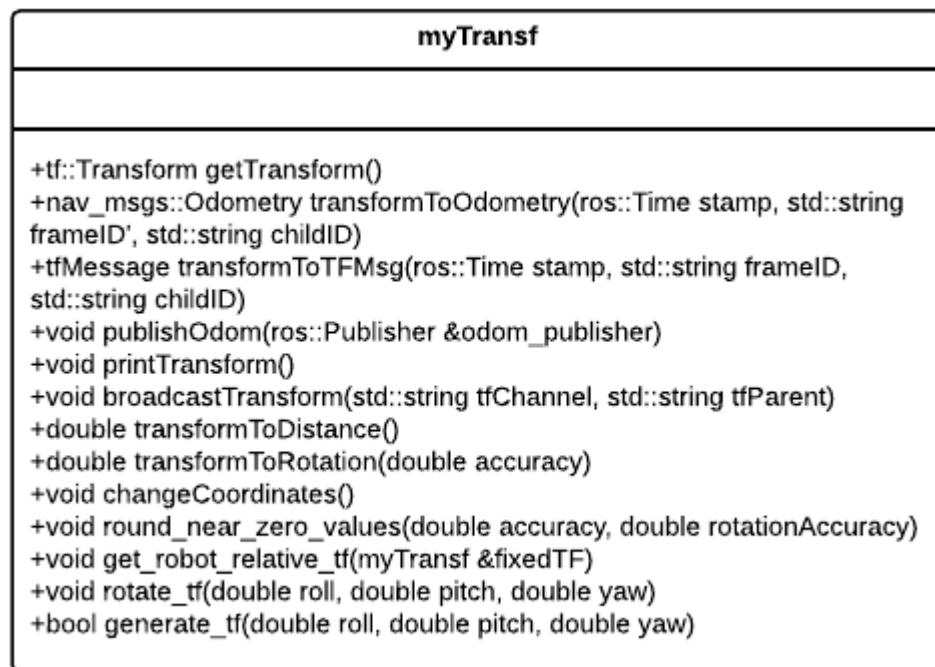


Figura 4.9: Bloque UML de la clase myTransf del nodo odometría

4.5. Estructura física de la base de datos y/o de los ficheros

Dado que el sistema es meramente de procesamiento, y las estructuras de datos utilizadas para la implementación son, en su totalidad, en tiempo de ejecución, no se ha requerido el uso de otras estructuras física tales como bases de datos. Únicamente se han utilizado ficheros para permitir el almacenamiento perdurable de los datos de configuración: para el almacenamiento en disco del diccionario de parámetros de la interfaz se sigue la estructura definida por el lenguaje YAML; y las variables de configuración guardadas en el servidor de parámetros son manejadas por el propio entorno ROS tanto en su almacenamiento como en su carga posterior.

Se trata de almacenar la mínima cantidad imprescindible de datos que el software necesita para retomar su funcionamiento con la configuración deseada por el usuario y, sobre todo, de forma que los resultados puedan ser repetidos y las pruebas reproducidas de forma lo más determinística posible gracias al mantenimiento de la configuración. En total se necesitan: un fichero de traducción para la interfaz, uno de configuración para su servidor de parámetros privado y dos posibles (pero no imprescindibles) listados de parámetros para cargar la configuración de ambos nodos directamente en el servidor de parámetros del entorno. Esto se representa en la imagen numerada como 4.10.

La figura 4.11 representa la estructura utilizada en el fichero de traducción YAML, en el que se almacenan una serie de nombres de controladores (referenciados por su dirección de carga dentro del entorno ROS). Cada driver cuenta con una serie de parámetros de manera tal que su “nombre” es el identificador por el que serán conocidos desde la capa superior y traen asociada una “dirección” (su identificador original por parte del driver), su tipo de dato y el canal en el que pueden encontrarse: topic, servidor de parámetros o, potencialmente, un servicio ROS. Esta estructura con forma de árbol deberá guardar todo lo necesario para la comunicación entre aplicaciones y controladores a través de la interfaz.

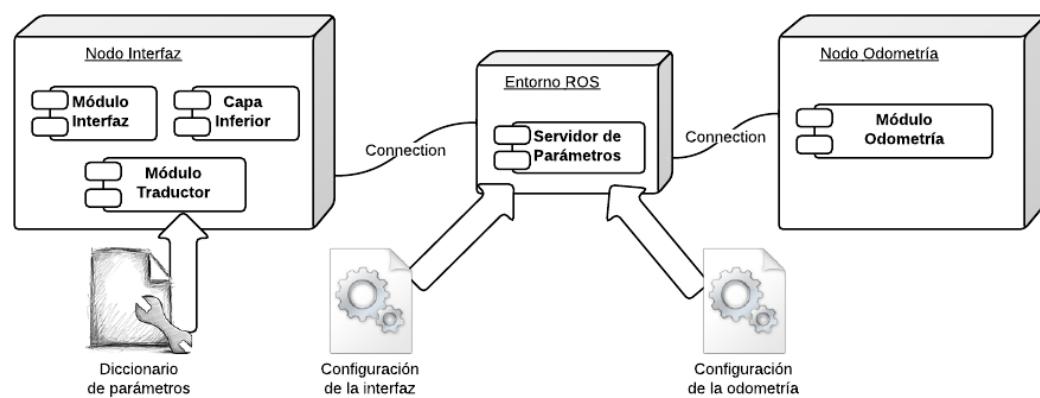


Figura 4.10: Diagrama de la estructura física de datos

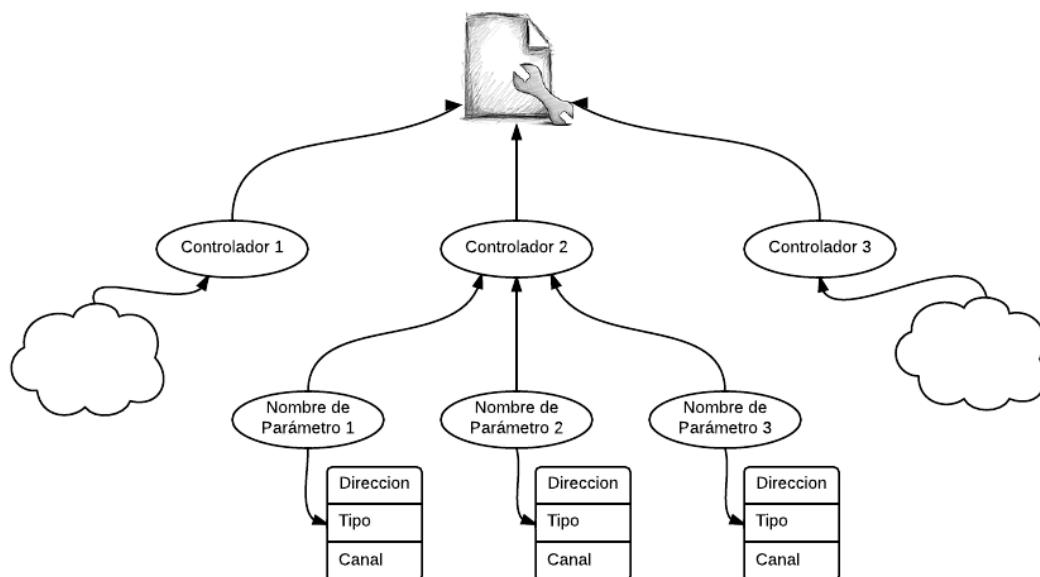


Figura 4.11: Diagrama en árbol de la estructura del diccionario

4.6. Código fuente del producto

El código fuente obtenido como resultado del desarrollo no se incluye en este apartado por motivos de organización y estética debido a su extensión pero está disponible para su consulta dentro del disco que acompaña a la documentación y también en los repositorios del código en los servidores del servicio *GitHub*¹. En su lugar, se puede consultar el *Manual técnico* la documentación de referencia correspondiente a las clases y funciones implementadas, mientras que aquí se explicarán las funciones *main* utilizadas para lanzar correctamente ambos nodos.

4.6.1. Nodo interfaz

El archivo de preparación y lanzamiento de la interfaz, que tal como se planificó utiliza lenguaje Python, tiene la responsabilidad de cargar los parámetros de configuración (debidamente cargados primariamente en el servidor de parámetros de ROS), registrarse a sí mismo como “nodo” de ROS y crear las instancias de los objetos que participan en el proceso. El primer fragmento de código, a continuación, es el punto de entrada “`__main__`” del código invocando la función “`mainFunction`” con el nombre del ejecutable como parámetro.

```
if __name__ == '__main__':
    try:
        basename = sys.argv[0].split('/')[-1]
        if basename == "":
            basename = sys.argv[0].split('/')[-2]
        if basename[-3:] == ".py":
            basename = basename[0: -3]
        mainFunction(basename)
    except Exception as e:
        rospy.logerr("WHOLE_SCOPE_EXCEPTION__NODE_SHUTTING_DOWN")
        rospy.logerr(("Error: %s" % e).center(80, '*'))
    finally:
        rospy.signal_shutdown("testing_shutdown")
```

¹Los repositorios con el código del proyecto se encuentran en las direcciones:

- “https://github.com/Michi05/my_adaptor”
- “https://github.com/Michi05/my_odometry”

La función main secundaria utiliza “init_node” para registrar el proceso como nodo del entorno ROS, carga los parámetros del servidor (o sus valores por omisión, en su defecto) con la función “privateParam” y trata de crear las instancias necesarias para funcionar de tal manera que si fallan debe eliminarlas y comenzar de nuevo.

```

def mainFunction(basename):
    ## Reading from arguments:
    rospy.loginfo("Initializing_node")
    rospy.init_node(basename) # Inicializacion del nodo ROS

    # Carga de los parametros de configuracion:
    globals()["topicTimeOut"] = privateParam(KEY_TOPIC_TIMEOUT, DEFAULT_TOPIC_TIMEOUT)
    globals()["serviceTimeOut"] = privateParam(KEY_SERVICE_TIMEOUT, DEFAULT_SERVICE_TIMEOUT)
    globals()["getStrSrv"] = privateParam(KEY_GET_STRING_SERVICE, DEFAULT_GET_STRING_SERVICE)
        ##... mas lineas similares omitidas ...

    while not rospy.is_shutdown(): # Hasta recibir una senyal de detencion del proceso
        try:
            # Inicializar las 3 capas de comunicacion si es posible
            # Traductor con la configuracion del fichero "propertyConfigFile"
            mainTranslator = iface_translator(globals()["propertyConfigFile"])

            # Manejador de los controladores. Tiene en cuenta los servidores dinamicos
            #cargados anteriormente que se obtienen mediante "dynamicServers()"
            mainDriverManager = driver_manager(dynServers = mainTranslator.dynamicServers())

            # Y ya es posible cargar la capa superior con referencias a los objetos anteriores
            globals()["upwdIface"] = upward_interface(translator = mainTranslator, driverMgr =
                mainDriverManager)

            # Los identificadores del grupo "renaming" son cargados y borrados del diccionario
            rospy.loginfo("Waiting_before_cleaning_renamings.")
            rospy.sleep(5)
            mainTranslator.cleanRenamings(mainDriverManager)

        except KeyboardInterrupt:
            sys.exit(0)
        except Exception as e:
            rospy.logerr("Error_while_trying_to_Initialise_node._Waiting_10s")
            try:
                ## En caso de error se eliminan todas las instancias para comenzar de cero
                del mainTranslator
                del mainDriverManager
                del globals()["upwdIface"]
            except:
                pass
            rospy.sleep(10)
            rospy.loginfo("Reinitializing_Node")
        else:
            ## Si todo funciona la interfaz esta en marcha hasta que una senyal la detenga:
            try:
                rospy.loginfo("...Image_Adaptor_initialized...")
            except KeyboardInterrupt:
                rospy.loginfo("Shutting_down_node.")
            else:
                rospy.spin()
    return

```

4.6.2. Nodo odometría

El paquete de odometría, aunque usa como lenguaje C++ a diferencia de la interfaz, sigue la misma estructura en su código: un punto de entrada que lanza una función de inicialización, con la salvedad de que en este caso la segunda es un método de la clase “odometryComm” que se inicializa a sí mismo y representa el nodo al completo. Una vez más la función main principal envía como parámetro el nombre del ejecutable para que éste dé nombre, a su vez, al nodo. Adicionalmente se crea el manejador de señales que en Python se realizaba de forma implícita.

```
int main(int argc, char **argv) {
    // Lee el nombre del ejecutable para dar nombre al nodo
    std::string nodeName(basename(argv[0]));

    // Carga el manejador de señales con valor "SIGINT"
    signal(SIGINT, signalHandler);

    // Crea el objeto y lo inicializa mediante "init"
    pcl_odometry::odometryComm odoComm;
    if (!odoComm.init(argc, argv, nodeName, ros::init_options::NoSigmoidHandler)) {
        ROS_ERROR("main:_error_initialising_node");
        return 1;
    }
}
```

El método de inicialización, integrado en la propia clase, se encarga de los “cerros” para evitar bloqueos en los manejadores de eventos (“callbacks”), el registro del nodo al igual que en el apartado anterior y cargar y aplicar la configuración plasmada como parámetros en el servidor de parámetros de ROS. Además debe registrar los canales de comunicación que se van a utilizar: topics y servicios.

```
bool odometryComm::init(int argc, char** argv, std::string nodeName, uint32_t
    rosInitOptions) {
    // Cerrojo para determinar cuando la inicialización fue ya completada
    boost::lock_guard<boost::recursive_mutex> isInitialisedLock(
        mIsInitialisedMutex);

    // Si el objeto estaba inicializado se detiene y se vuelve a iniciar
    if (mIsInitialised && !shutdown()) {
        ROS_WARN("odometryComm::init:_ignoring_error_in_shutdown()");
    }

    // Inicializar nodo
    try {
        ros::init(argc, argv, nodeName, rosInitOptions);
    } catch(ros::Invalid.nodeNameException& e) {
        ROS_ERROR("odometryComm::init:_error_initialising_node");
        return false;
    }
    ros::start();
}
```

```

// Carga los parametros de configuracion
try {
    // Manejador del nodo para las operaciones
    ros::NodeHandle nodeHandlePrivate("~");

    // Carga parametros del servidor o, en su defecto, sus valores por omision
    nodeHandlePrivate.param(PARAM_KEY_ICP_ITERATIONS, maxIterations,
                           PARAM_DEFAULT_ICP_ITERATIONS);
    // ...
    // Almacena en el servidor de parametros los valores utilizados finalmente
    nodeHandlePrivate.setParam(PARAM_KEY_ICP_ITERATIONS, maxIterations);
    // ...

    printConfiguration();

} catch (ros::InvalidNameException& e) {
    ROS_ERROR("TeleopSourceNode:: init:_error_initialising_parameters");
    ros::shutdown();
    return false;
}

// Instancia y notifica los topics de publicacion y recepcion
try {
    // Manejador del nodo para las operaciones
    ros::NodeHandle nodeHandlePrivate("~");

    // Crea topics de publicacion con buffer de tamano 1 y "latching" (
        // mantener publicacion)
    int output_queue_size = 1;
    pcl_pub_aligned = nodeHandlePrivate.advertise<PointCloudT> (
        outputAlignedCloud_topic, output_queue_size, true);
    pcl_pub_initial = nodeHandlePrivate.advertise<PointCloudT> (
        outputInitialCloud_topic, output_queue_size, true);
    pcl_pub_final = nodeHandlePrivate.advertise<PointCloudT> (
        outputFilteredCloud_topic, output_queue_size, true);
    odom_answer_publisher = nodeHandlePrivate.advertise<my_odometry::odom_answer> (
        outputOdometryAnswer_topic, output_queue_size, true)
    ;

    // Crea topics de recepcion con buffer de tamano 1
    int input_queue_size = 1;
    if (manualMode==true)
        pcl_sub = nodeHandlePrivate.subscribe<std_msgs::String> (
            inputStrRequest_topic, input_queue_size, &odometryComm::STR_callback, this);
    else
        pcl_sub = nodeHandlePrivate.subscribe<PointCloudT> (
            inputPCL_topic, input_queue_size, &odometryComm::PCL_callback, this);
    camera_tf_sub = nodeHandlePrivate.subscribe<tf::tfMessage> (
        cameraTF_topic, input_queue_size, &odometryComm::cameraTF_callback
        , this);

} catch (ros::InvalidNameException& e) {
    ROS_ERROR("odometryComm:: init:_error_creating_publisher");
    ros::shutdown();
    return false;
}

```

```

ROS_INFO("Trying_to_launch_Visual_Odometry_with_the_next_parameters_for
         _epsilon=%f,_maxIterations=%d_and_maxDistance=%f.\r\n", epsilon,
         maxIterations, maxDistance);

setDownsampleFiltering(leafSize);
setDistanceFiltering(minDepth, maxDepth);

// Actualizar cerrojo para notificar que la inicializacion fue completada
mIsInitialised = true;

// Launch services
ros::NodeHandle nHandle("~");
server_updateOdometry = new ros::ServiceServer(nHandle.advertiseService(
    "updateOdometry", &odometyComm::update_odometry, this));
server_resetGlobals = new ros::ServiceServer(nHandle.advertiseService(
    "resetGlobals", &odometyComm::reset_globals, this));
server_getLastStatus = new ros::ServiceServer(nHandle.advertiseService(
    "getLastStatus", &odometyComm::get_last_status, this));

//Create and start single-threaded asynchronous spinner to handle incoming
//ROS messages via our sole subscriber. Use only one thread, since the
//callback method is not thread-safe.
mSpinner = new ros::AsyncSpinner(1);
mSpinner->start();

// The fixed camera tf needs to be initialized according to rotations
fixed_camera_transform.generate_tf(kinectRoll, kinectPitch, kinectYaw);
fixed_camera_transform.setOrigin(tf::Vector3(kinectX, kinectY, kinectZ));
std::cout << "***initial_camera_fixed_tf:" << std::endl;
fixed_camera_transform.printTransform();

// Initialise status (no error and not-yet waiting for data)
setOdomStatus(INITIALIZED);

ROS_INFO("Completely_Initialized");

//Return result
return true;
}
  
```

Una vez completado el proceso anterior, el nodo de odometría queda registrado en el entorno junto con sus servicios y topics disponibles para comunicarse con el exterior, y se mantiene activo a la espera de los eventos que estos producen: llegada de una nube de puntos, petición de estimación nueva por parte de un usuario, actualización de la TF fija de la cámara, etc.

4.7. Limitaciones de desempeño de la solución

En la solución desarrollada existen una serie de limitaciones físicas, software y de movimiento en relación al tiempo que vienen dadas tanto por los dispositivos hardware que intervienen como el propio entorno de ejecución. Los siguientes márgenes de uso razonable se han ido perfilando durante el desarrollo y tenido en cuenta durante las pruebas, y en adelante dictan en qué casos la solución debe funcionar correctamente y a partir de qué valores o acciones es esperable un posible error.

El paquete interfaz es el intermediario entre aplicaciones de alto nivel y los drivers que utilizan, pero sólo a nivel de “identificadores” o “nombres”, es decir: no es esperable que ésta haga conversiones de tipos datos. Su propósito es transmitir mensajes entre sus dos extremos siempre y cuando el emisor (como la aplicación) sepa a qué clase de recurso van dirigidos y la configuración, en realidad, sólo adapte el nombre de los recursos ya existentes. Además existen las siguientes limitaciones por parte del entorno:

- Incapacidad para *move topics* (modificar su localización) en tiempo de ejecución. La única opción es retransmitirlos (repetirlos) para lo cual se utiliza el multiplexador con el consecuente gasto adicional de recursos y la confusión que supone tener dos topics para lo mismo.
- Uso de servicios no genérico, dado que las conversiones a los tipos de los mensajes no son lo bastante automáticas y, en la práctica, es necesario conocer cada tipo de mensaje asociado a cada servicio (y el nombre de cada campo dentro de él, aunque la estructura de tipos sea idéntica).
- Imposible el reconocimiento de parámetros dependientes. Si un parámetro depende de otro (como la resolución a lo alto y a lo ancho), el usuario debe tener conocimientos suficientes del driver o realizar pruebas empíricas. La solución utilizada por algunos diseñadores ha sido agrupar las configuraciones asociadas en índices de tal manera que cada índice de ‘1’ a ‘n’ corresponde a una combinación.

La cámara de profundidad fue diseñada como dispositivo de juegos, por lo que obedece a una serie de limitaciones que son irrelevantes durante su uso original frente a la consola:

- Rango de sensibilidad a la distancia: de 0.8m a 3.5m (los puntos fuera de esos valores se consideran simplemente inexistentes y se interpretan como “vacío”).
- Falla en superficies demasiado inclinadas o que reflejen mal la luz infrarroja y puede verse cegada puntualmente por reflejos o luz directa del sol.
- Ángulo de visión de 43° en vertical y 57° en horizontal y resolución máxima de 640x480. Estos valores determinan la porción de la escena que es capturada y la cantidad de información que se obtiene de ella.

El algoritmo de odometría es capaz de calcular, a partir de las nubes de puntos capturadas, los movimientos de la cámara que se mantengan dentro de unos límites determinados. Dado que no era objeto de estudio determinar las limitaciones exactas del algoritmo, que dependen de muchas condiciones y posibles optimizaciones para escenarios determinados, no se ha realizado un verdadero estudio empírico completo de este aspecto. Se conocen, sin embargo, tres criterios fundamentales para el éxito de las estimaciones:

- Las nubes empleadas en el cálculo deben tener en común una porción amplia del escenario que el algoritmo pueda utilizar para establecer las correlaciones y calcular el desplazamiento.
- La traslación de cada punto de la escena es tenida en cuenta por el algoritmo para limitar las posibles correspondencias. Cuanta mayor distancia se contempla, mayor la dificultad en los cálculos.
- El escenario debe contener información útil para las medidas de profundidad:
 - Debe haber obstáculos suficientes para qué el robot se oriente, pero también espacio para evitar occlusiones. Como ya se dijo, la cámara sólo recoge información entre 0.8 y 3.5 metros.
 - Los obstáculos deben ser distinguibles entre sí y aportar información:
 - La posición del suelo no varía, por lo que no permite calcular los movimientos.
 - Una pared en curva puede resultar confusa en los giros si no existe otra referencia.
 - Los enrejados provocan “aliasing”, es decir: se confunden las rejillas entre sí al seleccionar las correspondencias.
 - Los escenarios deben ser estáticos. El algoritmo carece de información para discernir si se mueve la cámara o el mundo que la rodea, inclusive si se trata de elementos individuales pues se buscan correspondencias para todos los puntos por igual.

A pesar de la ausencia de estudio empírico que lo corrobore o información suficiente para un estudio teórico, se trata de establecer, a raíz de lo anterior y algunas pruebas adicionales, unos *valores orientativos de partida para las limitaciones*:

En las pruebas de aceptación se consiguieron buenos resultados con movimientos lineales de hasta 30 centímetros y rotaciones de hasta 15 grados, pero utilizando una configuración “mixta” (es decir, válida para movimientos lineales, giros e inclinaciones o combinaciones de lo anterior) es imprescindible limitar más los movimientos. Las recomendaciones en el propio foro de usuarios de *PCL*² refuerzan la idea de que una traslación de 50 centímetros por cada punto resulta un valor alto para el algoritmo.

En aras de establecer unas limitaciones conservadoras para los movimientos de la cámara se tendrán en cuenta los valores ya conocidos como aceptables y se dividirán entre dos para mayor seguridad, es decir: **15 centímetros para los movimientos lineales, 7.5 grados para los giros** y, para los movimientos mixtos, un porcentaje de la escena extrapolado de lo anterior. En un movimiento lineal, el porcentaje de escenario que varía es más dependiente de la distancia a la que se encuentran los objetos que cuando se trata de un giro; por eso se utilizan los giros como referencia. Un giro de 15° con un ángulo de visión horizontal de 57° equivale aproximadamente a un 25 %. Si se asume la extrapolación como válida, sólo un 12.5 % de la escena se considera prescindible, situando en un **87.5 % la cantidad del escenario que tiene que ser común** entre cada par de capturas utilizadas. Una vez más: se trata de cálculos orientativos que no pretenden sustituir a un posible estudio riguroso que no forma parte de los objetivos de la presente documentación.

²En el foro de usuarios de PCL (“wwwpcl-users.org”) se hacen referencias a las limitaciones del algoritmo, como por ejemplo en la pregunta referente a este mismo desarrollo:

“wwwpcl-users.org/Rotated-point-clouds-not-aligning-correctly-td4020648.html”

La velocidad del robot, por su parte, dependerá de la capacidad de cálculo del PC, por lo que su límite debería ser calculado para cada caso particular. Según lo anterior, se deberían realizar al menos 7 estimaciones por metro avanzado (7 estimaciones de 15 cm) y 12 por cada giro de ángulo recto. En conclusión, para que la velocidad del robot calcance su límite físico, se requieren 7 estimaciones cada segundo para alcanzar la velocidad máxima del dispositivo (1 metro por segundo) y 14 para superar su velocidad de giro ($100^{\circ}/s$).

Debe tenerse en cuenta que el algoritmo aún puede ser optimizado y, además, complementado con información de color, acelerómetros y odometría de ruedas, ya que el objetivo era depender exclusivamente de la información tridimensional del escenario. Una vez insertadas dichas optimizaciones se podrá acelerar ampliamente el cálculo de la odometría.

4.8. Especificación del plan de pruebas

En este apartado se exponen las distintas funcionalidades a verificar para garantizar, en la medida de lo posible, que el sistema es completo y correcto. Para ello se enumeran las distintas acciones que debe realizar cada subsistema y los aspectos concretos a examinar de modo que sirva como guía para la creación de los casos de prueba. El propósito es verificar, con las pruebas de aceptación del sistema, que la implementación responde a lo esperado en el diseño y que, bajo las condiciones adecuadas, se obtienen las respuestas esperadas.

Existen dos tipos de pruebas que se pueden realizar sobre el proyecto según las metodologías más conocidas: las pruebas de caja blanca y las pruebas de caja negra. Por desgracia, las primeras consumen mucho tiempo en favor de comprobaciones más precisas y completas (se verifica directamente el código para evitar que los errores pasen desapercibidos ante un escenario determinado). Para ajustarse al tiempo planificado para el proyecto, se explicarán ambos métodos, se pasará directamente a las pruebas de caja negra sobre las funcionalidades específicas.

- **Pruebas de caja Blanca.-** Al realizar pruebas de caja blanca se prueban individual e independientemente los distintos métodos o funciones que existen en el código a nivel de programación. Algunas técnicas consisten en analizar la cobertura de caminos, decisiones o sentencias del código, es decir, comprobar la coherencia de éste al pasar por distintos grupos de órdenes hasta probarlas todas. En cualquier caso el objetivo es garantizar que todas las líneas del código responden a la lógica que se espera según la descripción del método o función.
- **Pruebas de Caja Negra.-** Las pruebas de caja negra consisten en realizar peticiones básicas al sistema en función de las distintas funciones, funcionalidades o acciones que éste ofrece para comprobar si realizan lo que se espera de ellos o pueden causar problemas. En definitiva se trata de comprobar que las acciones disponibles para el usuario según las especificaciones son realmente funcionales y ofrecen los resultados adecuados. Se pueden utilizar las “clases de equivalencia” (rangos de valores que, hipotéticamente, deberían ser tratados de modo similar internamente) como criterio para planificar los valores de entrada para las pruebas.

4.8.1. Pruebas de requisitos funcionales de la interfaz

En este apartado se detallan las distintas funcionalidades que se deben verificar para garantizar, en la medida de lo posible, que el nodo interfaz sea correcto. Una vez enumeradas las distintas acciones que deberán estar disponibles, ésto servirá como guía para la creación de los casos de prueba.

Leer valor actual de parámetro

La prueba más básica que se puede realizar sobre la interfaz es la capacidad para recuperar un parámetro desde el controlador para conocer su valor de configuración actual. Consiste en introducir el nombre (a nivel de aplicación) del parámetro y esperar el valor correspondiente.

Modificar valor de parámetro

Además de obtener los valores actuales de configuración debe ser posible modificarlos para adaptar el comportamiento del driver al deseado por el agente externo que lo solicite. En ese caso se introduciría el nombre del parámetro a nivel de aplicación y un valor del tipo adecuado para sustituir al actual.

Obtener ubicación de un topic

La ubicación real de los topics que contienen recursos del controlador debe ser recuperable en caso de necesidad para poder realizar una conexión directa desde la capa superior.

Reubicar Topic

También debe ser posible modificar la ubicación de un topic determinado, de tal manera que pueda encontrarse en una nueva dirección si fuera necesario.

Notificación de cambio en los parámetros

Se espera que el nodo interfaz sea capaz de lanzar una notificación si hubiera nuevos cambios en el driver para informar a las capas superiores (en particular a otros nodos) de que la configuración ha variado.

4.8.2. Pruebas de requisitos funcionales de la odometría

Debe realizarse pruebas exhaustivas que garanticen también el comportamiento del nodo de odometría y que éste cumple con cierta coherencia tanto al obedecer a las órdenes como en los valores obtenidos como resultado. Se enumerarán primero las funcionalidades esperadas para construir, más tarde, sus casos de prueba correspondientes.

Obtener estimación de odometría

Se deberá someter la aplicación a la prueba de calcular la odometría entre dos escenas y publicar un resultado coherente para el usuario o las aplicaciones que lo requieren. Es importante destacar que no se trata de una prueba precisa pero sí clara, es decir, que los valores pueden ser inexactos pero dentro de un límite que se especificará.

Reiniciar medidas. Puesta a cero

Al recibir una petición de puesta a cero, todos los valores de odometría deben quedar en la posición (0, 0, 0) y orientación (0, 0, 0).

Asignar nueva posición y orientación de la cámara

Son también necesarias las pruebas con la posición y orientación de la cámara. Estos valores deben ser configurables para tenerlos en cuenta al calcular el movimiento del robot que es relativo al de la cámara.

Publicar las últimas estimaciones conocidas

Como parte del comportamiento esperado, el nodo de odometría debe publicar los resultados de cada cálculo realizado ya sea a petición de un agente externo o de forma automática al recibir una escena nueva.

4.8.3. Casos de prueba del subsistema interfaz

Una vez definidas las funcionalidades a probar, tal como se adelantaba, es posible establecer los casos de prueba y llevar a cabo las pruebas de aceptación correspondientes. Primeramente se presenta una tabla con las funcionalidades que se probarán especificando las acciones por parte del usuario y las reacciones que se esperan como salida por parte del sistema. Después se pasará a realizar las pruebas de facto anotando todos los resultados correspondientes para, finalmente, poder analizar los comportamientos observados y, en caso de encontrar errores, tratar de discernir el motivo de los mismos y una posible solución.

Tabla de casos de prueba

Funcionalidad	Acción Iniciadora	Resultado Esperado
Leer parámetro (A)	Ejecutar servicio “get/<tipo>Param” ³ a través de la línea de comando	El valor del parámetro será mostrado como respuesta
Leer parámetro (B)	Consulta de parámetros mediante la herramienta “reconfigure_gui”	El parámetro debe figurar en la lista asociado a un valor
Modificar parámetro (A)	Ejecutar servicio “set/<tipo>Param” ² a través de la línea de comando	El parámetro debe actualizarse con el nuevo valor pasado a través del servicio
Modificar parámetro (B)	Modificación del parámetro mediante la herramienta “reconfigure_gui”	El parámetro debe actualizarse al modificar su valor a través de la herramienta
Obtener ubicación de topic	Ejecutar servicio “get/TopicLocation” a través de la línea de comando	La dirección del topic será mostrada como respuesta
Reubicar Topic	Ejecutar servicio “set/TopicLocation” a través de la línea de comando	El topic deberá quedar duplicado en la nueva dirección y un mensaje de respuesta confirmará la acción
Notificación de cambio en parámetros	El controlador modifica su configuración dinámica y la interfaz lo detecta como un evento	La interfaz actualiza sus parámetros y desencadena un evento del cambio para los nodos que estén a la escucha

Tabla 4.1: Tabla de pruebas de caja negra del subsistema Interfaz

³Existen varios servicios para leer y modificar los parámetros en función de su tipo. Debe sustituirse “<tipo>” por el tipo correspondiente en inglés para acceder al servicio según el caso.

Casos de prueba de la interfaz

A continuación, se especifican todos los casos de prueba individualmente junto con la entrada utilizada y el resultado obtenido para cada una.

Funcionalidad: Leer valor actual de parámetro (opción A)
Caso de prueba: P 1.1
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ La interfaz debe ser ejecutada con la configuración por defecto y mediante el lanzador <code>kinectDrivers.launch</code> : « <code>roslaunch my_adaptor kinectDrivers.launch</code> ». ■ El usuario solicita la lectura de un parámetro, en este caso “<code>depth_resolution</code>”: « <code>rosservice call get/IntParam depth_resolution</code> ». ■ El valor resultante puede ser leído en la respuesta del comando.
Salida esperada: El valor del parámetro debe ser ‘0’ y aparecer por pantalla como respuesta: “ <code>paramValue: 2</code> ”.
Resultado obtenido: El servicio responde en la siguiente línea con el texto “ <code>paramValue: 2</code> ”.

Tabla 4.2: Resultado de la prueba sobre *Leer valor actual de parámetro (opción A)*

Funcionalidad: Leer valor actual de parámetro (opción B)
Caso de prueba: P 1.2
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ La interfaz debe ser ejecutada con la configuración por defecto y mediante el lanzador <code>kinectDrivers.launch</code> : « <code>roslaunch my_adaptor kinectDrivers.launch</code> ». ■ Se lanza la herramienta de reconfiguración de parámetros de ROS mediante el comando « <code>rosrun dynamic_reconfigure reconfigure_gui</code> ». ■ Ha de elegirse el nodo correspondiente a la interfaz dentro de la herramienta externa. ■ El usuario puede buscar la línea con el nombre del parámetro, en este caso “<code>depth_resolution</code>”, y leer el valor correspondiente.
Salida esperada: El parámetro debe figurar en el listado mostrado por la herramienta con el valor ‘2’ a su derecha.
Resultado obtenido: El parámetro figura en el listado y muestra el valor ‘2’ a su derecha.

Tabla 4.3: Resultado de la prueba sobre *Leer valor actual de parámetro (opción B)*

Funcionalidad: Modificar valor de parámetro (opción A)
Caso de prueba: P 1.3
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ La interfaz debe ser ejecutada con la configuración por defecto y mediante el lanzador <code>kinectDrivers.launch</code> : « <code>roslaunch my_adaptor kinectDrivers.launch</code> ». ■ El usuario solicita la modificación del parámetro “depth_resolution” mediante llamada al servicio correspondiente con el valor deseado: « <code>rosservice call set/IntParam depth_registration 1</code> ». ■ El sistema responderá con una confirmación de la modificación.
Salida esperada: El nuevo valor del parámetro debe quedar fijado y aparecer por pantalla como respuesta: “setAnswer: 1”.
Resultado obtenido:

Tabla 4.4: Resultado de la prueba sobre *Modificar valor de parámetro (opción A)*

Funcionalidad: Modificar valor de parámetro (opción B)
Caso de prueba: P 1.4
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ La interfaz debe ser ejecutada con la configuración por defecto y mediante el lanzador <code>kinectDrivers.launch</code> : « <code>roslaunch my_adaptor kinectDrivers.launch</code> ». ■ Se lanza la herramienta de reconfiguración de parámetros de ROS mediante el comando « <code>rosrun dynamic_reconfigure reconfigure_gui</code> ». ■ Ha de elegirse el nodo correspondiente a la interfaz dentro de la herramienta externa. ■ El usuario puede buscar la línea con el nombre del parámetro, “depth_resolution” y escribir el valor deseado: 1.
Salida esperada: Al lado del parámetro “depth_resolution” debe figurar el nuevo valor insertado: ’1’. En la configuración de la interfaz debe quedar grabado dicho valor asociado al mismo.
Resultado obtenido: El parámetro figura en el listado y muestra el valor ’1’ a su derecha.

Tabla 4.5: Resultado de la prueba sobre *Modificar valor de parámetro (opción B)*

Funcionalidad: Obtener ubicación de topic
Caso de prueba: P 1.5
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ La interfaz debe ser ejecutada con la configuración por defecto y mediante el lanzador <code>kinectDrivers.launch</code> : « <code>roslaunch my_adaptor kinectDrivers.launch</code> ». ■ El usuario solicita la ubicación actual de un topic (en esta prueba “depth_reg_raw”) mediante llamada al servicio correspondiente con el valor deseado: « <code>rosservice call /kinect1/get/TopicLocation depth_reg_raw</code> ». ■ El sistema enviará el valor deseado como respuesta.
Salida esperada: La ubicación o dirección del topic correspondiente dentro del entorno ROS debe ser mostrada en pantalla como respuesta al servicio. En este caso: “depth_registered/image_rect_raw”.
Resultado obtenido: El servicio responde con la cadena de texto esperada: “paramValue: depth_registered/image_rect_raw”.

Tabla 4.6: Resultado de la prueba sobre *Obtener ubicación de topic*

Funcionalidad: Reubicar topic
Caso de prueba: P 1.6
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ La interfaz debe ser ejecutada con la configuración por defecto y mediante el lanzador <code>kinectDrivers.launch</code> : « <code>roslaunch my_adaptor kinectDrivers.launch</code> ». ■ El usuario solicita la reubicación de un topic escribiendo el parámetro al que se refiere y el destino deseado junto con la llamada al servicio correspondiente: « <code>rosservice call set/TopicLocation depth_reg_raw my_rdepth</code> ». ■ El sistema responderá con una confirmación de la modificación.
Salida esperada: El topic debe quedar reubicado en la dirección indicada y el servicio responderá con la cadena “success”.
Resultado obtenido: En la línea de comando se recibe como respuesta un mensaje “Success”. El nuevo topic aparece correctamente en el listado del entorno al escribir « <code>rostopic list grep rdepth</code> »

Tabla 4.7: Resultado de la prueba sobre *Reubicar topic*

Funcionalidad: Notificación de cambio en parámetros

Caso de prueba: P 1.7

Descripción de los valores a introducir:

- La interfaz debe ser ejecutada con la configuración por defecto y mediante el lanzador *kinectDrivers.launch*:
`« roslaunch my_adaptor kinectDrivers.launch ».`
- Se lanza un nodo que permita suscribirse a las actualizaciones, como por ejemplo la herramienta de “dynamic_reconfigure”:
`« rosrun dynamic_reconfigure reconfigure_gui ».`
- Se modifica un valor externamente mediante un servicio. Sea en este caso el parámetro “depth_resolution”:
`« rosservice call set/IntParam depth_resolution 1 ».`
- Se observa si hubo cambios en el listado de parámetros de “dynamic_reconfigure”. En tal caso significará que recibió la notificación.

Salida esperada: La interfaz debe lanzar una notificación que desencadene la actualización del valor en “dynamic_reconfigure” de 2 a 1.

Resultado obtenido: El valor asociado al parámetro “depth_resolution” en el listado de la herramienta pasa de 2 a 1.

Tabla 4.8: Resultado de la prueba sobre *Notificación de cambio en parámetros*

4.8.4. Casos de prueba del subsistema de odometría

De nuevo con las funcionalidades ya definidas es posible establecer los casos de prueba y llevar a cabo las pruebas de aceptación correspondientes. La tabla de funcionalidades abre el presente apartado especificando las acciones por parte del usuario y las reacciones que se esperan como salida por parte del sistema. Después se pasará a realizar las pruebas de facto anotando todos los resultados correspondientes para, finalmente, poder analizar los comportamientos observados y, en caso de encontrar errores, tratar de discernir el motivo de los mismos y una posible solución.

En este caso, a diferencia de la interfaz, existe un comportamiento “físico” y unos resultados que también deben ser evaluados. Por ese motivo se divide la sección en dos partes: las pruebas de aceptación de funcionalidades y las pruebas de aceptación para los resultados del algoritmo.

Tabla de casos de prueba de funcionalidades

Funcionalidad	Acción Iniciadora	Resultado Esperado
Obtener estimación de odometría ⁴	Ejecutar servicio “updateOdometry”	El nodo solicita una nube de puntos, la compara con la anterior, actualiza sus datos y responde con los datos de la estimación
Reiniciar medidas	Ejecutar servicio “resetGlobals”	Los valores de odometría acumulados y la posición de la cámara vuelven a sus valores iniciales y se borra la nube de puntos previa almacenada
Asignar posición de la cámara	Publicar una TF en el topic correspondiente a la posición de la cámara	El nodo recibe la TF y la almacena para aplicarla en los cálculos posteriores
Publicar últimas estimaciones	El nodo obtiene una nube de puntos nueva (ya sea de forma automática o manual)	Al término de la estimación, los valores de odometría son publicados en los topics habilitados a tal efecto

Tabla 4.9: Tabla de pruebas de caja negra del subsistema de odometría (funcionalidades)

⁴Prueba de aceptación de la función que permite obtener la estimación. Los valores obtenidos se testarán en la segunda parte.

Casos de prueba de funcionalidades

Las siguientes figuras representan los casos de prueba realizados incluyendo los valores introducidos y las respuestas obtenidas para cada prueba individual.

Funcionalidad: Obtener estimación de odometría
Caso de prueba: P 2.1
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ El usuario solicita la estimación nueva mediante llamada al servicio “updateOdometry”: « rosservice call updateOdometry ». ■ El sistema enviará en respuesta toda la información conocida sobre la odometría en el instante actual: última estimación de movimiento, posición acumulada y estado del algoritmo.
Salida esperada: El nodo debe capturar una nube de puntos nueva, notificarlo por pantalla y responder al servicio con la estimación obtenida.
Resultado obtenido: En la pantalla de ejecución del nodo se muestra el mensaje confirmando la nueva nube de puntos recibida y, tras mostrar también la estimación, ésta es enviada también a través del servicio como respuesta.

Tabla 4.10: Resultado de la prueba sobre *Obtener estimación de odometría*

Funcionalidad: Reiniciar medidas. Puesta a cero
Caso de prueba: P 2.2
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ El usuario ejecuta el servicio “resetGlobals” para reiniciar todos los cálculos: « rosservice call resetGlobals ». ■ El sistema, tras realizar la tarea, responderá con los datos resultantes tras el reinicio; posición inicial, sin movimiento previo y estado a la espera.
Salida esperada: La estimación global con la posición del robot debe volver a sus valores iniciales y ser enviada en respuesta al servicio para su comprobación.
Resultado obtenido: El servicio recibe como respuesta una estimación de odometría “nula” con movimiento nulo y posición acumulada (0, 0, 0).

Tabla 4.11: Resultado de la prueba sobre *Reiniciar medidas. Puesta a cero*

Funcionalidad: Asignar nueva posición y orientación de la cámara
Caso de prueba: P 2.3
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ Primeramente el parámetro “topic_camera_tf” permite averiguar qué topic tiene asignada la función de recibir la TF de la cámara: « rosparam get topic_camera_tf ». ■ Seguidamente se puede editar el fichero “camera_fixed_tf.yaml” para especificar los valores de translación y rotación en sus respectivos campos correspondientes. ■ La ejecución del comando « rostopic pub --once -f camera_fixed_tf.yaml /odometry_main/input_camera_tf tf/tfMessage » enviará los valores al nodo.
Salida esperada: El nodo debería recibir y almacenar la nueva TF de la cámara, informar por pantalla y utilizarla en sus estimaciones en adelante.
Resultado obtenido: Al enviar la TF de la cámara se imprime el mensaje “nueva <i>camera_fixed_tf</i> ” seguido de los valores nuevos de la misma.

Tabla 4.12: Resultado de la prueba sobre *Asignar nueva posición y orientación de la cámara*

Funcionalidad: Publicar las últimas estimaciones conocidas
Caso de prueba: P 2.4
Descripción de los valores a introducir:
<ul style="list-style-type: none"> ■ El nodo de odometría se encuentra a la escucha en modo automático. ■ Un nodo externo que hace las veces de “fuente” publica una nube de puntos. ■ La llegada de una nueva nube desencadena el cálculo de la estimación de odometría. ■ Una vez calculada la TF entre las dos últimas nubes, ésta es publicada mediante el topic ROS correspondiente. ■ El comando « rostopic echo /odometry_main/odometry_answer » permite observar si se están publicando las estimaciones esperadas.
Salida esperada: Se deberá publicar cada estimación obtenida después de cada nube de puntos adquirida y acompañada de la estimación global de la posición.
Resultado obtenido: Al ejecutar el comando “rostopic echo” se observan los mensajes publicados en el topic de odometría que corresponden a sucesivas estimaciones de movimiento entre las nubes de puntos.

Tabla 4.13: Resultado de la prueba sobre *Publicar las últimas estimaciones conocidas*

Tabla de casos de prueba de odometría visual

Para verificar que se realiza correctamente el cálculo de la odometría se establecen una serie de movimientos básicos que deberán ser “interpretados” por el algoritmo. El escenario, visible en la figura 4.12, lo forman una silla con el respaldo en la parte más cercana, dos paredes y una caja de almacenamiento estándar; a la derecha una puerta abierta que llega a verse en algunas escenas. El objetivo es disponer de un plano que permita medir fácilmente el acierto de las rotaciones, un prisma rectangular que sea fácilmente identificable por el algoritmo gracias a su forma simple, y una silla para verificar que el algoritmo no se equivoque con elementos más complejos. Se organizaron los elementos de tal forma que la nube de puntos tenga cierta profundidad y puntos con información repartidos a lo alto y ancho de la misma.

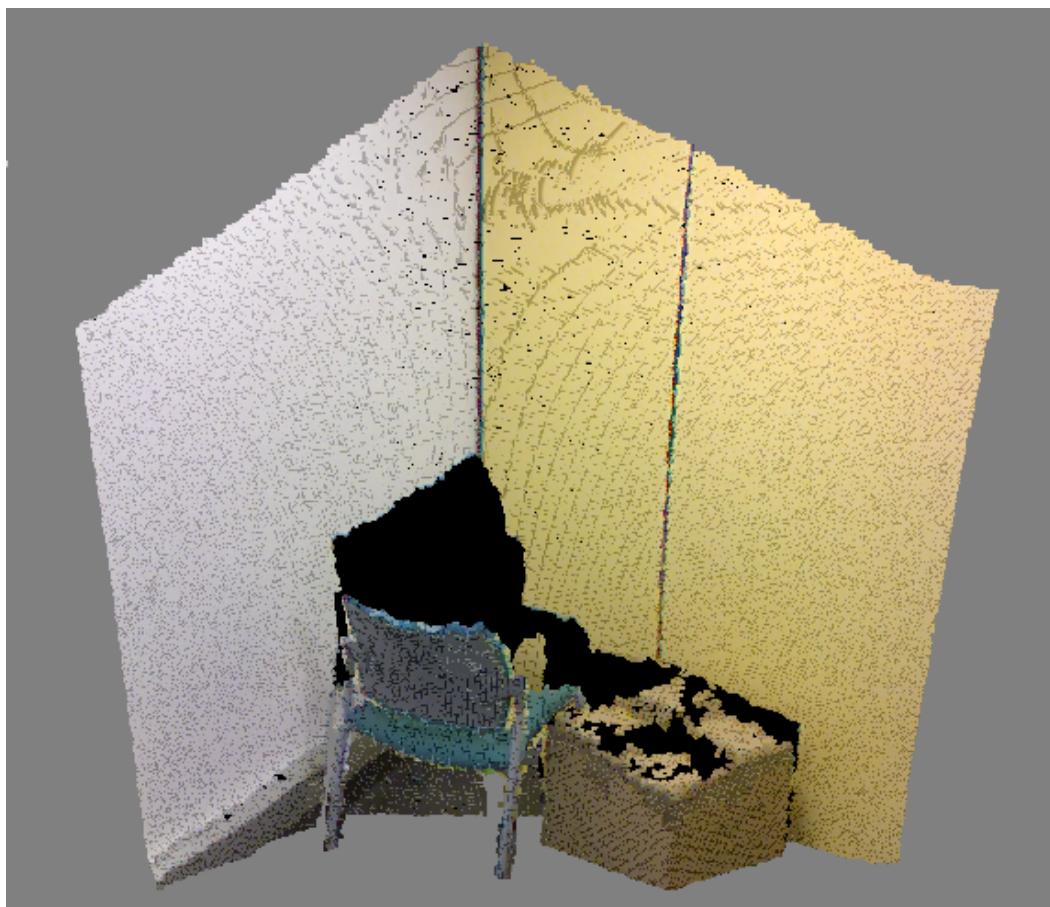


Figura 4.12: Nube de puntos a color correspondiente al escenario de las pruebas

La tabla con las pruebas propuestas se encuentra justo a continuación bajo la numeración 4.14, especificando la naturaleza de cada prueba y el resultado esperado en cada caso. Todo ello se ampliará en cada caso de prueba correspondiente.

³Prueba de aceptación de la función que permite obtener la estimación. Los valores obtenidos se testarán en la segunda parte.

Prueba	Resultado Esperado
Movimiento lineal hacia delante de 20 cm	La estimación debe presentar un valor de +0.2 (metros) en el eje x
Movimiento lineal hacia atrás de 20 cm	La estimación debe presentar un valor de -0.2 (metros) en el eje x
Movimiento lineal lateral hacia la izquierda de 20 cm	La estimación debe presentar un valor de +0.2 (metros) en el eje y
Movimiento lineal lateral hacia la derecha de 20 cm	La estimación debe presentar un valor de -0.2 (metros) en el eje y
Movimiento diagonal de 20 cm al frente y hacia la derecha	La estimación debe presentar un desplazamiento de unos 0.2 metros con un valor positivo de "x" y negativo de "y"
Movimiento diagonal de 20 cm hacia la izquierda y atrás	La estimación debe presentar un valor de unos 0.2 metros con un valor negativo de "x" y positivo de "y"
Rotación de 28.5 grados hacia la izquierda	La estimación debe presentar un valor de +0.5 radianes en Y (Yaw)
Rotación de 28.5 grados hacia la derecha	La estimación debe presentar un valor de -0.5 radianes en Y (Yaw)
Rotación de 28.5 grados hacia la izquierda con cabeceo de 12 grados en sentido positivo	La estimación debe presentar un valor de +0.5 radianes en Y (Yaw)
Rotación de 28.5 grados hacia la derecha con cabeceo de 12 grados en sentido positivo	La estimación debe presentar un valor de -0.5 radianes en Y (Yaw)
Cabeceo de 22.5 grados en sentido positivo	La estimación debe presentar un valor de 0.39 radianes en P (Pitch)
Cabeceo de 22.5 grados en sentido negativo	La estimación debe presentar un valor de -0.39 radianes en P (Pitch)

Tabla 4.14: Tabla de pruebas de caja negra de la odometría visual

Casos de prueba de odometría visual

Para ilustrar gráficamente los datos obtenidos se muestran las nubes de puntos obtenidas en cada caso. En este caso se ha elegido el blanco como color básico para la captura inicial, de forma que el verde pueda representar la situación final (es decir, antes y después de cada movimiento). La nube de puntos roja, por su parte, representa el movimiento **estimado**, que idealmente deberá coincidir con los puntos de color verde si la estimación es completamente correcta.

Cabe aclarar que el convenio que se utiliza para los sistemas de referencia en este apartado se corresponde con el de ROS que es lo bastante generalizado en el mundo de la robótica y que se explica en el apartado correspondiente dentro de la *Documentación auxiliar* de la *Memoria* (1.7.2). En resumen se utiliza la regla de la mano derecha para el sentido de los giros y el convenio del mismo nombre para los ejes de coordenadas.

Movimientos en el eje X de la cámara

Primeramente debe comprobarse el funcionamiento de la odometría en desplazamientos básicos en X o, lo que es lo mismo, avanzando y retrocediendo en línea recta. Para ello se realizan pruebas en dos escenarios distintos y se comparan los resultados obtenidos con las medidas físicas conocidas sobre el movimiento realizado. La precisión de los valores está afectada por algunos errores de medida de la cámara, por errores del movimiento físicamente (al estar hecho de forma manual) y la estimación del algoritmo en sí misma. Con todo, no se considera correcto un error mayor que 2 centímetros.

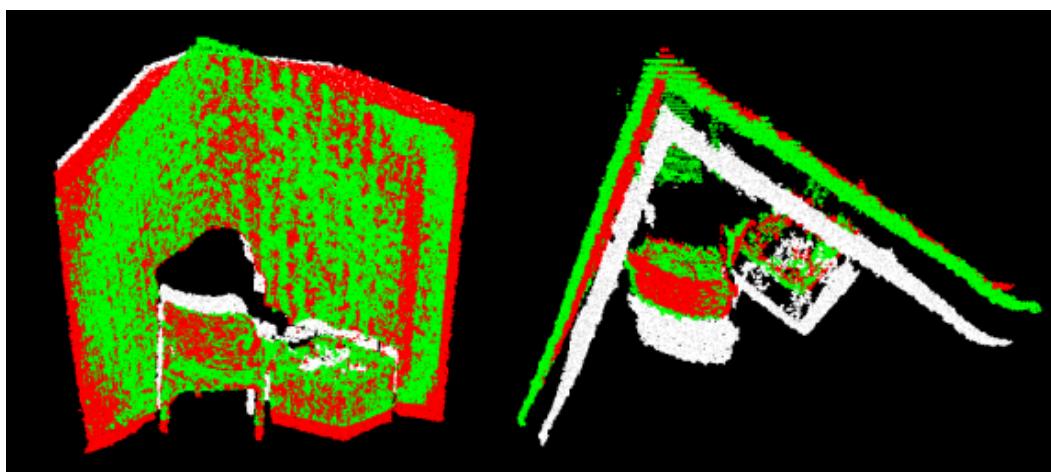


Figura 4.13: Ajuste de las nubes de puntos para las dos primeras pruebas visto desde el frente y desde arriba

1. Movimiento lineal hacia delante de 20 cm.

Funcionalidad: Estimación de +20 cm en x
Caso de prueba: P3.1
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas a una distancia de 20 cm entre sí sin cambiar su orientación y de tal modo que la primera se haya tomado 20 centímetros más al frente que la segunda sobre su mismo eje 'x', sin desplazamientos laterales (sobre 'y').
Salida esperada: Debe obtenerse una TF describiendo un desplazamiento de aproximadamente 0.2 metros hacia adelante sin rotaciones:
$(x, \ y, \ z) = \begin{matrix} +0.2 \\ 0 \end{matrix}$ $(R, \ P, \ Y) = \begin{matrix} 0 \\ 0 \end{matrix}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) ubica la nueva posición 20.05 cm más allá que su valor anterior.
Puntuación de ajuste: 0.000343417 con 60995 puntos.
$(x, \ y, \ z) = \begin{matrix} 0.204529 \\ 0.0105554 \\ -1.98059e-06 \end{matrix}$ $(R, \ P, \ Y) = \begin{matrix} -0.0145104 \\ 0.000207654 \\ -0.0207915 \end{matrix}$

Tabla 4.15: Resultado de la prueba sobre *Estimación de +20 cm en x*

El resultado numérico muestra una ligera rotación y movimientos de hasta 1.5cm en altura que pueden explicarse por el soporte de la cámara que no es completamente estable y puede tener pequeñas inclinaciones. El valor de “x = 0.205” es, en todo caso, suficientemente cercano al esperado y se considera la prueba exitosa.

En la figura 4.13, imagen de la izquierda, se muestran las capturas de la cámara (en blanco y verde según su orden de adquisición) y el movimiento estimado (en rojo), que coincide exactamente con la nube verde. Es la confirmación visual de que la estimación es válida

2. Movimiento lineal hacia atrás de 20 cm.

Funcionalidad: Estimación de -20 cm en x
Caso de prueba: P3.2
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas a una distancia de 20 cm entre sí sin cambiar su orientación y de tal modo que la primera se haya tomado 20 centímetros más atrás que la segunda sobre su mismo eje 'x', sin desplazamientos laterales (sobre 'y').
Salida esperada: Debe obtenerse una TF describiendo un desplazamiento de aproximadamente 0.2 metros hacia atrás sin rotaciones:
$(x, y, z) = -0.2 \quad 0 \quad 0$ $(R, P, Y) = 0 \quad 0 \quad 0$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) ubica la nueva posición aproximadamente 20 cm atrás de su valor anterior.
Puntuación de ajuste: 7.58128e-05 con 58227 puntos.
$(x, y, z) = -0.207547 \quad 0 \quad 0.0184336$ $(R, P, Y) = 0 \quad 0 \quad 0$

Tabla 4.16: Resultado de la prueba sobre *Estimación de -20 cm en x*

En este caso las rotaciones son exactamente cero, no por ausencia de ruido en las medidas, sino porque estas son filtradas al no llegar al mínimo especificado en la configuración. Vuelve a haber un error, ya en el límite de lo aceptable, de 1.8cm en la altura, pero con “x = -0.208” se acepta definitivamente la prueba como válida. El ajuste de la estimación se puede observar en la imagen derecha de la figura 4.13.

Movimientos en el eje 'y' de la cámara

Los robots modelo “Amigobot” utilizados en el proyecto, no tienen capacidad de desplazamiento lateral, sin embargo es muy probable que con distintas configuraciones físicas de la cámara (en función de cómo se instale sobre el robot) existan movimientos laterales compuestos con giros. En todo caso es un movimiento básico que se probará a continuación de modo que quede ya documentado.

1. Movimiento lineal lateral hacia la izquierda de 20 cm.

La cámara se desplaza 20 cm lateralmente en línea recta para obtener dos capturas sobre el mismo eje “y” de su propio sistema de referencia. En la primera prueba el movimiento es positivo, es decir, hacia la izquierda.

Funcionalidad:	Estimación de +20 cm en 'y'					
Caso de prueba:	P3.3					
Descripción de los valores a introducir:	Se introducen dos nubes de puntos tomadas a una distancia de 20 cm entre sí sin cambiar su orientación y de tal modo que la primera se haya tomado la derecha de la segunda y prolongando su mismo eje 'y'					
Salida esperada:	Debe obtenerse una TF describiendo un desplazamiento de aproximadamente 0.2 metros a la izquierda sin rotaciones:					
$(x, \ y, \ z) = \begin{matrix} 0 \\ 0 \end{matrix}$ $(R, \ P, \ Y) = \begin{matrix} 0.2 \\ 0 \\ 0 \end{matrix}$						
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) ubica la nueva posición aproximadamente 20 cm a la izquierda de su valor anterior.						
Puntuación de ajuste: 0.000669356 con 61192 puntos.						
$(x, \ y, \ z) = \begin{matrix} 0 \\ 0 \end{matrix}$ $(R, \ P, \ Y) = \begin{matrix} 0.209855 \\ 0 \\ 0 \end{matrix}$						

Tabla 4.17: Resultado de la prueba sobre *Estimación de +20 cm en 'y'*

El resultado de la prueba es de unos 21 centímetros de los 20 esperados, lo cual es un resultado positivo siempre teniendo en cuenta los objetivos del desarrollo que no son, en este caso, optimizar los resultados ni el tiempo para obtenerlos.

2. Movimiento lineal lateral hacia la derecha de 20 cm.

Se realiza un desplazamiento lateral de -20 cm lateralmente en línea recta. Una vez más se obtienen dos capturas sobre el mismo eje "y" de la cámara, pero esta vez hacia la derecha; se trata de un movimiento en sentido negativo.

Funcionalidad: Estimación de -20 cm en y
Caso de prueba: P3.4
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas a una distancia de 20 cm entre sí sin cambiar su orientación y de tal modo que la primera se haya tomado la izquierda de la segunda y prolongando su mismo eje 'y'
Salida esperada: Debe obtenerse una TF describiendo un desplazamiento de aproximadamente 0.2 metros a la derecha sin rotaciones:
$(x, \ y, \ z) = \begin{matrix} 0 \\ 0 \end{matrix}$ $(R, \ P, \ Y) = \begin{matrix} -0.2 \\ 0 \\ 0 \end{matrix}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) ubica la nueva posición aproximadamente 20 cm a la derecha de su valor anterior.
Puntuación de ajuste: 0.000750632 con 60559 puntos.
$(x, \ y, \ z) = \begin{matrix} 0 \\ 0 \end{math>$

Tabla 4.18: Resultado de la prueba sobre *Estimación de -20 cm en y*

Un resultado mejor que en el caso anterior que permite confirmar el buen comportamiento del algoritmo en los desplazamientos laterales. Sólo un desvío de 0.4 centímetros en este caso. La figura 4.14 muestra el ajuste visual de ambas pruebas de desplazamiento lateral, permitiendo percibir el movimiento y su estimación de forma más intuitiva.

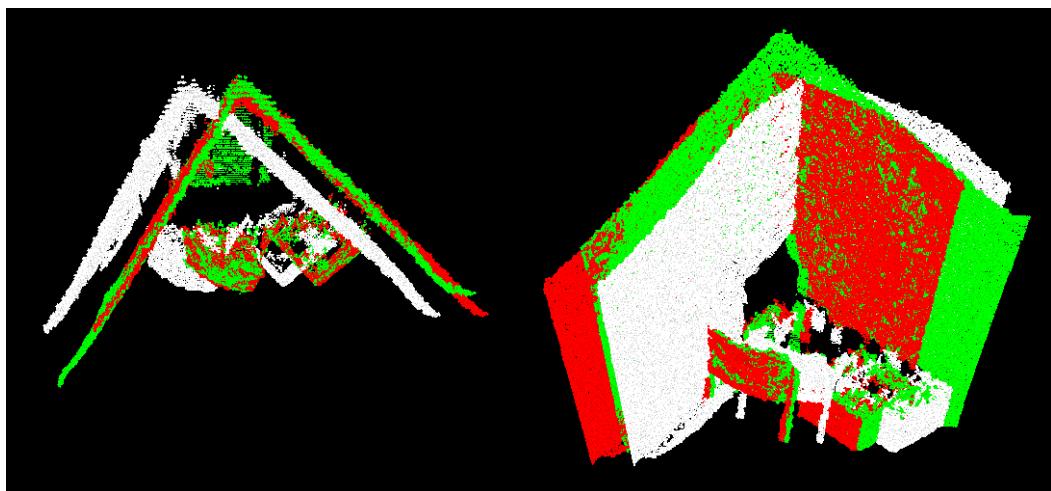


Figura 4.14: Pruebas de movimiento lateral. $y = +20$ cm en planta - $y = -20$ cm en perspectiva.

Movimientos en Diagonal

Como resultado de combinar los movimientos laterales y de avance de la cámara, se producen movimientos en diagonal cuyas pruebas, aunque son imprescindibles, no se descartaron dado que pueden aportar información adicional sobre el comportamiento del algoritmo.

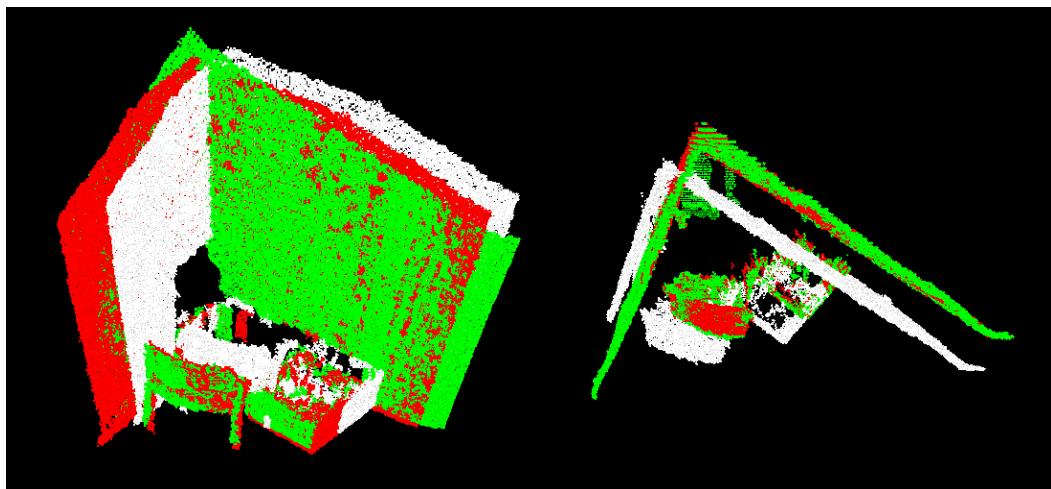


Figura 4.15: Pruebas de movimiento diagonal en (x,y) : en perspectiva a la izquierda, y a la derecha en planta.

1. Movimiento diagonal de 20 cm al frente y hacia la derecha

La cámara avanza hacia delante al mismo tiempo que hacia la derecha un total de 20 cm en diagonal y tratando, de forma aproximada, de equilibrar ambas direcciones, es decir, unos 14 cm en “x” y -14 cm en “y”. Es importante mencionar que sigue sin cambiar la orientación de la cámara; el movimiento es una traslación.

Funcionalidad: Estimación diagonal de 20 cm con “x” negativo e “y” positivo
Caso de prueba: P3.5
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas a una distancia de 20 cm entre sí sin cambiar su orientación y de tal modo que el valor de “x” sea negativo y el de “y” positivo con valores absolutos similares.
Salida esperada: Debe obtenerse una TF describiendo un desplazamiento de aproximadamente 0.2 metros sin rotaciones similar (aunque no exacta) a:
$(x, y, z) = \begin{array}{cccc} -0.1414 & 0.1414 & 0 \\ (R, P, Y) = \begin{array}{cccc} 0 & 0 & 0 \end{array} & \end{array}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) ubica la nueva posición aproximadamente 20 cm atrás y a la izquierda de su valor anterior.
Puntuación de ajuste: 0.000543965 con 60890 puntos.
$(x, y, z) = \begin{array}{cccc} 0.140735 & -0.157254 & -0.0189644 \\ (R, P, Y) = \begin{array}{cccc} 0 & 0 & 0 \end{array} & \end{array}$

Tabla 4.19: Resultado de la prueba sobre *Estimación diagonal de 20 cm con “x” negativo e “y” positivo*

Aunque se aprecia una medición indeseada en la altura, debida probablemente a cierta inclinación de la cámara debida a su inestabilidad, el valor total del desplazamiento medido equivale a la raíz cuadrada de “ $0.14^2+0.15^2$ ”, es decir, 0.2051 centímetros. Así, el error se sigue manteniendo dentro de los márgenes previstos.

2. Movimiento diagonal de 20 cm hacia la izquierda y atrás

La cámara avanza hacia atrás al mismo tiempo que a la derecha un total de 20 cm en diagonal y tratando, de forma aproximada, de equilibrar ambas direcciones, es decir, unos 14 cm negativos en “x” y +14 cm en “y”. Es importante mencionar que sigue sin cambiar la orientación de la cámara; el movimiento es una traslación.

Funcionalidad: Estimación diagonal de 20 cm con “x” positivo e “y” negativo
Caso de prueba: P3.6
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas a una distancia de 20 cm entre sí sin cambiar su orientación y de tal modo que el valor de “x” sea positivo y el de “y” negativo con valores absolutos similares.
Salida esperada: Debe obtenerse una TF describiendo un desplazamiento de aproximadamente 0.2 metros sin rotaciones similar a:
$(x, y, z) = \begin{matrix} 0.1414 \\ 0 \end{matrix}$ $(R, P, Y) = \begin{matrix} -0.1414 \\ 0 \\ 0 \end{matrix}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) ubica la nueva posición aproximadamente 20 cm al frente y a la derecha de su valor anterior.
Puntuación de ajuste: 0.000171606 con 52044 puntos.
$(x, y, z) = \begin{matrix} -0.142624 \\ 0 \end{matrix}$ $(R, P, Y) = \begin{matrix} 0.138239 \\ 0 \\ 0.0222241 \end{matrix}$

Tabla 4.20: Resultado de la prueba sobre *Estimación diagonal de 20 cm con “x” positivo e “y” negativo*

Si se compara con la medición anterior, se observa un incremento del error en la altura, pero también un cambio de signo que corrobora la hipótesis de que existe cierta inclinación residual del periférico. En cuanto al valor total del desplazamiento, al hacer los cálculos, se obtiene que 19.87 centímetros es la raíz cuadrada de “ $0.143^2+0.138^2$ ”. La diferencia es de sólo 1.3 milímetros respecto al valor esperado en este caso.

Rotaciones en el eje “z”

El segundo aspecto importante a tener en cuenta son las rotaciones en el plano del suelo (“xy”) en torno al eje “z”, es decir, lo que serían los giros o cambios de dirección sobre el suelo. Se analizarán en este caso los giros estacionarios (i.e. sin desplazamiento) entendiendo que las mediciones serán mejor interpretables y que los giros con desplazamiento son un combinación de estas pruebas y las ya mostradas. Las medidas se han realizado tomando como referencia la mitad del ángulo de visión de la cámara que es de 57 grados, por lo que cada prueba corresponderá a un giro de 28.5 grados.

1. Rotación de 28.5 grados hacia la izquierda

Se realiza un giro de la cámara en sentido positivo sobre el eje “z” sin que ésta se desplace. El eje “z” de la cámara debe quedar en el lugar original pero rotado 28.5 grados a la izquierda, es decir: unos 0.5 radianes.

Funcionalidad: Giro de +28.5 grados en el eje “z”
Caso de prueba: P3.7
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas antes y después de un giro positivo de 0.5 radianes sobre “z”.
Salida esperada: Debe obtenerse una TF describiendo el giro realizado tal que así:
$(x, y, z) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$ $(R, P, Y) = \begin{matrix} 0 \\ 0 \\ 0.5 \end{matrix}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) queda orientada 28.5 grados a su izquierda.
Puntuación de ajuste: 6.98102e-05 con 59337 puntos.
$(x, y, z) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$ $(R, P, Y) = \begin{matrix} 0.0311715 \\ 0.000286925 \\ 0.465911 \end{matrix}$

Tabla 4.21: Resultado de la prueba sobre *Giro de +28.5 grados en el eje “z”*

El resultado se ajusta aceptablemente a las expectativas si bien es necesario tener en cuenta que para este caso se utilizó una configuración muy restrictiva para los movimientos en x,y,z debido a unos valores residuales muy altos que, en las imágenes de control, demostraban no ser correctos. Filtrando dichos valores residuales, las imágenes de control son como se muestra en la figura 4.16.

2. Rotación de 28.5 grados hacia la derecha

Se realiza un giro de la cámara en sentido negativo sobre el eje “z” sin que ésta se desplace. El eje “z” de la cámara debe quedar en el lugar original pero rotado 28.5 grados a la derecha, es decir: unos -0.5 radianes.

Funcionalidad: Giro de -28.5 grados en el eje “z”

Caso de prueba: P3.8

Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas antes y después de un giro negativo de 0.5 radianes sobre “z”.

Salida esperada: Debe obtenerse una TF describiendo el giro realizado tal que así:

$$\begin{array}{lll} (x, \ y, \ z) = & 0 & 0 \\ (R, \ P, \ Y) = & 0 & -0.5 \end{array}$$

Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría *global* (acumulada) queda orientada 28.5 grados a su derecha.

Puntuación de ajuste: 0.000160604 con 59366 puntos.

$$\begin{array}{lll} (x, \ y, \ z) = & 0 & 0 \\ (R, \ P, \ Y) = & -0.00100641 & -0.000761005 \quad -0.499229 \end{array}$$

Tabla 4.22: Resultado de la prueba sobre *Giro de -28.5 grados en el eje “z”*

En este segundo caso, con la rotación hacia la derecha, los valores obtenidos son remarcablemente más exactos, si bien no está claro el motivo de esta diferencia. Se baraja la posibilidad de que, a pesar de la imprecisión de la cámara, los errores en este caso se han compensado de forma óptima. La configuración es idéntica a la del apartado anterior.

Rotaciones en el eje “z” con la cámara inclinada

Se especifica en los requisitos que la cámara no siempre comparte la orientación del robot, por lo que la siguiente prueba permite comprobar si, en caso de que la cámara esté orientada hacia el suelo y el robot en horizontal a éste, el movimiento obtenido es el del robot.

Este apartado requiere una configuración previa consistente en introducir el valor de cabeceo para que el software pueda compensarlo (tal como se explica en el manual de usuario en el apartado de configuración). Se utiliza una inclinación de 12 grados hacia abajo, es decir 0.21 radianes en sentido positivo del eje “y”.

1. Rotación de 28.5 grados hacia la izquierda con cabeceo de 12 grados en sentido positivo

Con la cámara orientada hacia el suelo, 0.21 radianes sobre el eje “y”, se realiza un giro en sentido positivo sobre el eje “z” sin que ésta se desplace. El eje “z” de la cámara debe quedar en el lugar original pero rotado 28.5 grados a la izquierda, es decir: unos 0.5 radianes.

Funcionalidad: Giro de +28.5 grados en el eje “z” con cabeceo de 12 grados
Caso de prueba: P3.9
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas con un cabeceo de 12 grados de la cámara, antes y después de un giro positivo de 0.5 radianes sobre “z”.
Salida esperada: Debe obtenerse una TF describiendo el mismo giro que si no hubiera cabeceo:
$(x, \ y, \ z) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$ $(R, \ P, \ Y) = \begin{matrix} 0 \\ 0 \\ 0.5 \end{matrix}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) queda orientada 28.5 grados a su izquierda.
Puntuación de ajuste: 0.00134077 con 51032 puntos.
$(x, \ y, \ z) = \begin{matrix} -0.0239631 \\ 0.000843968 \\ 0.0298359 \end{matrix}$ $(R, \ P, \ Y) = \begin{matrix} 0.0250248 \\ 0.0207128 \\ 0.485172 \end{matrix}$

Tabla 4.23: Resultado de la prueba sobre *Giro de +28.5 grados en el eje “z” con cabeceo de 12 grados*

En el resultado se observa un amplio error de unos 3 centímetros en altura (eje “z”) que es lo más destacable de la medición y hace dudosa la validez de la prueba actual. Dado que es un error persistente (se repitieron los cálculos sobre las mismas capturas), se interpreta que el algoritmo tiene dificultades con este ejemplo en particular. Por otro lado, es destacable que el giro medido se approxima correctamente a los 0.5 radianes esperados, por lo que se puede clasificar este resultado como “correcto pero con excesivo error en una de las medidas”.

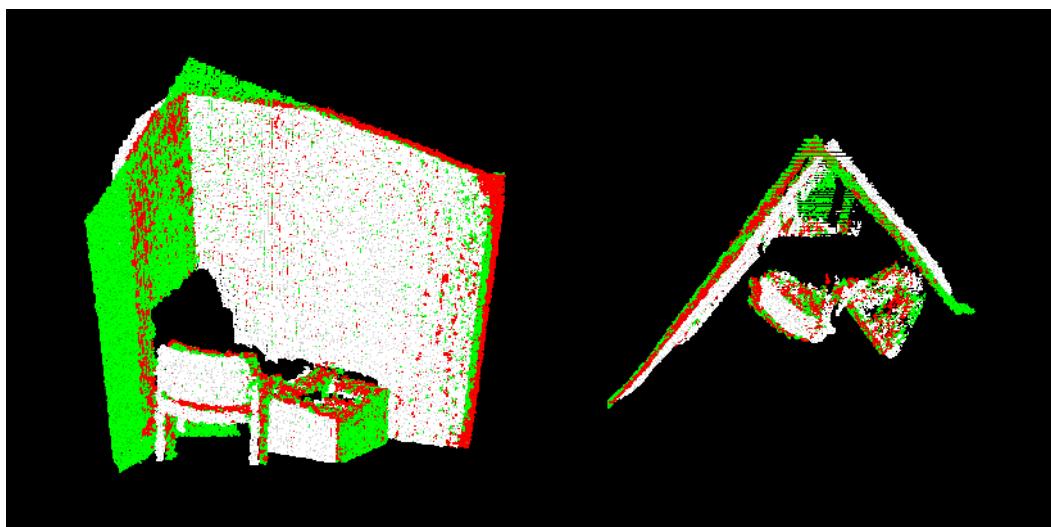


Figura 4.16: Pruebas de giros en el eje “z”

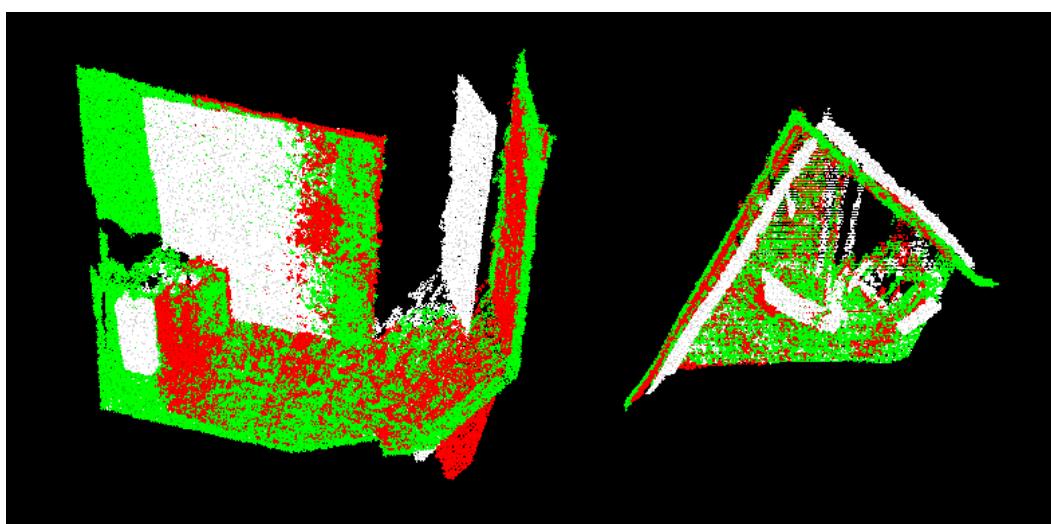


Figura 4.17: Pruebas de giros con la cámara inclinada

2. Rotación de 28.5 grados hacia la derecha con cabeceo de 12 grados en sentido positivo

Con la cámara orientada hacia el suelo, 0.21 radianes sobre el eje “y”, se realiza un giro en sentido negativo sobre el eje “z” sin que ésta se desplace. El eje “z” de la cámara debe quedar en el lugar original pero rotado 28.5 grados a la derecha, es decir: unos -0.5 radianes.

Funcionalidad:	Giro de -28.5 grados en el eje “z” con cabeceo de 12 grados		
Caso de prueba:	P3.10		
Descripción de los valores a introducir:	Se introducen dos nubes de puntos tomadas con un cabeceo de 12 grados de la cámara, antes y después de un giro negativo de 0.5 radianes sobre “z”.		
Salida esperada:	Debe obtenerse una TF describiendo el mismo giro que si no hubiera cabeceo:		
(x, y, z) =	0	0	0
(R, P, Y) =	0	0	-0.5
Resultado obtenido:	La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) queda orientada 28.5 grados a su derecha.		
Puntuación de ajuste:	8.24156e-05	con 61812 puntos.	
(x, y, z) =	-0.00779832	-0.0186766	0.000739282
(R, P, Y) =	-0.0334476	-0.0285023	-0.480757

Tabla 4.24: Resultado de la prueba sobre *Giro de -28.5 grados en el eje “z” con cabeceo de 12 grados*

Una vez más se acusan diferencias respecto a la prueba con giro positivo. Aunque el valor buscado es algo menos exacto (-0.48), las medidas en x,y,z se mantienen en valores razonables menores a 2 centímetros. Los valores de cabeceo y guiñada (giros en “x” e “y”) se pueden deber a la manipulación de la cámara durante el giro, si bien se trató de evitar esto en la medida de lo posible.

Rotaciones en el eje “y”: cabeceo

Una prueba adicional que no estaba contemplada inicialmente es tratar de medir el cabeceo (giro sobre el eje “y”, en inglés *pitch*) de la cámara. Ésta incorpora un mecanismo para moverse un número determinado de grados, lo que facilita las mediciones en este sentido y aporta valor a esta prueba. Una vez más, se utiliza como referencia el ángulo de visión de la cámara, que en vertical es de 45 grados y se realizarán medidas de 22.5 grados mediante acumulación (es decir, a partir de una secuencia y no sólo dos capturas).

1. Cabeceo de 22.5 grados en sentido positivo

Suponiendo una orientación paralela al suelo, la cámara se inclina 22.5 grados hacia el suelo sobre el eje “y”, es decir, un cabeceo que debe equivaler a 0.39 radianes en sentido positivo.

Funcionalidad: Cabeceo de 22.5 grados en sentido positivo
Caso de prueba: P3.11
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas antes y después de un giro positivo de 0.39 radianes sobre “y”.
Salida esperada: Debe obtenerse una TF describiendo el cabeceo de la cámara, esta vez en la “P” de <i>pitch</i> :
$(x, y, z) = \begin{matrix} 0 \\ 0 \end{matrix}$ $(R, P, Y) = \begin{matrix} 0.39 \\ 0 \end{matrix}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) queda orientada 22.5 grados hacia abajo.
Puntuación de ajuste: 0.000484956 con 58366 puntos.
$(x, y, z) = \begin{matrix} -0.00226356 \\ 4.32842e-05 \\ -0.0242938 \end{matrix}$ $(R, P, Y) = \begin{matrix} 0.00732758 \\ 0.375822 \\ 0.00847287 \end{matrix}$

Tabla 4.25: Resultado de la prueba sobre *Cabeceo de 22.5 grados en sentido positivo*

Vuelve a aparecer, en los siguientes resultados, un error en altura de unos 2.5 centímetros que podría ser un problema tras la acumulación de una serie mayor de medidas. En cuanto al ángulo medido, sí que se corresponde con el valor esperado que dista sólo 0.014 radianes de los 0.39 radianes que corresponden en realidad.

2. Cabeceo de 22.5 grados en sentido negativo

La cámara se inclina 22.5 grados hacia arriba sobre el eje “y”, es decir, un cabeceo que debe equivaler a 0.39 radianes en sentido negativo.

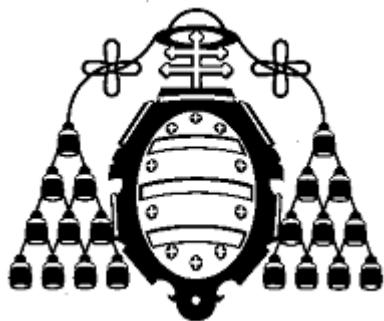
Funcionalidad: Cabeceo de 22.5 grados en sentido negativo
Caso de prueba: P3.12
Descripción de los valores a introducir: Se introducen dos nubes de puntos tomadas antes y después de un giro negativo de 0.39 radianes sobre “y”.
Salida esperada: Debe obtenerse una TF describiendo el cabeceo de la cámara, esta vez en la “P” de <i>pitch</i> :
$(x, y, z) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$ $(R, P, Y) = \begin{matrix} 0 \\ -0.39 \\ 0 \end{matrix}$
Resultado obtenido: La TF del movimiento es obtenida y almacenada, la nube de puntos más nueva se guarda para la siguiente iteración y la TF de odometría <i>global</i> (acumulada) queda orientada 22.5 grados hacia abajo.
Puntuación de ajuste: 0.000189373 con 60278 puntos.
$(x, y, z) = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$ $(R, P, Y) = \begin{matrix} -0.0185721 \\ -0.388211 \\ -0.00150684 \end{matrix}$

Tabla 4.26: Resultado de la prueba sobre *Cabeceo de 22.5 grados en sentido negativo*

TODO: Usar la siguiente explicación si es necesario: “si se supone una inclinación mínima de 2 grados debida simplemente a la holgura de la cámara, el resultado pasa a ser el siguiente:”

TODO 1: Usar la siguiente explicación si es necesario: “si se supone una inclinación mínima de 2 grados debida simplemente a la holgura de la cámara, el resultado pasa a ser el siguiente:”

Con un tiempo de cálculo similar al obtenido para el giro positivo, se observan en este caso, valores nulos para x,y,z que no deben entenderse como tal, sino que en cada medida individual éstos eran lo bastante pequeños como para ser despreciados. En cualquier caso, la medida de cabeceo es casi exacta en esta ocasión, demostrando que determinadas capturas son mejor entendidas por el algoritmo aún para una misma configuración del mismo.



UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE
GIJÓN

**ÁREA DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL**

PROYECTO FIN DE CARRERA N° 3122067

**INTEGRACIÓN DEL DISPOSITIVO 'MICROSOFT KINECT'
COMO PARTE DE UNA ARQUITECTURA ROBÓTICA
GENÉRICA PARA SU USO COMO DISPOSITIVO DE ENTRADA**

DOCUMENTO N° V

MANUAL DE USUARIO



MIGUEL A. GARCÍA GONZÁLEZ

NOVIEMBRE – 2012

TUTOR: LUCIANO SÁNCHEZ RAMOS

CAPÍTULO 5

MANUAL DE USUARIO

5.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

5.1.1. Identificación del proyecto

Título:	“Integración del dispositivo ‘Microsoft Kinect’ como parte de una arquitectura robótica genérica”
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

5.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo “Microsoft Kinect” comercializado por la empresa “Microsoft Corporation”. La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

5.1.3. Visión general del documento

En el presente documento se ofrece toda la información necesaria para poner en funcionamiento el software desarrollado y comenzar su utilización *por parte del usuario final*. En él se realizará un repaso por las funciones de las dos piezas software que componen el proyecto y la forma de utilizarlas para obtener los resultados deseados, se incluirán también comentarios de las *acciones que son realizadas de forma pasiva o automatizada para comodidad del usuario*.

5.2. Manual de usuario de la interfaz

A continuación se explica cómo manejar las funciones de la interfaz que se ha desarrollado, mediante la utilización de comandos, ficheros de lanzamiento (*launch files*) y canales de comunicación de ROS. Se incluyen la lista de requisitos, la instalación del software necesario y el manejo de las funciones más relevantes para el correcto funcionamiento del proyecto.

Para facilitar la lectura, los comandos a utilizar se encuentran recuadrados siguiendo a la explicación su funcionamiento en cada caso. Sin embargo parece ser que se modifican caracteres al “seleccionar y copiar” automáticamente el contenido debido a la naturaleza del fichero PDF y será necesario, en cada caso, escribir los comandos manualmente.

5.2.1. Requisitos necesarios

- Requisitos hardware (equipo de las pruebas)
 - Procesador Intel Atom -Silverthorne- (1.3 GHz) o superior
 - 1 GiB de Memoria RAM
 - 10 GiB de espacio en el disco duro
 - Teclado y ratón para configurar y controlar el entorno
 - Dispositivo de destino (por ejemplo Microsoft Kinect)
 - Puerto de conexión USB 2.0, WiFi b/g, Ethernet 10/100, etc. Según dispositivo de destino
- Requisitos software
 - Ubuntu Linux 10.04 (Lucid Lynx)
 - Robot Operating System (ROS) versión *Fuerte*
 - Paquetes de Linux:
 - python2.6
 - python2.6-dev
 - python-yaml
 - Software adicional
 - Paquetes ROS de los controladores de posibles dispositivos a manejar. En el caso del Kinect:
 - ◊ ros-fuerte-openni-kinect
 - ◊ ros-fuerte-kinect-aux

5.2.2. Instalación del entorno

Antes de poder utilizar la interfaz es necesario instalar el entorno ROS sobre el que trabaja, así como el conjunto de ficheros de la propia interfaz y algunos paquetes adicionales utilizados por el código de la misma. No será necesario repetir los pasos que se hayan realizado en una posible instalación anterior de la parte de odometría.

La instalación recomendada se realiza, acorde a los requisitos mencionados, sobre Ubuntu Linux y con una versión de ROS que puede ser “electric” o una posterior. En este caso se utiliza Ubuntu Linux 10.04 junto con ROS fuerte y la versión 2.6 de python por ser versiones con las que ya se ha probado el software.

1. El primer paso será la instalación del propio entorno ROS siguiendo las instrucciones de la página oficial¹, pero que se enumeran a continuación como parte imprescindible del presente documento.

a) Si el sistema operativo no está configurado para permitir repositorios “restricted”, “universe” y “multiverse”, estos pueden activarse desde la ventana de “Software Resources” o “Recursos Software”, en el menú de “Sistema / Administración” que se muestra en la Figura 5.1.

La explicación original y actualizada, en caso de que hubiera cambios en las futuras versiones, puede encontrarse en la siguiente dirección:

“<https://help.ubuntu.com/community.Repositories/Ubuntu>”

b) Una vez activados los repositorios, ya se pueden incluir sus direcciones en el fichero *ros-latest.list* dentro de “/etc/apt/sources.list.d”. Para las versiones utilizadas en este manual, se puede ejecutar directamente el siguiente comando.

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

c) También es necesario configurar las claves para el acceso al repositorio de la siguiente manera:

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

d) Con los repositorios correctamente preparados para la instalación se utiliza la herramienta “apt-get” según los pasos habituales con el entorno ROS como objetivo.

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

Sólo para la parte de odometría: la versión “-full” será utilizada debido a que se requieren los paquetes de navegación y de percepción que no se incluyen en la versión “desktop” básica.

```
# Si se actualizo recientemente con "sudo apt-get update"  
sudo apt-get install ros-fuerte-desktop-full
```

e) Recién acabada la instalación, es recomendable incluir el fichero “setup.bash” de ROS en el fichero “.bashrc” para que el primero sea ejecutado al inicio definiendo así las variables de entorno necesarias para el correcto funcionamiento de ROS.

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

¹Instrucciones de instalación de ROS en su web: “www.ros.org/wiki/fuerte/Installation/Ubuntu”



Figura 5.1: Ubuntu - Software Sources Panel

La línea equivalente, si no se desea que los cambios sean permanentes, es «`source /opt/ros/fuerte/setup.bash`». Las variables de entorno desaparezcan al cargar de nuevo el entorno Linux.

Hecho esto, el entorno ROS, que es la base sobre la que se trabaja, deberá haber quedado completamente instalado y funcional.

2. Será necesario instalar Python 2.6 tanto en su versión de intérprete como de desarrollo y la librería YAML para las configuraciones. Para ello se utiliza la herramienta apt-get:

```
# Si se actualizo recientemente con "sudo apt-get update" basta escribir:  
sudo apt-get install python2.6 python2.6-dev python-yaml
```

3. El driver que se quiera controlar desde la interfaz debe ser instalado individualmente según las necesidades del usuario. Para el caso particular del Kinect, la web de ROS recomienda el paquete *openni_kinect* asociado a la versión de ROS utilizada y el paquete *camera_drivers*. Eso se traduce en lo siguiente al utilizar apt-get para la instalación:

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install ros-fuerte-openni-kinect ros-fuerte-camera-drivers
```

Adicionalmente, se puede instalar también *kinect_aux* para mayor control sobre el dispositivo Kinect y aprovechando la capacidad de la interfaz para actuar como acceso común a toda la configuración de ambos drivers. En este caso es necesario utilizar *git* y compilar el paquete en la máquina local al no encontrarse disponible mediante *apt-get*.

- a) Es necesario instalar primero el propio cliente git (que sí está disponible mediante *apt-get*).

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install git
```

- b) Después se accede al directorio de ROS que contiene los paquetes de tipo *stack* (conjuntos o colecciones de paquetes relacionados) dado que *kinect_aux* es de dicho tipo. El directorio actual de stacks se encuentra en */opt/ros/fuerte/stacks* y la instalación se realiza de la siguiente manera:

```
cd /opt/ros/fuerte/stacks
sudo git clone https://github.com/ros-pkg-git/kinect.git
sudo chmod 777 -R kinect_aux # Para que rosmake pueda funcionar sin privilegios
cd kinect_aux
rosmake
```

4. Antes de llegar al último paso es necesario instalar el cliente git que está disponible mediante *apt-get* y permite descargar y actualizar de forma fácil el código fuente de los paquetes.

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install git
```

5. Tal como cabe esperar: el último paso, una vez instaladas todas las dependencias, es instalar los ficheros de la interfaz, en este caso desde el repositorio en el que se encuentran alojados, dentro de *GitHub.com*². A efectos de mostrar un ejemplo más completo, se instalarán los archivos en una carpeta nueva de proyectos en vez de utilizar las carpetas ROS existentes.

```
cd /home/usr_name # Nombre de usuario "usr_name" como ejemplo
sudo git clone https://github.com/michi05/my_adaptor
cd my_adaptor
rosmake
```

5.2.3. Configuración de la interfaz: fichero de lanzamiento

Durante el desarrollo, se trató de aprovechar el sistema de ficheros de lanzamiento de ROS, que favorece el uso de una jerarquía de responsabilidades en los mismos de tal manera que cada fichero invoca a otros hasta que todo el entorno es arrancado. A continuación se detallan los distintos lanzadores disponibles para poder configurar el comportamiento de la interfaz y sus opciones de configuración correspondientes.

robotDrivers.- Define y lanza un único robot al completo pudiendo incluir múltiples dispositivos confinados en un

²La dirección del repositorio es “https://github.com/michi05/my_adaptor”

espacio de nombres común para el robot. Hay dos argumentos que se pueden asignar a este nivel y sus valores por defecto tras el signo “igual”:

- **robot_ns** = robot1

Define el espacio de nombres relativo en el cual incluir todos los componentes del robot.

- **absolute_namespace** = /

Contiene el espacio de nombres absoluto bajo el cual se está ejecutando *de facto* el fichero de lanzamiento actual. Va contra el carácter genérico del proyecto pero es imprescindible para que el driver openni de kinect funcione correctamente.

kinectDrivers.- Es un lanzador a nivel de dispositivo exclusivamente para el Kinect. Se ocupa de lanzar tanto la interfaz (objeto de este manual) como los drivers, para no introducir más complejidad con un nivel adicional innecesario. Este fichero requiere los mismos argumentos que *robotDrivers.launch* y, adicionalmente, los siguientes:

- **camera_name** = kinect

El nombre de la cámara para determinar el espacio de nombres.

- **main_file_basename** = interface_node_main

Contiene el nombre del archivo principal.

- **node_name** = interface_node_main

El nombre del nodo principal que es por defecto el del archivo principal.

- **pkg_name** = my_adaptor

Contiene el nombre del nodo principal.

- **pkg_location** = \$(find my_adaptor)

Contiene la localización del paquete especificado bajo *pkg_name*.

- **translation_file** = parameterDictionary.yaml

Determina el nombre del diccionario de parámetros [siguiente subsección].

Para utilizar dichos ficheros se utiliza el comando « *roslaunch* » y sus parámetros pueden cambiarse con antelación modificando el código de cada fichero o pueden incluirse a continuación usando el operador “:=”.

```
roslaunch paqueteROS ficheroLanzador.launch argumento1:=valor1 argumento2:=valor2
```

Por ejemplo:

```
roslaunch my_adaptor kinectDrivers.launch camera_name:=kinect1
```

El hecho de que el primer lanzador incluya al segundo pero sea éste el que tiene más argumentos se debe a que los argumentos son privados y deben ser declarados en cada fichero “.launch”. Esto significa que si en *robotDrivers* se declaran todos los argumentos de los lanzadores que incluye, el listado podría ser demasiado largo. Por defecto cada cuál tiene sus propios argumentos y será un usuario avanzado el que decida si desea cambiar la estructura de lanzamiento.

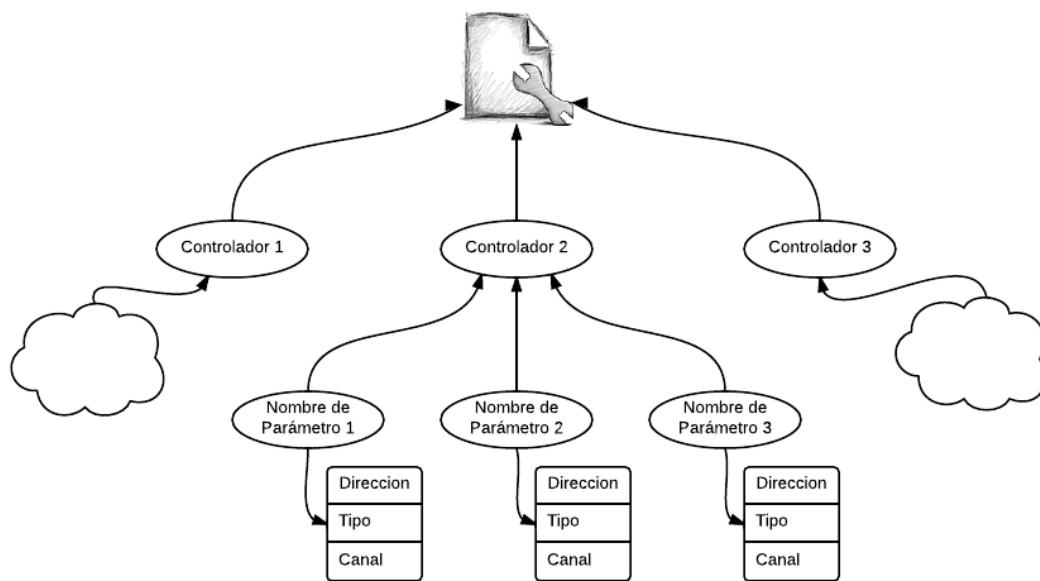


Figura 5.2: Diagrama en árbol de la estructura del diccionario

5.2.4. Configuración de la interfaz: diccionario de parámetros

Quizá la aportación más importante al diseño sea la posibilidad de configurar en un mismo fichero todos los parámetros manejados por el driver (o varios drivers) para que formen parte de la misma lista de *parámetros dinámicos* y, sobre todo, puedan ser configurados de manera análoga y con identificadores (nombres) al gusto del usuario. La estructura del fichero, basada en lenguaje YAML, es la que se describió en el apartado 4.5 del documento *Diseño*, que se correspondía con la figura 5.2.

```
# Los comentarios usan almohadilla segun el convenio del lenguaje YAML
'ruta/driver': # La ruta puede ser vacia.
  #Si un driver usa varias se agregan como si fueran drivers distintos
  nombre_parametro: # El nombre nuevo deseado para este parametro
    - alias_parametro # El nombre o identificador del parametro segun el controlador
    - tipo_dato # El tipo de dato basico: entero (int), texto (string)...
    - clase_parametro # La clase de parametro segun el canal de comunicacion con el driver
      #Por ahora puede ser: "topic" o "parametro_dinamico" (dynParam)
```

Y este es un ejemplo real utilizado durante el desarrollo:

```
'camera/driver/':
  image_mode:
    - camera	driver/image_mode
    - int
    - dynParam
```

El fichero utilizado por defecto se llama “parameterDictionary.yaml” aunque puede ser sustituido o modificarse fácilmente con los parámetros del apartado anterior.

Fichero de configuración de “dynamic_reconfigure”

Hasta la versión actual de ROS, y desde la reciente introducción del servidor dinámico de parámetros (“dynamic_reconfigure”), es necesario escribir un fichero de extensión “.cfg” por cada nodo que utiliza dicho servidor para conocer los parámetros de los que debe ocuparse. Si bien es probable que este sistema cambie a corto plazo, el usuario se ve obligado a repetir la configuración del fichero de traducción en dicho fichero “.cfg”.

La configuración de dicho fichero consiste en la introducción de una línea de la forma «`gen.add ("nombre_parametro", tipo_dato, 0, "Descripcion_si_se_desea", valor_por_defecto, valor_min, valor_max)`» con los campos indicados (nombre, tipo de dato, un cero³, descripción, valor por defecto, valor mínimo y valor máximo) por cada parámetro a controlar. Los valores mínimo y máximo sólo se utilizan para los tipos enteros y reales, y los tipos posibles son: “int_t”, “double_t”, “str_t” y “bool_t”. Si se desea una explicación más amplia y detallada, se puede consultar el tutorial de “Cómo configurar tu primer fichero CFG” en la web de ROS⁴.

En el proyecto actual existe ya un fichero “Properties.cfg” (dentro de la carpeta “cfg”) que responde al fichero de traducciones por defecto. Para que el entorno reconozca cambios en el fichero habrá que ejecutar lo siguiente:

```
# Colocarse en la carpeta del proyecto
roscd my_adaptor
# Marcar el archivo como ejecutable
#(en caso de que haya borrado el original)
chmod a+x cfg/Properties.cfg
# Compilar el proyecto con la nueva configuracion
rosmake
```

Para futuros desarrollos, si el diseño actual ofrece buenos resultados, se planea programar un generador que produzca ambos ficheros de forma paralela para facilitar la labor del usuario en caso de que el fichero de configuración de su dispositivo no exista ya de antemano.

5.2.5. Arranque inicio y cierre de la interfaz

Una vez completada la instalación del software y las dependencias necesarias, será posible arrancar la interfaz mediante un simple comando `roslaunch paquete launchfile` en la consola de Linux y cerrarlo, en condiciones normales, enviando una señal al mismo mediante la combinación de teclas *Ctrl+C*. Esto se debe al diseño de la interfaz, orientado a facilitar su arranque rápido y configuración automática una vez que los parámetros han sido plasmados en los ficheros de configuración correspondientes.

El comando básico para el arranque es «`roslaunch my_adaptor robotDrivers.launch`», si bien puede sustituirse el fichero de lanzamiento (*robotDrivers.launch*) o realizar el lanzamiento manualmente ejecutando primero los drivers y luego la interfaz tras introducir los parámetros necesarios en el servidor de parámetros:

³Aclaración: el cero (0) del tercer campo sirve para distinguir grupos de parámetros. No tiene sentido en el manual de usuario pues los parámetros no recibirán un tratamiento especial sin modificar el código fuente del nodo.

⁴Se puede acceder al tutorial “Cómo configurar tu primer fichero CFG” a través del a siguiente dirección: “www.ros.org/wiki/dynamic_reconfigure/Tutorials/HowToWriteYourFirstCfgFile”

```

roscore & # Debe existir una instancia de roscore en todo momento
# Lanzar los drivers deseados y configurados en el espacio de nombres adecuado
roslaunch openni_launch openni.launch # Driver Openni del Kinect
# Guardar los parametros de lanzamiento en el servidor de parametros
#un posible ejemplo seria...
rosparam set property_config_file parameterDictionary.yaml
# Ejecutar el nodo de la interfaz
rosrun my_adaptor interface_node_main.py

```

5.2.6. Consultar el valor de un parámetro

Una vez está en marcha la interfaz, ésta hará de intermediaria entre aplicaciones de la capa superior y los controladores, de forma que no se requiera ninguna acción adicional por parte del usuario. Aún así es posible consultar y modificar la configuración del driver con ayuda de la interfaz. La consulta se puede llevar a cabo de dos maneras distintas: mediante el servicio adecuado en la línea de comando, o con una pequeña aplicación incluida de antemano por ROS.

En el primer caso bastaría con llamar al servicio “get/<tipo>Property” cambiando “tipo” por el tipo de dato del parámetro entre: Int, String, Bool o Float. La llamada completa a un parámetro de tipo entero quedaría como en el siguiente ejemplo:

```
rosservice call /kinect1/get/IntProperty depth_resolution
```

En este ejemplo se consulta “depth_resolution” y se obtendría una salida similar a « paramValue:2 ».

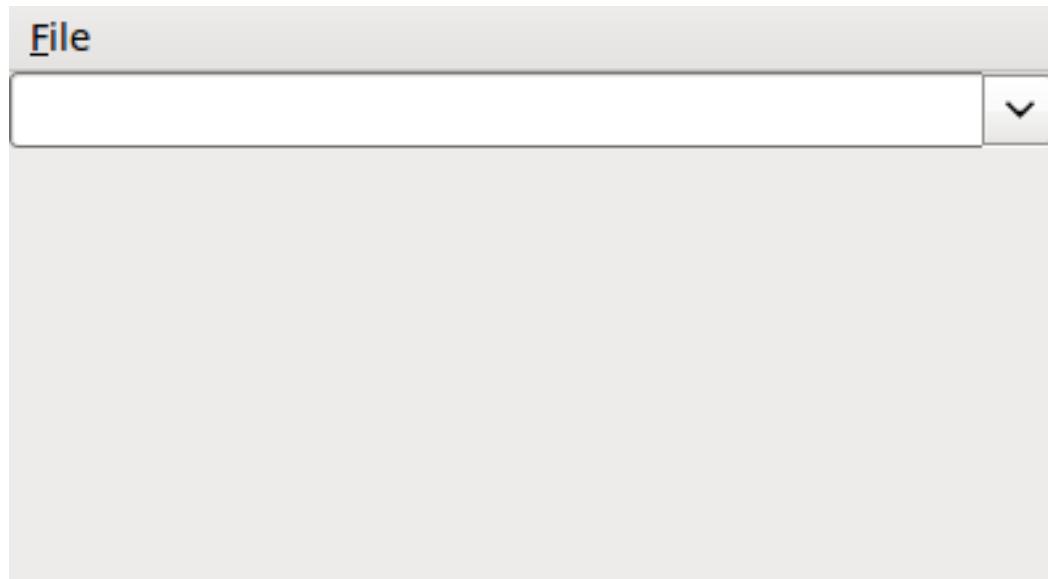


Figura 5.3: Pantalla inicial del configurador de parámetros dinámicos

La ejecución de « **rosrun** dynamic_reconfigure reconfigure_gui » realizaría la misma acción, de tal forma que emergería una ventana de interfaz tan simple como la de la figura 5.3. En ella se debe elegir un nodo del desplegable para llegar a la siguiente pantalla (figura 5.4). En este caso será el que termine con el nombre del nodo: *img_iface* aunque puede personalizarse la ruta desde los ficheros

de lanzamiento. El usuario puede consultar desde ahí la lista completa de parámetros incluyendo el del ejemplo y los valores actuales aparecen en la columna de la derecha.

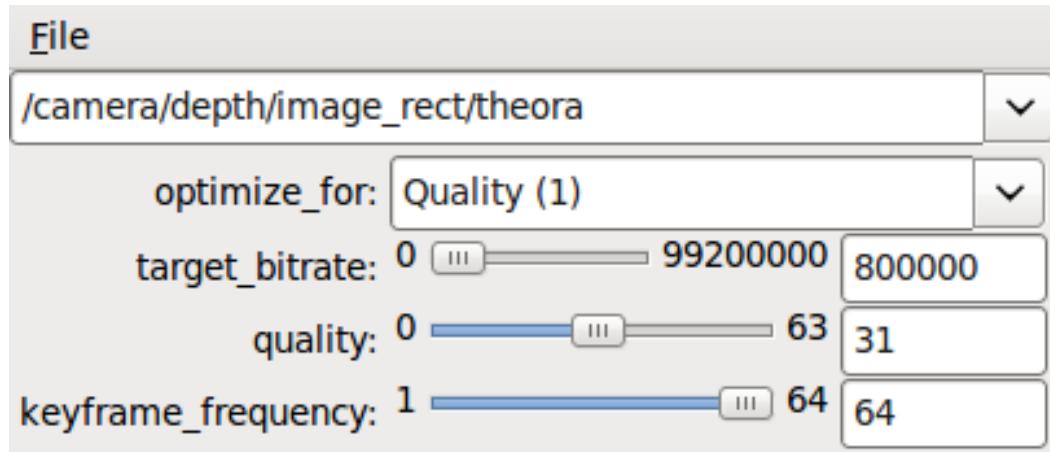


Figura 5.4: Lista de parámetros dinámicos configurables

5.2.7. Modificar parámetros del driver en tiempo de ejecución

También se ofrece la opción de realizar ajustes manuales mediante el servidor dinámico de parámetros (*Dynamic Parameter Server*) ya sea creando un nodo ROS personalizado que actúe como cliente de *dynamic_reconfigure* o mediante la aplicación *reconfigure GUI* del propio entorno ROS. Sólo se explicará aquí la segunda, pues la primera está orientada a desarrolladores.

Para lanzar la herramienta de reconfiguración se ejecuta, con la interfaz ya en marcha: « `rosrun dynamic_reconfigure reconfigure_gui` ». Inmediatamente, aparece la ventana de la figura 5.3 y, como ya se dijo en el apartado anterior, se elige el nodo *img_iface* en el desplegable.

Una vez dentro, en la siguiente pantalla, la propia interfaz indica los valores mínimos y máximos (si los hay) y permite la modificación de los parámetros ya sea mediante barras deslizantes, marcas on/off o cajas de texto. Al ser parámetros dinámicos, los cambios que se realicen serán aplicados de forma instantánea a la interfaz y, desde ahí, a los controladores.

Como alternativa se puede escribir « `rosservice call /kinect1/set/<tipo>Property nombre_parametro` » adaptado al parámetro que se desee sustituyendo “tipo” por el tipo de dato (Bool, Int, Float, String) y siguiendo con el nombre y el nuevo valor del parámetro respectivamente.

```
rosservice call /kinect1/set/IntProperty depth_resolution 1
```

5.2.8. Consultar la dirección de un topic

Puede ser deseable, en un momento determinado, averiguar la dirección de un recurso publicado en forma de “topic”, de tal manera que la aplicación que lo solicita pueda conectarse directamente a él sin depender de la interfaz. Existe, en este caso, sólo una manera de realizar esa consulta y es mediante el

servicio apropiado: “get/TopicLocation”. Éste responderá al alias de un parámetro o recurso como único argumento, es decir, al nombre que se le asignó en el diccionario (hay que tener en cuenta que las aplicaciones nunca tienen que conocer el identificador a nivel del controlador). Un ejemplo sería la consulta siguiente:

```
rosservice call /kinect1/get/TopicLocation depth_reg_raw
```

Dado que “depth_reg_raw” es un topic, deberá tener asociada una dirección relativa donde se publica dentro del entorno ROS. En este caso se obtiene la respuesta “paramValue: depth_registered/image_rect_raw”, que significa que el recurso puede ser encontrado en la dirección absoluta “/kinect1/depth_registered/image_rect_raw”.

5.2.9. Renombrar un topic

Es posible renombrar (o reubicar) los topics publicados por el controlador, pero se debe ser consciente de que, para cada topic, se creará un nodo ROS que “repetirá” uno por uno cada mensaje del topic, con las desventajas de rendimiento que ello conlleva. Para conseguir este resultado existen 2 métodos según las necesidades del usuario:

Al inicio, gracias al diccionario de parámetros. Incluyendo un parámetro adicional en el fichero “parameterDictionary.yaml” presentado en 5.2.4, la interfaz renombrará el topic del parámetro correspondiente durante el arranque. El código tiene que ser como el siguiente:

```
'ruta/driver':  
    nombre_parametro: # El nombre asignado o alias (no en el driver)  
        - ruta_nueva # La dirección de topic para ser asignada  
        - tipo_dato # El tipo del topic se mantiene solo por compatibilidad  
        - renaming # La clase determina la acción a realizar
```

Al escribir “renaming” como clase del parámetro, la interfaz entiende que se trata de un topic y que debe reubicarlo en la ruta nueva especificada. Tras reubicar el topic se almacena la ruta nueva para el parámetro “nombre_parametro” y se continua con normalidad.

Mediante el servicio apropiado, enviando una petición. Para algunos casos puede ser interesante cambiar la ruta de un topic en tiempo de ejecución. Para ello se prové un servicio “set/TopicLocation” que recibe el identificador del topic y la ruta deseada, y modifica su ruta (ya conocida) dentro del entorno ROS.

Un ejemplo de llamada a dicho servicio sería: « `rosservice call /ruta_adaptador/set/TopicLocation nombre_parametro ruta_nueva` », que cambiaría la dirección del topic asociado a “nombre_parametro” por “ruta_nueva”, que a su vez puede ser una ruta absoluta o relativa, en cuyo caso se ubicará usando el driver como raíz.

5.3. Manual de usuario de la odometría.

El nodo dedicado al cálculo de la odometría visual, si bien es prácticamente automático una vez puesto en marcha, también requiere de cierta explicación previa a su utilización. Su responsabilidad principal consiste en recibir pares de nubes de puntos que, como precondición, se consideran consecutivas al inicio y final de un movimiento; y determinar el movimiento para publicarlo inmediatamente en términos de desplazamiento y cambio de orientación. Intervienen en este caso los topics de lectura de nubes de puntos y un grupo de servicios de control adicionales. Toda la configuración se realiza mediante parámetros básicos estáticos de ROS que son leídos al ejecutar el nodo. Además de la lista de requisitos, se enumeran a continuación los pasos de instalación y los que corresponden al manejo completo de la aplicación en el caso más complejo.

5.3.1. Requisitos necesarios

- Requisitos hardware (equipo de las pruebas)
 - Procesador Intel Pentium T4500 (2.30 GHz) o superior
 - 2 GiB de Memoria RAM
 - 20 GiB de espacio en el disco duro
 - (incluyendo un pequeño margen para actualizaciones y ficheros temporales)
 - Teclado para el control del entorno
 - Microsoft Kinect
 - Puerto de conexión USB 2.0 para el Kinect
- Requisitos software
 - Ubuntu Linux 10.04 (Lucid Lynx)
 - Robot Operating System (ROS) versión *Fuerte*
 - Paquetes de Linux:
 - python2.6
 - python2.6-dev
 - Paquetes (de Linux) específicos para ROS:
 - ros-fuerte-openni-kinect (controlador openni para el Kinect)
 - ros-fuerte-perception-pcl (librería de nubes de puntos PCL)

El paquete *python2.6-dev* es utilizado por ROS cuando se modifica el fichero de configuración del servidor dinámico de parámetros, que se introducirá más adelante. Pueden existir requisitos “implícitos” que no son especificados en la documentación de ROS porque se instalan como dependencias de los nombrados.

5.3.2. Instalación del entorno

Antes de poder utilizar el nodo de odometría visual es necesario instalar el entorno ROS que ofrece los canales de comunicación para el mismo, el propio nodo y también los distintos paquetes adicionales requeridos para el código por el código para los cálculos y las comunicaciones. No será necesario repetir los pasos que se hayan realizado ya durante la instalación de la interfaz en caso de haberse realizado primero.

Se utiliza Ubuntu Linux 10.04 para mostrar la instalación básica y como versión de ROS se utilizará “ROS fuerte” debido a ciertas incompatibilidades que ofrece la versión anterior de ICP con algunos parámetros que antes no eran modificables. En cualquier caso la versión de Python es la 2.6 por ser, también, la más antigua con la que se puede garantizar el funcionamiento correcto.

1. El primer paso será la instalación del propio entorno ROS siguiendo las instrucciones de la página oficial⁵, pero que se enumeran a continuación como parte imprescindible del presente documento.
 - a) Si el sistema operativo no está configurado para permitir repositorios “restricted”, “universe” y “multiverse”, estos pueden activarse desde la ventana de “Software Resources” o “Recursos Software”, en el menú de “Sistema / Administración” que se muestra en la Figura 5.5.
La explicación original y actualizada, en caso de que hubiera cambios en las futuras versiones, puede encontrarse en la siguiente dirección:
“<https://help.ubuntu.com/community.Repositories/Ubuntu>”
 - b) El segundo paso es incluir las direcciones de los repositorios. A continuación se muestra el comando exacto para las versiones especificadas anteriormente que son utilizadas en este manual.

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

- c) Es necesario configurar las claves para el acceso al repositorio de la siguiente manera:

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

- d) Con los repositorios correctamente preparados para la instalación se utiliza la herramienta “apt-get” según los pasos habituales con el entorno ROS como objetivo.

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

Sólo para la parte de odometría: la versión “-full” será utilizada debido a que se requieren los paquetes de navegación y de percepción que no se incluyen en la versión “desktop” básica.

```
# Si se actualizo recientemente con "sudo apt-get update"  
sudo apt-get install ros-fuerte-desktop-full
```

- e) Una vez acabada la instalación, es necesario incluir el fichero “setup.bash” de ROS en el fichero “.bashrc” para que el primero sea ejecutado al inicio definiendo así las variables de entorno necesarias para el correcto funcionamiento de ROS.

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

⁵Instrucciones de instalación de ROS en su web: “www.ros.org/wiki/fuerte/Installation/Ubuntu”

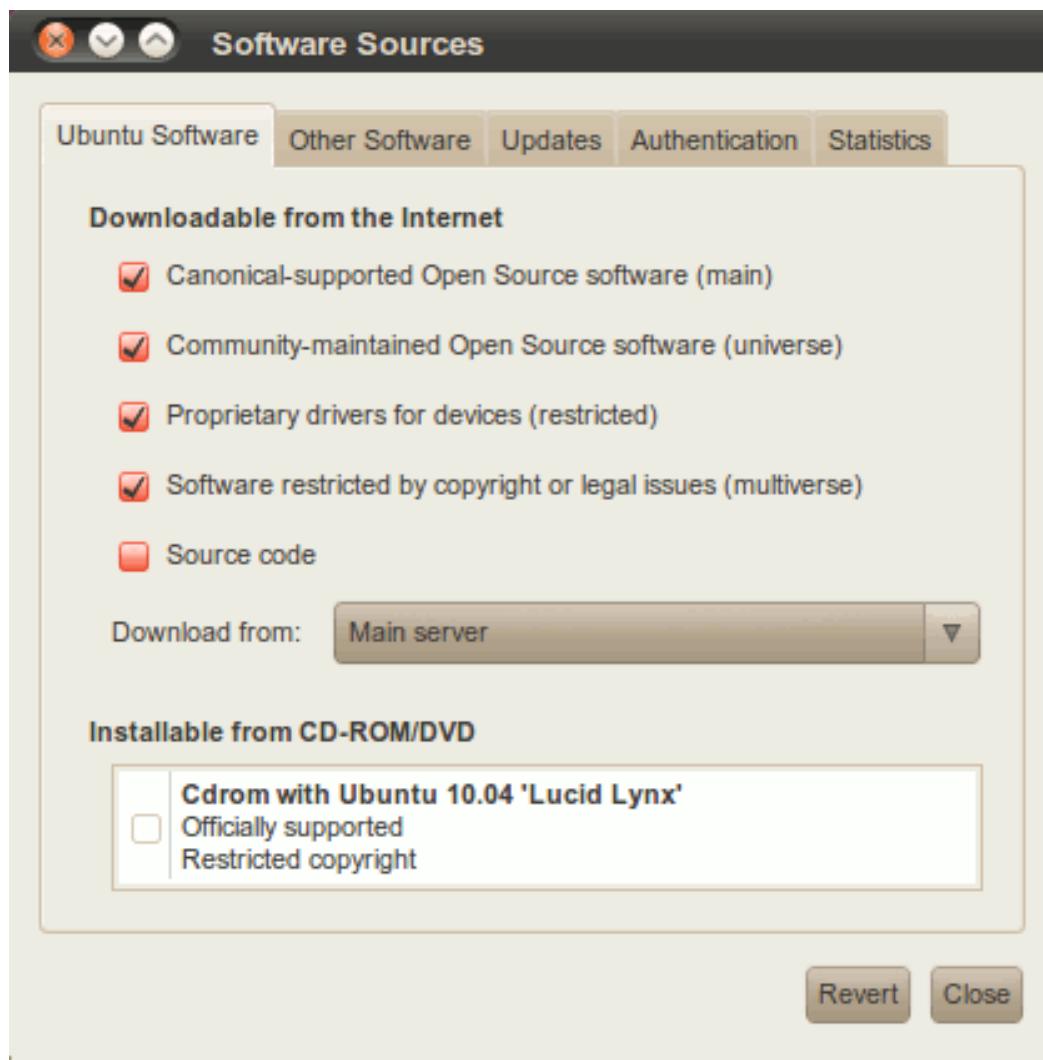


Figura 5.5: Ubuntu - Software Sources Panel

(En su lugar se puede ejecutar directamente «`source /opt/ros/fuerte/setup.bash`» para que los cambios no sean permanentes y desaparezcan al cargar de nuevo el entorno Linux.

Hecho esto, el entorno ROS, que es la base sobre la que se trabaja, deberá haber quedado completamente instalado y funcional.

2. En caso de no encontrarse instalado Python 2.6 o superior (y del mismo modo en caso de duda), es necesario obtener los paquetes adecuados que son necesarios para el funcionamiento del paquete, en este caso a través de apt-get:

```
# Si se actualizo recientemente con "sudo apt-get update" basta escribir:  
sudo apt-get install python2.6 python2.6-dev python-yaml
```

3. El driver del periférico de entrada deberá estar instalado de forma independiente de la odometría. Si no lo estuviera, a continuación se sugiere una posible instalación para el caso particular del Kinect con los drivers de *openni_kinect*. Eso se traduce en lo siguiente al utilizar apt-get para la instalación:

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install ros-fuerte-openni-kinect ros-fuerte-camera-drivers
```

4. Si, a pesar de la nueva tendencia a incluirlos, los paquetes relacionados con PCL no son instalados junto con la instalación inicial de ROS (o simplemente se ha realizado la instalación mínima), será necesario instalar concretamente los paquetes: *ros-fuerte-perception-pcl* y *ros-fuerte-pcl*.

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install ros-fuerte-perception-pcl ros-fuerte-pcl
```

5. Será necesario para el siguiente paso instalar un cliente git que está disponible mediante apt-get y que permitirá descargar y actualizar de forma fácil el código fuente de los paquetes.

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install git
```

6. Tras completar la instalación de todos los paquetes de dependencias gracias a los pasos anteriores, llega el paso central del proceso que es instalar el propio software de odometría visual desde su repositorio⁶. Se instalarán los archivos en una carpeta nueva de proyectos en vez de utilizar las carpetas ROS existentes, de manera que se muestre así un ejemplo más completo.

```
cd /home/usr_name # Nombre de usuario "usr_name" como ejemplo
sudo git clone https://github.com/michi05/my_odometry
cd my_odometry
rosmake
```

5.3.3. Puesta en marcha y detención de la odometría

Siguiendo los mismos principios que en el desarrollo de la interfaz, se trató de realizar un componente muy simple que, sobre todo, realizara funciones básicas y sirviera como base funcional para otros desarrollos, en vez de desarrollar un software excesivamente completo y complicado. Por ese motivo, y tal como ocurre con la interfaz, toda la configuración se realiza antes del arranque (se explicará en la subsección de configuración) mediante parámetros ROS y/o ficheros de lanzamiento. Sin embargo, en este caso, no es imprescindible ningún fichero de lanzamiento que invoque nodos complementarios ni que realice la configuración, sino que se puede lanzar por sí solo tras cargar la configuración adecuada.

Dicho lo anterior sólo queda especificar que la combinación de teclas para la terminación es *Ctrl+C* y el comando de arranque « *rosrun my_odometry pcl_to_tf* » siempre y cuando se haya lanzado ya el núcleo de ros con « *roscore* & » que al utilizar *roslaunch* se ejecuta automáticamente.

5.3.4. Configuración de la odometría

Tal como se adelantaba, es necesaria una configuración adecuada para el funcionamiento correcto de la odometría, pero esta debe ser introducida de antemano en el servidor de parámetros. Para ello existen tres métodos igual de válidos y que son ofrecidos por el propio entorno ROS, a saber:

⁶El código se guarda bajo la siguiente dirección: “ https://github.com/michi05/my_odometry ”

El uso de ficheros de lanzamiento. Tal como se explica en la sección correspondiente (5.4), es posible utilizar ficheros de lanzamiento. En este caso bastaría con un fichero mínimo que lance un solo nodo y sobre el cual se pueda añadir la configuración a gusto del usuario:

```
<launch>
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" >
    <!-- Aquí los distintos parametros y sus valores -->
  </node>
</launch>
```

Cada parámetro de configuración se establecería con una nueva línea « <param name="nombre_parametro" value="nuevo_valor_cadena" /> » o, si se desea, con argumentos que pueden ser especificados durante la llamada:

```
<arg name="nombre_argumento" default="nuevo_valor_cadena" />
<arg name="velocidad_arg" default="5" />
<launch>
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" >

    <param name="nombre_parametro" value="$(arg_nombre_argumento)" />
    <param name="velocidad" value="$(arg_velocidad)" />
    <param name="modo" value="auto" />

  </node>
</launch>
```

La etiqueta “default” es opcional. Si se elimina, el argumento pasa a ser obligatorio para poder realizar el lanzamiento. Dicho argumento se especifica en la llamada inicial usando el operador “dos puntos-igual” (:=) de la siguiente manera: « *roslaunch* pckg file velocidad:= 3 nombre_argumento :=valor_argumento ».

La introducción manual de cada valor mediante el comando rosparam. Es un método más simple pero menos práctico dado que introduce los valores directamente en el servidor de parámetros que va a ser borrado al reiniciar el sistema. Consiste en introducir manual e individualmente cada parámetro y su valor con el comando « *rosparam* set /ruta/nombre_nodo/nombre_parametro valor », en el que la ruta puede variar, pero acabando con el nombre del nodo debido a que los parámetros se encuentran en la zona privada del mismo.

Cargar los valores desde un fichero de parámetros guardado en el disco. Se trata de una opción intermedia que ofrece el propio entorno ROS mediante el comando « *rosparam* load ». Para ello es necesario tener un archivo con los parámetros; ya sea escrito por el usuario (en forma de diccionario con lenguaje YAML) o bien generado mediante « *rosparam* dump nombre_fichero_params ». Una vez creado, bastará con ejecutar « *rosparam* load nombre_fichero_params » cada vez que se cargue el núcleo de ROS (*roscore*) y antes de ejecutar el nodo que hará uso de los parámetros.

5.3.5. Utilización de las funciones básicas.

A partir de este punto se procederá a ilustrar las acciones que debe efectuar el lector para hacer uso de las distintas funcionalidades ofrecidas por el nodo. Se comienza por la configuración previa al arranque, que define principalmente el modo de funcionamiento (automático o manual) y se seguirá con las acciones básicas que se pueden realizar.

Configurar comportamiento

El nodo de odometría visual sigue una estructura fija específica en cuanto a su configuración y funcionamiento de tal manera que su comportamiento puede adaptarse levemente mediante parámetros. Los dos parámetros más esenciales son “odometry_manual_mode” y “odometry_ignore_time”.

El primero es un valor booleano para indicar que las nubes de puntos serán recibidas sólo cuando se soliciten; mientras que, en caso contrario, se leerán y procesarán todas las que sea posible adquiriendo una nueva tras cada estimación. El otro parámetro, “odometry_ignore_time” necesita ser activado en el caso de que se utilicen nubes de puntos grabadas con anterioridad o una fuente cuyas marcas de tiempo no sean fiables o coherentes; de lo contrario, los mensajes pueden ser desestimados al ser considerados “viejos”.

Relación de topics a disposición del usuario

A modo de canal de comunicación continua, en el nodo de odometría se utiliza un pequeño grupo de topics para ciertas comunicaciones esenciales como son enviar los resultados de la odometría y recibir las nubes de puntos. Dichos topics pueden cambiar de nombre o reubicarse en nuevos espacios de nombres pero para tener una idea básica de su funcionamiento, en la tabla 5.1 se expone un listado con las direcciones por defecto de los topics; los parámetros configurables que permiten modificarlas y su utilidad.

Procesar nubes de puntos

Una vez establecidos los parámetros según los puntos anteriores, existen dos formas de empezar a generar nuevas medidas:

- **Solicitando cada nueva medida desde el modo manual.**

Mediante el uso del servicio “updateOdometry”, se obtendrá con cada llamada del usuario, el cálculo del movimiento que define la relación entre la siguiente nube que el nodo se capaz de obtener, y la última conocida. En el caso de la primera llamada se espera una captura y se devuelve un mensaje indicando la disposición para la siguiente llamada.

El servicio responde con un solo mensaje que incluye el último movimiento realizado, los movimientos acumulados (que denotan la posición) y un código que define el estado actual del algoritmo, todo ello repartido en los campos correspondientes de una estructura de datos de ROS. A continuación un ejemplo de llamada sin movimiento:

Parámetro	Topic	Utilidad
topic_input_str	inputStr	Recibe la dirección de un topic entendido como fuente de nubes de puntos. Adquiere una nube de puntos y olvida de inmediato la dirección recibida.
topic_input_pcl	inputPCL	Si está a la espera, recibe mensajes en forma de nubes de puntos que introduce inmediatamente al algoritmo.
topic_camera_tf	input_camera_tf	Permite enviar al nodo la posición y orientación de la cámara con respecto al robot. Los datos deben llegar en forma de TF “tf/tfmessage” de ROS.
global_odometry_out_topic	globalOdometry	Topic de salida en el que se publican la posición y orientación acumuladas del robot.
relative_odometry_out_topic	lastRelativeOdometry	Topic de salida en el que se publica la TF del último movimiento del robot.
topic_aligned_cloud	aligned_cloud	Publicación de la nube de puntos alineada. Si la estimación es correcta, la nube alineada quedará superpuesta a la tomada desde la posición final.
topic_initial_cloud	initial_cloud	Publicación de la nube de puntos obtenida desde la posición inicial.
topic_filtered_cloud	final_filtered_cloud	Publicación de la nube de puntos obtenida desde la posición final.
topic_odometry_answer	odometry_answer	Publicación del mensaje de odometría con la estimación definitiva del último movimiento, de la posición y orientación actuales y el estado actual del algoritmo.

Tabla 5.1: Listado de topics de comunicación con el nodo de odometría.

```
$ rosservice call /pcl_to_tf/updateOdometry
>
answer:
globalTF:
transforms:
-
  header:
    seq: 0
    stamp:
      secs: 0
      nsecs: 0
    frame_id: my_odom_tf
    child_frame_id: base_link
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 0.0
...
statusCode: 3
```

- **Enviando nubes de puntos al topic correspondiente en modo automático.**

En el modo automático, el nodo espera hasta recibir dos nubes de puntos válidas y tratará de calcular la transformación que las relaciona en función del movimiento realizado entre las dos capturas. Una vez obtenida una medida, se desprecia la primera nube y se espera una nueva para volver a repetir sucesivamente la operación para cada par consecutivo. De este modo se procesa el máximo posible de capturas según las limitaciones de procesamiento o recepción, obteniendo.

En este caso es posible observar los resultados monitorizados, mediante el comando “rostopic”, a través del topic “odometry_main/odometry_answer” que publica en cada mensaje la odometría del último movimiento y el acumulado globalmente. Debajo se muestra un ejemplo de la llamada realizada, si bien la respuesta se omite pues se mostraría con aspecto idéntico al ejemplo anterior.

```
# Es importante poner el espacio de nombres, pero en este caso es la raíz: /
$ rostopic pub /odometry\_main/odometry\_answer
```

Resetear estimaciones. Puesta a cero

En caso de que se desee comenzar de nuevo con las mediciones de odometría, o se quiera borrar la posición estimada porque acumula ya demasiado error, se debe utilizar el servicio “resetGlobals”. El reseteo no requiere de ningún argumento para funcionar y retorna el mensaje de odometría estándar del nodo, es decir, incluyendo la odometría global y local y el estado del algoritmo para comprobar que todo ello ha quedado como se espera.

```
$ rosservice call /odometry\_main/resetGlobals
```

Asignar una nueva TF a la cámara

En caso de que la cámara de profundidad no sea el centro del sistema de referencia cuyo movimiento desea estimarse, como en el ejemplo de la figura 5.6, existe la opción de proporcionar su posición relativa a este para que el nodo calcule el movimiento del sistema completo.

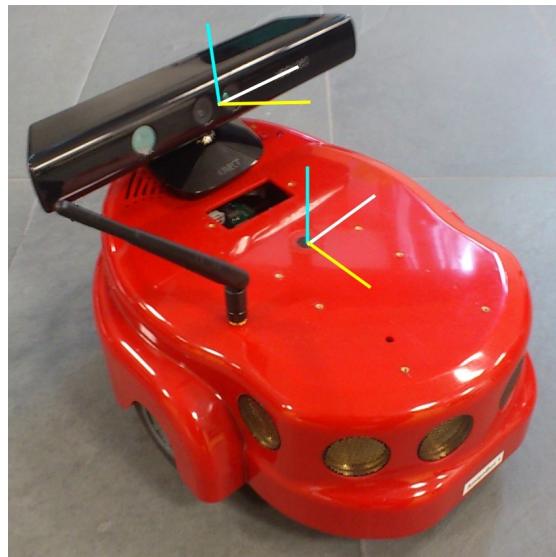


Figura 5.6: Robot amigobot con el MS Kinect y sus dos sistemas de referencia

Se trata de publicar un mensaje con los valores de posición y orientación de la cámara tomando como referencia la posición cero y la orientación consideradas para el robot. Para ello es necesario escribir un fichero con los valores adecuados asignados a los distintos campos del tipo ROS “tf::transform” usando lenguaje YAML. Se propone utilizar el fichero “camera_fixed_tf.yaml” como modelo, que se encuentra disponible en el directorio raíz del código y se trata de un fichero idéntico al que se desea construir pero con valores nulos:

```

transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: my_odom_tf
    child_frame_id: base_link
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 0.0
  
```

En el ejemplo siguiente se observa cómo publicar el fichero “camera_fixed_tf.yaml”, una vez prepara-

do, a través del topic correspondiente “input_camera_tf”. Se utiliza el comando “rostopic pub”⁷ especificando, necesariamente, el tipo de mensaje publicado “tf/tfMessage”, mientras que el modificador “–once” determina que sólo se realizará una publicación, frente a la opción por defecto de repetir el mensaje indefinidamente hasta cerrar la herramienta mediante la combinación de teclas “Ctrl + C”.

```
rostopic pub —once —f camera_fixed_tf.yaml /odometry_main/input_camera_tf tf/tfMessage
```

⁷La documentación completa sobre “rostopic pub” se puede consultar en “http://mediabox.grasp.upenn.edu/roswiki/rostopic.html#rostopic_pub”

5.4. Creación de ficheros de lanzamiento personalizados

Si bien este apartado no se refiere a una parte del proyecto sino a una herramienta que pertenece al entorno ROS, todo el desarrollo está orientado a ser compatible con los ficheros de lanzamiento por su extrema utilidad e importancia. Son una parte irrenunciable del sistema y una pieza más de la solución que simplifica al máximo el lanzamiento y arranque de nodos y grupos de nodos incluyendo configuraciones preestablecidas y también parametrizadas según las necesidades del usuario.

Los tutoriales completos y oficiales de cómo sacarle el máximo partido a dichos ficheros se encuentran en la página web de ROS⁸, por lo que sólo corresponde al presente texto una pequeña introducción que ilustre cómo incluir los nodos del proyecto en un fichero de lanzamiento junto con sus parámetros de configuración. De los susodichos se extrae el siguiente ejemplo mínimo de lanzamiento de un nodo:

```
<launch>
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" />
</launch>
```

Para completarlo es posible agregar otros nodos que se quieran lanzar (como controladores de dispositivos) o incluso una jerarquía de ficheros de lanzamiento según las necesidades del usuario. Se utiliza la etiqueta “node” en el primer caso e “include” en el segundo. En el siguiente ejemplo (no funcional) se lanza primero el driver secundario (“kinect_aux”) del kinect y después se incluye el lanzador de *my_adaptor*, dejando el nodo de *my_odometry* para el final.

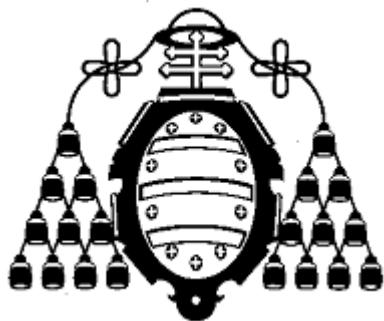
```
<launch>
  <node name="kinect_aux_node" pkg="kinect_aux" type="kinect_aux_node" />
  <include file="robotDrivers.launch" />
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" />
</launch>
```

En el ejemplo anterior falta una parte importante de los ficheros de lanzamiento sobre configurar los parámetros de los nodos. Se trata de la etiqueta “arg” que realiza una carga de valores en el servidor de parámetros, que a su vez conformarán la configuración de los nodos. Su uso es tan simple como añadir una línea por cada parámetro junto con su nombre tras el modificador “name” y su valor entre comillas usando “value”.

⁸El uso de los ficheros de lanzamiento se puede encontrar en: “<http://ros.org/wiki/rosLaunch>”

```
<launch>
  <include file="included.launch">
    <!-- las variables que necesita included.launch deben estar definidas -->
    <arg name="hoge" value="fuga" />
  </include>
</launch>
```

En todos los casos se han omitido los atributos opcionales y las etiquetas innecesarias para limitar esta sección a lo básico. Para más información al respecto se puede consultar la documentación oficial mencionada al principio de este apartado.



UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE
GIJÓN

**ÁREA DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL**

PROYECTO FIN DE CARRERA N° 3122067

**INTEGRACIÓN DEL DISPOSITIVO 'MICROSOFT KINECT'
COMO PARTE DE UNA ARQUITECTURA ROBÓTICA
GENÉRICA PARA SU USO COMO DISPOSITIVO DE ENTRADA**

DOCUMENTO N° VI

MANUAL TÉCNICO



MIGUEL A. GARCÍA GONZÁLEZ

NOVIEMBRE – 2012

TUTOR: LUCIANO SÁNCHEZ RAMOS

CAPÍTULO 6

MANUAL TÉCNICO

6.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

6.1.1. Identificación del proyecto

Título:	“Integración del dispositivo ‘Microsoft Kinect’ como parte de una arquitectura robótica genérica”
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

6.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo “Microsoft Kinect” comercializado por la empresa “Microsoft Corporation”. La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

6.1.3. Visión general del documento

El documento de manuales técnicos ha sido concebido con el objetivo principal de aportar toda la información necesaria para guiar a futuros desarrolladores que deseen reutilizar, corregir o adaptar el código de la manera que le sea necesario. Se trata de un aspecto importante dado que se solicitó expresamente al inicio del proyecto que el código estuviera orientado un mantenimiento sencillo mediante el uso de clases con responsabilidades claras y separadas, además de las indispensables aclaraciones mediante comentarios en el código fuente.

Este documento puede verse, de alguna manera, complementado con el documento 4 sobre el *Diseño e Implementación* donde se introduce la estructura de la solución tanto a nivel de clases como de ficheros y de comunicaciones con el entorno. Por ese motivo se repetirán aquí algunos de los conceptos y diagramas más importantes con el objetivo de que el presente texto sea autosuficiente para su objetivo, pero sin perjuicio de que una lectura del primero podría aportar una mejor comprensión del conjunto.

Se comenzará por enumerar la relación de recursos necesarios para la modificación del código fuente y su puesta en marcha en forma de producto ejecutable, siguiendo con su estructura de clases y una explicación más en profundidad de cada clase individual. Finalmente, se hablará de los ficheros en que se divide el código para terminar detallando la capacidad de configuración implementada para los nodos.

6.2. Recursos necesarios

Para el mantenimiento del código no se necesitan grandes medios pero sí un PC con el software adecuado y unos determinados conocimientos de programación. A continuación se expone una relación de los recursos que deberán ser suficientes para dichas modificaciones.

Recursos hardware

Un PC de características básicas es suficiente para realizar cualquier modificación al código de la interfaz y generar la solución correspondiente. Se obviarán en este apartado los dispositivos físicos requeridos para las pruebas y para el uso del software, pues no son requisitos y dependen plenamente de necesidades puntuales. El equipo que se toma como referencia es un portátil “Asus eeePC”, el más básico de los disponibles en el laboratorio.

- Procesador Intel Atom -Silverthorne- (1.3 GHz) o superior
- 1 GiB de Memoria RAM
- 10 GiB de espacio en el disco duro
- Teclado y ratón

Recursos software

Toma mayor importancia en este caso el entorno software que el hardware en el que se instala. Se necesitará, por supuesto, el entorno de destino (un sistema ROS dentro de Ubuntu Linux), así como el compilador del lenguaje (Python, en este caso) y una aplicación que permita modificar ficheros de texto plano. Una vez más se excluyen del listado los paquetes relacionados con el dispositivo final al no ser requisitos para la modificación del código en sí.

- Ubuntu Linux 10.04 (Lucid Lynx) o superior
- Robot Operating System (ROS) versión *Fuerte*
- Paquetes de Linux:
 - python2.6
 - python2.6-dev
 - **Sólo para la interfaz** Librería de YAML para Python: “python-yaml”
 - **Sólo para la odometría** Paquete ROS de la librería de nubes de puntos (*PCL*): “ros-fuerte-perception-pcl”¹

¹Referencia de perception_pcl: “<http://www.ros.org/wiki/pcl>”

- **Sólo para pruebas con MS Kinect** Paquete ROS de los controladores *OpenNI*: “ros-fuerte-openni-kinect”²
- Y un editor de texto para el código como pueden ser por ejemplo:
 - Editor de texto básico como *gedit text editor*. Instalado por defecto con Ubuntu.
 - Entorno de desarrollo integrado como *Eclipse IDE*. Descargable desde su web oficial “ www.eclipse.org ”

Todos los paquetes mencionados tanto de Ubuntu como de ROS están disponibles en los repositorios de *Ubuntu* y el propio sistema operativo puede ser tanto descargado, como solicitado por correo postal desde la página web “ <http://www.ubuntu.com/> ”.

Recursos humanos

También son importantes los conocimientos necesarios para el mantenimiento y modificación del código. Siempre es conveniente tener conocimientos amplios y abundantes de todas las herramientas que intervienen, pero se enumeran a continuación los campos importantes que podrían ser estudiados para adquirir una buena comprensión del código y del entorno previamente a realizar modificaciones en el primero.

- **Robot Operating System** Es necesario poseer conocimientos sobre el funcionamiento de ROS y algunas de sus herramientas para comprender y adaptar el comportamiento de la aplicación, del propio entorno y, sobre todo, de la comunicación entre ellos. Se trata de un sistema modular con una colección de herramientas de las cuales se destacan las más indispensables para el caso actual:
 - Ficheros de lanzamiento ROS - “ <http://ros.org/wiki/roslaunch> ”
 - Servidor de parámetros - “ www.ros.org/wiki/ParameterServer ”
 - Canales de comunicación: topics y servicios - “ www.ros.org/wiki/Topics ” y “ www.ros.org/wiki/Services ”
 - Y adicionalmente, herramientas de prueba y diagnóstico: rosrun, rostest, etc.
- **Ficheros de lanzamiento ROS** Si bien este punto forma parte del anterior, al tratarse de una herramienta de ROS, se destaca específicamente por ser una herramienta opcional en el entorno e imprescindible en el presente proyecto. Especialmente en el caso de la interfaz, los ficheros de lanzamiento permiten secuenciar el lanzamiento de nodos para garantizar que los controladores están en marcha.
- **Git revision control** Es necesario manejar el sistema de control de versiones (o revisiones) utilizado (*Git*) para obtener el código inicial y, muy especialmente, si se pretende que el repositorio actual refleje los cambios futuros.

²Referencia de openni_kinect: “ http://www.ros.org/wiki/openni_kinect ”

Sólo para el paquete interfaz

- **Lenguaje Python** El código está escrito en lenguaje Python. No se utilizan operaciones ni estructuras de gran complejidad en el código de la interfaz, por lo que tener unas nociones básicas será suficiente para poder realizar ciertas modificaciones.
- **Lenguaje YAML** YAML es un lenguaje de marcado que es el utilizado tanto en el diccionario de parámetros (5.2.4), como en los ficheros generados por el servidor de parámetros de ROS.

Sólo para el paquete odometría

- **Lenguaje C++** Dado que en el paquete sobre odometría la eficiencia era uno de los aspectos más importantes, ésta fue implementada en lenguaje C++ y es, por tanto, necesario comprender dicho lenguaje para poder realizar futuras modificaciones del nodo. Es, además, conveniente que el desarrollador esté familiarizado con la programación orientada a objetos dado que el código trata de seguir dicho paradigma.
- **Librería de nubes de puntos (PCL)** Las estimaciones de odometría visual corren a cargo de la librería de nubes de puntos (conocida como *PCL*) que implementa el algoritmo un algoritmo *ICP* (por sus siglas en inglés “*Iterative Closest Point*” y que es explicado en el anexo) para la correlación de nubes de puntos y algunas herramientas auxiliares para su tratamiento. Toda la información sobre esta librería se encuentra en su página web “ <http://pointclouds.org> ” e incluye otras herramientas y algoritmos que podrían ser útiles para futuras ampliaciones del proyecto actual.

6.3. Instalación del entorno de programación

Una vez se han especificado los recursos software necesarios, el siguiente paso es instalarlos en un ordenador que se utilizará para las modificaciones. Dado que Python es un lenguaje semi interpretado y que, en todo caso, es necesario compilar el paquete una vez descargado, la instalación del *Manual de usuario de la interfaz* (5.2.2) es exactamente la misma que la de esta sección y el lector puede remitirse a dicho apartado para encontrarla.

6.4. Ficheros fuente

Es necesario especificar la forma en que se organiza el código fuente dentro del sistema de ficheros del ordenador, pues de ello puede depender dónde se encontrarán distintos elementos y aspectos del software o dónde deben introducirse otros nuevos. A continuación se analiza la estructura que forman los ficheros en su conjunto, lo que contienen y cómo interactúan entre ellos para formar el producto final.

El código fuente completo se encuentra almacenado y publicado en la web de “GitHub” que se encarga del sistema de control de versiones (siguiendo el sistema “Git” en particular) así como de permitir a otros usuarios obtener el código o tratar de contribuir si fuera necesario. Los espacios registrados a tal efecto se encuentran en “https://github.com/Michi05/my_adaptor” para el paquete de interfaz y “https://github.com/Michi05/my_odometry” para la odometría.

6.4.1. Estructura general

Dado que el proyecto está dividido en dos partes que representan dos paquetes software individuales y autónomos, será necesario estudiar cada uno por separado. Se utiliza, sin embargo, la estructura común que viene dada por el entorno ROS y que es explicada en detalle en su página de consulta “www.ros.org/wiki/Packages”. A continuación el listado de ficheros y directorios comunes:

- **bin/**: archivos ejecutables ya compilados.
- **cfg/**: directorio para la configuración del servidor dinámico de parámetros (sólo en la interfaz).
- **include/**: archivos de cabecera de C++ (.h, .hpp).
- **launch/**: *launch files* encargados de lanzar los nodos con sus correspondientes configuraciones y dependencias.
- **msg/**: Especificación de los tipos de datos asociados a los topics y servicios.
- **nodes/**: Archivos de código fuente específico para nodos. Se utilizará para los ficheros python de la interfaz.
- **scripts/**: Archivos de secuencias de comandos (no se utilizó).
- **src/**: Archivos de código fuente principal (.c, .cpp). Se eligió usar esto sólo en el segundo paquete (*my_odometry*).
- **srv/**: Tipos de datos específicos para los servicios (conjunto de tipos de mensajes agrupados en “solicitud” y “respuesta”).
- **CMakeLists.txt**: Configuración para “CMake” con los directorios y ficheros a incluir en la compilación.
- **manifest.xml**: Manifiesto del paquete (información sobre su estructura y dependencias).

Los tipos de mensajes y servicios imprescindibles para determinar qué información se podrá transmitir y en qué manera u orden. Los tipos de mensajes determinan estructuras de datos complejas formadas a partir de tipos básicos y/o de otros mensajes; mientras que los servicios agrupan estructuras de datos construidas del mismo modo pero asociadas a dos partes separadas: solicitud y respuesta.

Durante la compilación también se generan algunos directorios y ficheros adicionales que varían con la configuración y las herramientas del entorno utilizadas, pero que no es necesario conocer ni mucho menos es aconsejable modificar porque se trata de ficheros para uso propio del sistema.

6.4.2. Ficheros de la interfaz

El código fuente, a nivel de ficheros, se basa en dos elementos principales: “interface_node_main.py” y “ctrl_interface.py” que se encuentran en el directorio *nodes* y representan, respectivamente: la función de partida principal del nodo (función “*main*”) y la definición de las distintas clases que intervienen durante la ejecución. Intervienen además 4 carpetas más con ficheros de importancia:

- **cfg/** contiene el fichero “properties.cfg” que determina los parámetros conocidos por el servidor dinámico de parámetros. El modo de configurarlo se explica en la sección de configuraciones (6.7) y se profundiza en la documentación oficial de ROS³.
- **launch/** permite escribir ficheros de lanzamiento que realicen automáticamente una serie de tareas principalmente de configuración, ejecución del nodo principal y de las dependencias. Es una parte indispensable de la interfaz dado que tiene que ejecutar los controladores de los dispositivos antes de lanzarla, pero también se encarga de asignar un espacio de nombres común a los nodos dentro del entorno ROS.
 - **kinectDrivers.launch** es el fichero de lanzamiento básico para controlar el Kinect.
 - **adaptor_openni.launch** lanza sólo los drivers del Kinect. Es utilizado por los otros lanzadores.
 - **robotDrivers.launch** permite lanzar “kinectDrivers” dentro de un espacio de nombres y lleva más configuración.
 - **runMultiplexers.launch** ayuda a lanzar los multiplexadores de topics (utilidad interna para el adaptador).
- **srv/** guarda en su interior las especificaciones de los servicios que vienen definidos por dos mensajes o conjuntos de mensajes: de solicitud y de respuesta.
 - **Solicitudes de lectura para tipos de datos básicos:** booleanValue.srv, intValue.srv, floatValue.srv, stringValue.srv
 - **Solicitudes para modificar tipos de datos básicos:** setBoolean.srv, setInteger.srv, setFloat.srv, setString.srv
 - **Obtención de la dirección (topic) asociada a un parámetro:** requestTopic.srv

³Enlace: “http://www.ros.org/wiki/dynamic_reconfigure”

6.4.3. Ficheros de la odometría

El paquete de odometría visual, por su parte, está dividido en tres partes: una función *main* principal y dos clases que se implementan según el convenio habitual de código y cabecera por separado (con extensiones .cpp y .hpp respectivamente). Sus correspondientes ficheros se denominan “odometry_main.cpp”, “pcl_odometry.cpp” y “pcl_myTransf.cpp” junto con sus respectivas cabeceras con extensiones .hpp. También en este caso cabe mencionar algunas carpetas adicionales indispensables para las modificaciones:

- **launch/** permite escribir ficheros de lanzamiento que realicen automáticamente una serie de tareas principalmente de configuración, ejecución del nodo principal y de las dependencias. En el caso de la odometría se encarga de cargar la configuración guardada y se incluye un segundo lanzador que, sin ejecutar el nodo principal, publica una TF estándar prefijada para las simulaciones.
 - **my_odometry.launch** es una propuesta de lanzador básico que carga un fichero de configuración que puede ser distinta de la configuración por defecto del nodo y pone en marcha el nodo principal.
 - **tf_publisher.launch** lanza una herramienta ROS de publicación de TFs de referencia útiles para las simulaciones y visualizaciones.
- **msg/** es el directorio donde se almacenan las especificaciones de “mensajes” que, como ya se adelantó, son los tipos o estructuras de datos que se pueden transmitir a través de los canales ROS: topics y servicios.
 - *odom_answer.msg*
- **srv/** guarda en su interior las especificaciones de los servicios que vienen definidos por dos mensajes o conjuntos de mensajes: de solicitud y de respuesta.
 - **emptyRequest.srv** es necesario para los servicios que no requieren parámetros, con la particularidad de que incorpora en la respuesta los datos de estado del nodo, es decir, un mensaje de tipo *odom_answer*.
 - **odom_update_srv.srv** si bien se trata de un servicio idéntico a “emptyRequest” y podría prescindirse de él, se mantiene este servicio por separado

6.5. Estructura de clases de la interfaz

La interfaz se diseñó, en aras de favorecer la mantenibilidad, con un número de clases bajo como para favorecer la simplicidad pero suficiente para separar las responsabilidades, en este caso, organizadas en forma de capas de abstracción. Esto se traduce, concretamente, en tres clases complementarias encargadas de distintas partes de la comunicación: la más baja (*driver_manager*) es la que se comunica directamente con el driver del dispositivo, la capa de arriba, *upward_interface* es accesible por parte del usuario o por las aplicaciones que hacen uso de la interfaz, y ambas se comunican entre ellas si bien es la tercera clase *iface_translator* la que permite que se entiendan al traducir los “nombres” de los parámetros entre los conocidos por el driver del dispositivo y los especificados por el usuario en el fichero de configuración.

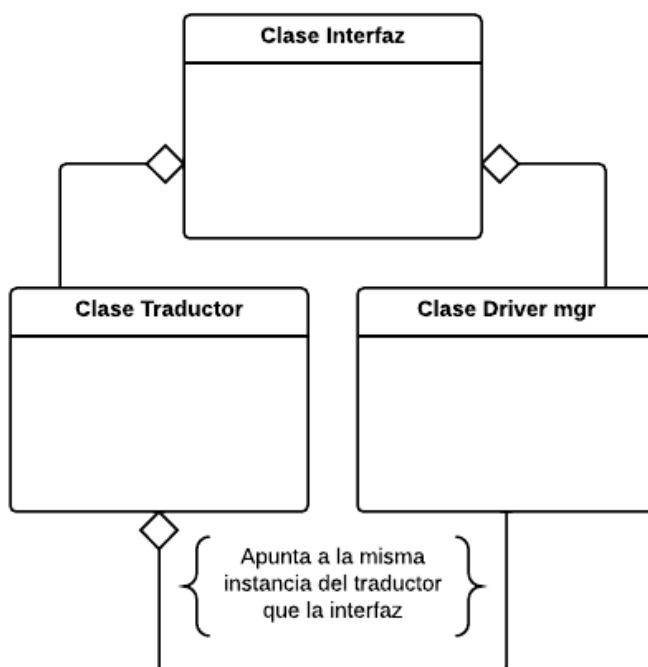


Figura 6.1: Diagrama de clases que incluye las de la interfaz

En la figura 6.1 se representa gráficamente la estructura que forman entre sí las clases del nodo y sus relaciones. En los apartados siguientes se profundiza sobre los datos miembro que incorpora cada una de ellas y se describe la utilidad de cada método que implementan usando el convenio de documentación de “Python Docstring” de escribir “@param” para especificar los parámetros de entrada y “@return” antes de exponer el contenido de la variable de retorno.

La instanciación de estas clases y el modo de inicializarlas durante el arranque forman parte de la explicación sobre el *Código fuente del producto* dentro del documento de *Diseño* (4.6).

6.5.1. Clases del paquete interfaz

En los siguientes apartados, que corresponden a las tres clases que componen la interfaz, se dedicará en cada caso: un punto a especificar los datos miembro que almacena cada una de ellas y otro a las funciones miembro que implementan en cada caso. Las explicaciones giran en torno un elemento principal, que son los parámetros configurables de los controladores; y cuatro conceptos básicos. Los dos primeros conceptos son los niveles de abstracción alto y bajo, que designan respectivamente lo que debe “conocer” el agente que usa la interfaz (usuario o aplicación) y lo que “entiende” el driver. Les siguen los “nombres locales” o “alias” de los parámetros, que son los identificadores que se utilizan para designar los parámetros desde el punto de vista del usuario (alto nivel). Finalmente, los nombres “remotos” o “a nivel de driver” se utilizarán sólo en operaciones de bajo nivel, para comunicarse con el controlador. Salvar el obstáculo entre los alias y los nombres a nivel de driver de forma transparente es el objetivo más importante del presente nodo.

Clase *upward_interface*

La denominada *upward_interface* es la clase que representa la capa más alta de abstracción dentro del código, es decir, orientada a la comunicación con el agente (aplicación o usuario) que quiere comunicarse con el controlador. Es importante advertir que esta clase contiene referencias a las otras dos para evitar las comunicaciones globales entre clases, lo cual no responde intuitivamente al diseño conceptual pero sí lo hace su comportamiento o funcionamiento resultante.

Su trabajo consiste en habilitar canales de comunicación ROS en forma de servicios y servidor de parámetros para recibir y transmitir modificaciones de parámetros tanto en dirección al driver por parte de un nodo externo, como en el sentido contrario en caso de que se producan cambios en la capa inferior que la interfaz interpretará como un evento para propagar dicho cambio.

Datos Miembro

- **listOf_services** es un dato estático de tipo lista en el que se almacenan las instancias de los servicios publicados para recibir peticiones externas. En caso de futuras implementaciones que permitan varias instancias de *upward_interface*, los servicios seguirían siendo únicos al almacenarse en la misma lista.
- **translator** almacena una referencia a la instancia de la clase *translator* que debe ser utilizada para las traducciones.
- **driverMgr** contiene una referencia a la instancia de la clase *driver_manager* que se usará para la comunicación con el controlador.
- **avoidRemoteReconf** contabiliza el número de cambios recibidos desde el driver que aún quedan por tratar, reduciéndose de nuevo hasta '0' tras tratarlos. Si el cambio es local, el contador no aumenta y los cambios son propagados hacia abajo. Su objetivo es evitar que los cambios se propaguen de nuevo hacia el controlador formando un bucle descontrolado de actualizaciones.
- **dynServer** es la instancia del *servidor dinámico de parámetros* del nodo interfaz. Una vez iniciado podrá ser leído desde nodos externos o recibir actualizaciones de estos.

Funciones Miembro

■ **dynServerCallback(self, dynConfiguration, levelCode)**

Manejador para los eventos del *Servidor de Reconfiguración Dinámica*. Retorna una configuración de parámetros en el mismo formato que la recibe (un diccionario.).

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param dynConfiguration - un diccionario {parámetro:valor} con los cambios a realizar. Precondición: el diccionario se refiere a los parámetros según sus nombres LOCALES.

@param levelCode - el código que se genera durante el *callback* como marca del grupo de parámetros que han cambiado.

@return El diccionario con la configuración final resultante (obligado para todos los callback).

■ **getAnyProperty(self, propertyName)**

Método genérico para obtener el valor de cualquier parámetro tras llamar al método específico correspondiente. Necesita el nombre LOCAL del parámetro y responde con el valor remoto correspondiente.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param propertyName - el nombre LOCAL del parámetro a leer.

@return el valor leído o *None* si hubo algún error.

■ **getDispImage(self, srvMsg)**

Método para obtener una secuencia de imágenes de disparidad (“Disparity Images”) cuando es llamado desde un servicio.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param srvMsg - una estructura de servicio incluyendo:

-int64 nImages - número de imágenes a retransmitir.

-string sourceTopic - topic original del cual obtener las imágenes.

-string responseTopic - dirección del topic al que enviar las imágenes.

@return las imágenes leídas de la fuente.

■ **getFixedTypeProperty(self, localPropName, valueType)**

Método principal para obtener valores de parámetros conociendo ya el nombre y tipo de dato para devolver el valor actual en el driver.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param localPropName - el nombre LOCAL del parámetro a leer.

@param valueType - el tipo de dato esperado del parámetro a leer.

@return el valor leído o *None* si hubo algún error.

▪ `getImage(self, srvMsg)`

Método para obtener una secuencia de imágenes cuando es llamado desde un servicio.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param srvMsg - una estructura de servicio incluyendo:

-int64 nImages - número de imágenes a retransmitir.

-string sourceTopic - topic original del cual obtener las imágenes.

-string responseTopic - dirección del topic al que enviar las imágenes.

@return las imágenes leídas de la fuente.

▪ `getStringProperty(self, srvMsg)`

Método genérico para obtener valores de tipo cadena que llama al método “getFixedTypeProperty” con los valores adecuados en función del tipo y canal al que corresponde el parámetro.

@param srvMsg - una estructura de servicio residual al hacer la llamada pero que es ignorada.

@return la respuesta inmediata del método “getFixedTypeProperty” al que llama.

▪ `getIntProperty(self, srvMsg)`

Método análogo a getStringProperty para datos de tipo entero.

▪ `getFloatProperty(self, srvMsg)`

Método análogo a getStringProperty para datos de tipo real.

▪ `getBoolProperty(self, srvMsg)`

Método análogo a getStringProperty para datos de tipo booleano.

▪ `getTopicLocation(self, getStrMsg)`

Obtiene el topic de un parámetro dentro del entorno ROS. Precondición: el parámetro tiene que estar publicado como topic.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param getStrMsg - una estructura incluyendo:

- string topicName - the name of the topic which location is to be returned.

- string topicValue - espacio para el valor de retorno en la respuesta.

@return un mensaje de texto describiendo si hubo éxito o describiendo un posible error.

▪ `get_property_list(self)`

Método auxiliar para obtener la lista de parámetros conocidos por el traductor.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@return una lista completa de parámetros del traductor.

▪ `listenToRequests(self)`

Método para poner en marcha los servicios principales que estarán a la espera de solicitudes. No recibe nada y sólo devuelve éxito por si pudiera necesitarse más adelante.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@return Verdadero si se completa. En caso contrario propaga una excepción en función del problema.

▪ `publishDispImages(self, srvMsg)`

Método específico para retransmitir una secuencia de imágenes de disparidad (“Disparity Images”) cuando es llamado desde un servicio. Retransmite la fuente especificada hacia el topic que se solicite en el parámetro correspondiente.

Specific purpose method meant to retransmit, when asked via-service, a DISPARITY image topic through a topic path receiving the original path, the new path and the amount of messages.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param srvMsg - una estructura de servicio incluyendo:

-int64 nImages - número de imágenes a retransmitir.

-string sourceTopic - topic original del cual obtener las imágenes.

-string responseTopic - dirección del topic al que enviar las imágenes.

@return un mensaje de texto describiendo si hubo éxito o describiendo un posible error.

▪ `publishImages(self, srvMsg)`

Método específico para retransmitir una secuencia de imágenes cuando es llamado desde un servicio. Retransmite la fuente especificada hacia el topic que se solicite en el parámetro correspondiente.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param srvMsg - una estructura de servicio incluyendo:

-int64 nImages - número de imágenes a retransmitir.

-string sourceTopic - topic original del cual obtener las imágenes.

-string responseTopic - dirección del topic al que enviar las imágenes.

@return un mensaje de texto describiendo si hubo éxito o no.

■ `setAnyProperty(self, propertyName, newValue)`

Método genérico para modificar el valor de cualquier parámetro tras llamar al método específico correspondiente. Necesita el nombre LOCAL del parámetro y actualiza el valor remoto correspondiente.

@param self - la instancia cuyo método es llamado (común en los métodos Python)
@param propertyName - el nombre LOCAL del parámetro a modificar.
@param newValue - el nuevo valor para asignar al parámetro.
@return Verdadero si hubo éxito; Falso si no.

■ `setFixedTypeProperty(self, localPropName, newValue, valueType)`

Método principal para modificar valores de parámetros conociendo ya el nombre y tipo de dato e informar de si éste fue cambiado correctamente en el driver. Precondition: localPropName debe ser un nombre LOCAL conocido y el cambio se realiza en un servidor ajeno (otro nodo). No está diseñado para realizar cambios en el servidor propio mediante “loopback”.

@param self - la instancia cuyo método es llamado (común en los métodos Python)
@param localPropName - el nombre LOCAL del parámetro a modificar.
@param newValue - el nuevo valor para asignar al parámetro.
@param valueType - el tipo de dato esperado del parámetro a modificar.
@return Verdadero si hubo éxito; Falso si no.

■ `setStrProperty(self, setterMessage)`

Método genérico para modificar valores de tipo cadena que llama al método “setFixedTypeProperty” con los valores adecuados en función del tipo y canal al que corresponde el parámetro.

@param srvMsg - una estructura de servicio residual al hacer la llamada pero que es ignorada.
@return la respuesta inmediata del método “setFixedTypeProperty” al que llama.

■ `setIntProperty(self, setterMessage)`

Método análogo a setStrProperty para datos de tipo entero.

■ `setFloatProperty(self, setterMessage)`

Método análogo a setStrProperty para datos de tipo real.

■ `setBoolProperty(self, setterMessage)`

Método análogo a setStrProperty para datos de tipo booleano.

■ `setTopicLocation(self, setStrMsg)`

Establece una nueva ubicación para un topic existente dentro del entorno ROS.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param setStrMsg - una estructura de servicio incluyendo:

-string topicName - el nombre de parámetro cuyo topic se reubicará.

-string newValue - la nueva dirección donde reubicar el topic.

-string setAnswer - espacio para almacenar la respuesta correspondiente.

@return un mensaje de texto describiendo si hubo éxito o describiendo un posible error.

■ `updateSelfFromRemote(self, dynConfiguration)`

Este método se diseñó para “traducir” nombres locales en sus asociados remotos (los del driver) en caso de ser necesario previamente a utilizar el método “`updateSelfParameters`” (orientado sólo a los nombres LOCALES).

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param dynConfiguration - un diccionario parámetro:valor con los cambios a realizar. Precondición: el diccionario se refiere a los parámetros según sus nombres REMOTOS.

@return El diccionario con la configuración final resultante tras los cambios (obligado para todos los callback).

■ `updateSelfParameters(self, newConfig, avoidPropagation=False)`

This method is responsible for changing the node's own dynamic reconfigure parameters from its own code ***but avoiding a chain of uncontrolled callbacks!!.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param newConfig - un diccionario {parámetro:valor} con los cambios a realizar. Precondición: el diccionario se refiere a los parámetros según sus nombres LOCALES.

@param avoidPropagation - valor booleano que indica si es necesario o no evitar la propagación de los cambios hacia abajo (en caso de que estos ya provengan del controlador).

@return El diccionario con la configuración final resultante tras los cambios (obligado para todos los callback).

6.5.2. Clase *driver_manager*

La clase *driver_manager* contiene las funciones necesarias para comunicarse con la clase superior (*upward_interface*) y para enviar los posibles mensajes correspondientes al controlador de dispositivo, pero también los procedimientos necesarios para leer la configuración del driver y un método de callback para ser informado de sus cambios.

Desde dicha clase se pueden realizar conexiones a servidores de parámetros, llamadas a servicios y escuchar y publicar en topics ROS. En este nivel se utilizan los identificadores de parámetros tal y como son conocidos por el driver, en vez de sus alias. Para comunicarse con la clase superior se solicitan las traducciones de parámetros a la clase *iface_translator*.

Datos Miembro

- **dynSrvTimeout** es un valor que representa el tiempo, en milésimas de segundo, que se espera una conexión a un servicio antes de descartarla como fallida.
- **dynServers** es una lista con las direcciones de los servidores de parámetros remotos de los drivers en el entorno ROS.
- **createdMuxes** almacena, en una estructura de tipo diccionario, las reubicaciones de topics que se han realizado, con el objetivo de saber dónde se encuentran actualmente dichos topics.
- **paramServers** es un diccionario que tiene como clave las direcciones a los distintos servidores de parámetros externos y el valor almacenado son las conexiones ya realizadas para poder comunicarse con ellos durante la ejecución de la interfaz.
- **avoidSelfReconf** contabiliza el número de cambios realizados localmente que aún quedan por tratar, reduciéndose de nuevo hasta '0' tras tratarlos. Si el cambio viene del driver, el contador no aumenta y los cambios son aplicados localmente. Su objetivo es evitar que los cambios generen nuevos eventos formando un bucle descontrolado de actualizaciones.

Funciones Miembro

- **callService(service, arguments, valueType)**
Realiza llamadas a servicios de forma genérica para poder llamar independientemente del tipo de dato. No está en uso actualmente pero podría ser útil en el futuro.
 @param service - el nombre del servicio que será llamado.
 @param arguments - los argumentos que se pasarán durante la llamada al servicio.
 @param valueType - el tipo de dato de los mensajes solicitud/respuesta según los ficheros .srv correspondientes.
 @return Verdadero en caso de éxito.
- **dynClientCallback(self, dynConfiguration)**
Método de callback que responde cuando hay cambios en el servidor de parámetros remoto, es decir: que recibe nombres REMOTOS y sus valores en la variable "dynConfiguration".
Si el cambio fue iniciado por este mismo nodo, el valor de "avoidSelfReconf" será distinto de "0" y no se realiza ninguna acción. En otro caso los cambios son enviados al método "updateSelfFromRemote" para actualizar los valores en el servidor local.
 @param self - la instancia cuyo método es llamado (común en los métodos Python)
 @param dynConfiguration - un diccionario con la configuración a aplicar.

@return la configuración resultante tras la actualización. (Que puede no ser la esperada)

■ **getTopic(self, topicName, data_type)**

Obtener el valor actual de un topic.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param topicName - nombre del topic en el que leer el valor.

@param data_type - el tipo de datos esperado en el topic.

@return el valor obtenido (sin previa comprobación o validación).

■ **getValue(self, propName, dynServerPath)**

Obtiene el valor actual de un parámetro en el servidor de destino a partir de su nombre REMOTO.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param propName - nombre REMOTO del parámetro a leer.

@param dynServerPath - dirección del servidor que almacena el parámetro.

@return El valor obtenido si fue posible y “None” si no.

■ **relocateTopic(self, oldAddress, newAddress)**

Este método está diseñado para cambiar en tiempo de ejecución la ubicación de un topic a una nueva dirección. Dado que el entorno no permite esta acción propiamente dicha, se replica el topic con la herramienta “mux” .

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param oldAddress - la dirección del topic (ya existente) a reubicar.

@param newAddress - la nueva dirección en la que publicar el topic.

@return Falso si hubo algún error (si el primer carácter de la nueva dirección NO es una letra), verdadero si se realizó con éxito. Si hubo un error desconocido se lanza una excepción.

■ **sendByTopic(self, topic, value, data_type)**

Publica el valor dado una sola vez en el topic especificado y retorna. Si no hay conexiones a la escucha, el método espera hasta un segundo antes de realizar la publicación y sale en cualquier caso.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param topic - el nombre del topic en el que publicar el valor.

@param value - valor a publicar.

@param data_type - el tipo de datos que se espera publicar en el topic.

■ `setRemoteValue(self, rPropName, newValue, dynServerPath, ifcNodeInstance)`

Trata de asignar un valor dado a un parámetro especificado según su nombre REMOTO en el servidor de destino.

`@param self` - la instancia cuyo método es llamado (común en los métodos Python)

`@param rPropName` - nombre REMOTO del parámetro a modificar.

`@param newValue` - valor para asignar al parámetro.

`@param dynServerPath` - dirección del servidor remoto en el que modificar el parámetro.

`@param ifcNodeInstance` - el objeto “`upward_interface`” que permite cambiar los parámetros en el servidor local.

`@return` El valor obtenido si es posible o “`none`” si hubo algún problema

■ `topic_is_published(topicName)`

Método estático que comprueba si un topic está publicado actualmente en el entorno.

`@param topicName` - el nombre del topic cuya existencia será comprobada.

`@return` Verdadero en caso de éxito.

6.5.3. Clase *iface_translator*

Es la parte con menor complejidad pero no menos importante pues aunque con numerosas funciones auxiliares, destacan tres funciones principales, a saber: “`readYAMLConfig`”, para leer las traducciones desde el diccionario de parámetros (fichero de configuración); “`interpret`”, que traduce un alias de parámetro a su nombre de bajo nivel para que sea reconocido por el driver; y “`reverseInterpret`”, que a partir de un identificador de bajo nivel, retorna su alias. Esta clase será utilizada por las otras dos clases para que puedan comunicarse y entenderse a pesar de utilizar “nombres” distintos.

Datos Miembro

- **`property_config_file`** es el nombre del fichero de configuración, que es útil también como identificador único en caso de utilizar múltiples instancias de la clase de traducción.
- **`translations`** almacena una lista de cadenas de texto con las direcciones ROS de los drivers y otra indexada de forma análoga (el mismo índice corresponde al mismo driver) de diccionarios con los pares “nombre local” - “nombre remoto”.
- **`ReversePropDict`** es una estructura de tipo diccionario que tiene como claves los nombres remotos y como valor los nombres locales para poder realizar la traducción en el sentido contrario sin recorrer por completo la variable “`translations`”.

Funciones Miembro

■ **canGet(self, propertyName)**

Comprueba si el parámetro cuyo nombre recibe puede ser leído o no en función de si es conocido, accesible, etc.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param propertyName - el nombre LOCAL del parámetro a comprobar.

@return Verdadero si el parámetro fue encontrado y puede ser leído. Falso si no.

■ **canSet(self, propertyName)**

Comprueba si el parámetro cuyo nombre recibe puede ser modificado o no en función de si es conocido, sólo-lectura, etc.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param propertyName - el nombre LOCAL del parámetro a comprobar

@return Verdadero si el parámetro fue encontrado y puede ser modificado. Falso si no.

■ **cleanRenamings(self, driverManager)**

Lanza un nodo “mux” de ROS que reubica los topics a nuevas direcciones según los parámetros de clase “renaming” en el fichero de traducciones. Precondición: el *driver_manager* tiene que estar previamente inicializado y los topics originales ya publicados.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param driverManager - referencia a una instancia de *driver_manager* que se usará en las reubicaciones

■ **dynamicServers(self, checkDynamic=True)**

Genera una lista con las direcciones relativas o absolutas de los diferentes servidores de reconfiguración dinámica según los datos del fichero de traducciones YAML cargado.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param checkDynamic - decide si comprobar que realmente hay parámetros dinámicos en cada servidor de la lista.

@return La lista de direcciones de los servidores disponibles.

■ **generateReverseDictionary(self)**

Genera un diccionario para el traductor usando como clave el nombre REMOTO (en el driver) y el nombre LOCAL como valor para hacer búsquedas inversas.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@return El diccionario inverso de parámetros indexados por nombre REMOTO (dos entradas por cada uno para incluirlos con y sin la dirección completa).

■ `getServerPath(self, propName)`

Dado un parámetro, obtiene la dirección ROS del servidor que lo contiene para poder modificar la configuración del driver correspondiente.

`@param propName` - el nombre local del parámetro a buscar.

■ `get_property_list(self)`

Método que devuelve la lista completa de parámetros.

`@param self` - la instancia cuyo método es llamado (común en los métodos Python)

`@return` lista completa de los parámetros conocidos por el traductor.

■ `get_topic_path(self, topicName)`

Método para obtener la dirección de un topic determinado. Si se recibe un topic existente se comprueba su existencia y se devuelve; si no se trata de traducir como un parámetro.

`@param self` - la instancia cuyo método es llamado (común en los métodos Python)

`@param topicName` - el nombre el parámetro o topic a buscar.

`@return` la dirección completa del topic asociado al nombre recibido.

■ `interpret(self, propertyName)`

Devuelve los datos asociados a un nombre LOCAL de los conocidos gracias al fichero de configuración (o traducción) según: [nombre REMOTO, tipo de dato, canal].

`@param self` - la instancia cuyo método es llamado (común en los métodos Python)

`@param propertyName` - el nombre LOCAL del parámetro a utilizar.

`@return` una tupla “[nombre REMOTO, tipo de dato, canal]” si encuentra el parámetro. Si no: “None”.

■ `prop_exists(self, propertyName)`

Comprueba la existencia en el diccionario de un parámetro dado según su nombre LOCAL.

`@param self` - la instancia cuyo método es llamado (común en los métodos Python)

`@param propertyName` - el nombre LOCAL del parámetro a comprobar.

`@return` Verdadero si se encontró el parámetro en el diccionario. Falso si no.

■ `readYAMLConfig(self, file_name)`

Carga todo el fichero “file_name” (que contiene los diccionarios de parámetros) en el traductor. Retorna los diccionarios para que estos sean recordados durante la ejecución.

a modo de in which to read the properties. Returned Value MUST be a tuple of two lists with strings and dictionaries respectively.

@param self - la instancia cuyo método es llamado (común en los métodos Python)
@param file_name - el nombre y dirección de un fichero del que leer la configuración (diccionarios).
@return Una tupla de dos listas. La primera contiene las direcciones a cada driver y la segunda el diccionario asociado a cada elemento de la primera.

■ **reverseInterpret(self, reverseProperty)**

Localiza el parámetro “reverseProperty” (según su nombre REMOTO) en el diccionario inverso para obtener el nombre LOCAL.

@param self - la instancia cuyo método es llamado (común en los métodos Python)
@param reverseProperty - el nombre REMOTO del parámetro.
@return El nombre LOCAL del parámetro si fue encontrado. Si no: “None”.

■ **updatePptyRef(self, propertyName, newPath)**

Cambia el nombre REMOTO o dirección de un parámetro en el traductor. (Sólo durante la ejecución, no se almacenan los cambios).

@param self - la instancia cuyo método es llamado (común en los métodos Python)
@param propertyName - el nombre LOCAL del parámetro.
@param newPath - el nombre REMOTO para asignarle al parámetro.
@return la nueva dirección o nombre asignado al parámetro si éste fue encontrado. Si no: “None”.

■ **get_basic_name(propName)**

Método estático para obtener el nombre de base en una dirección, es decir la última sección: /root-Dir/secondary/propName/ = /rootDir/secondary/propName = propName

@param propName - el nombre completo de un parámetro, posiblemente incluyendo una dirección relativa.
@return la última parte del nombre (la base) sin la dirección o una cadena vacía si no fue posible la acción.

6.6. Estructura de clases de la odometría

El segundo paquete de la solución, encargado de estimar la odometría visual tras cada movimiento, comparte el objetivo de favorecer la mantenibilidad, y está dividido en dos clases que son suficientes para dividirse entre la funcionalidad principal (clase *odometryComm*) y los métodos de tratamiento de TFs (clase *myTransf*). La primera clase se encarga de reunir toda la configuración necesaria para preparar el algoritmo ICP y ponerlo en marcha cada vez que obtiene una nueva nube de puntos; y la segunda hereda de la clase “*tf::Transform*” de ROS para agregarle nuevos métodos sin perjuicio de los ya existentes.

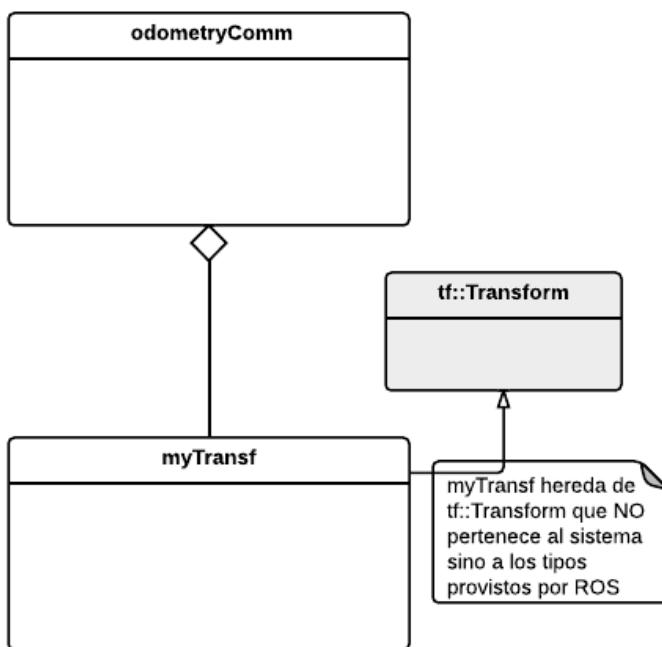


Figura 6.2: Diagrama de clases que incluye las del subsistema de odometría

A continuación se detallan las distintas clases de la estructura representada en el diagrama de clases de la figura 6.2. En cada una se describen los datos miembro y los métodos definiendo cada uno de ellos y siguiendo aún el convenio de “Python Docstring” para documentar especificando los parámetros de entrada mediante “@param” y los valores de retorno con “@return”.

El modo adecuado de lanzar el nodo de odometría, cargar las configuraciones y poner en marcha la clase principal se exponen como parte del documento de *Diseño* bajo el apartado *Código fuente del producto* (4.6).

6.6.1. Clases del paquete odometría

Clase *odometryComm*

Datos Miembro

Variables para el funcionamiento del algoritmo ICP

- **PointCloudT::Ptr previous** conserva la última nube de puntos recibida con la que se realizará la comparación en el algoritmo de ICP para estimar el movimiento realizado.
- **myTransf globalTF, lastRelativeTF, fixed_camera_transform** son las tres TFs básicas durante los cálculos: la posición y orientación actuales, el movimiento relativo a la posición global anterior y la posición fija de la cámara respecto al robot.
- **int odomStatus** guarda en cada instante el último estado de actividad del algoritmo para conocer si está en ocupado, a la espera o produjo algún error. Los estados pueden ser modificados y ampliados, pero actualmente se identifican con una estructura *enum* con las siguientes opciones: ERROR_STATUS=-1, UNLOADED=0, INITIALIZED=1, RUNNING=2, WAITING=3 y UNRELIABLE_RESULT=4.

Adicionalmente, cabría incluir en esta lista todas las variables del apartado “Parámetros del algoritmo ICP” enumeradas en la sección 6.7 sobre los parámetros de configuración.

Variables relacionadas con el comportamiento del nodo

- **bool mIsInitialised; boost::recursive_mutex mIsInitialisedMutex; ros::AsyncSpinner *mSpinner;** son variables auxiliares para ayudar a los métodos “init” y “shutdown” a controlar el estado y sincronización del nodo.
- **boost::mutex callback_mutex** evita que dos métodos manejadores de eventos entren en conflicto por ejecutarse al mismo tiempo.
- **ros::ServiceServer *server_updateOdometry, *server_resetGlobals, *server_getLastStatus** son punteros a los servicios publicados por el nodo, que deben existir durante toda la ejecución para que otros nodos puedan acceder a las funcionalidades correspondientes.
- **ros::Publisher pcl_pub_aligned, pcl_pub_final, pcl_pub_initial, odom_answer_publisher** guardan instancias de los topics publicados por el nodo y que deben mantenerse mientras estén en funcionamiento.
- **ros::Subscriber pcl_sub, camera_tf_sub** contienen los suscriptores a topics externos para las nubes de puntos y en caso de que llegue una nueva TF con la posición de la cámara.

Funciones Miembro

Control de ejecución del nodo

■ **bool init(int argc, char** argv, std::string nodeName, uint32_t rosInitOptions=0)**

Método encargado de inicializar el objeto tras su creación. Se distingue del constructor por diversos motivos prácticos como la capacidad para reinicializar un objeto desde la sección principal del código. De ocurrir así el método detiene toda acción del objeto y lo reinicializa desde cero.

@param argc - número de argumentos de línea de comando esperados.

@param argv - argumentos provenientes de la línea de comando.

@param nodeName - nombre con el que se publicará el nodo ROS en el entorno.

@param initOptions - opciones de inicialización.

@return Verdadero en caso de éxito y Falso en caso contrario.

■ **bool shutdown()**

Al llamar a este método, el objeto es “apagado”, es decir, se detienen las acciones que llevaba a cabo y se trata de limpiar en la medida de lo posible el rastro de recursos utilizados. Este método siempre es llamado durante la “destrucción” del objeto, pero no tiene efecto si se llama con el objeto ya parado (pero no destruido).

@return Verdadero en caso de éxito y Falso en caso contrario.

Habilitar filtrados de nubes

■ **void setDownsampleFiltering(float newLeafSize)**

Realiza los cambios necesarios para activar el filtrado de resolución (submuestreo) de las nubes de puntos. En la implementación actual se encarga de activar la marca (o *flag*) correspondiente y asignar el tamaño de “hoja” (*leaf*).

@param newLeafSize - distancia, en metros, entre los puntos de la nube para que sean considerados de interés (lo bastante alejados).

NOTA: La documentación no especifica el modo de medir la distancia, por lo que se asume que se trata de la distancia tridimensional al tener en cuenta todas las coordenadas (X, Y, Z).

■ **void setDistanceFiltering(float minimum, float maximum)**

Realiza los cambios necesarios para activar el filtrado por distancia en profundidad (sólo en función de X respecto a la cámara). En la implementación actual se encarga de activar la marca (o *flag*) correspondiente y asignar los valores mínimo y máximo.

@param minimum - valores mínimo de distancia para los puntos a conservar en la nube.

@param maximum - valores máximo de distancia para los puntos a conservar en la nube.

■ void setNoiseFiltering(float radDist, int noOfNeighbours)

Realiza los cambios necesarios para activar el filtrado de ruido (según la cantidad de vecinos y la distancia a la que estos se consideren tales). En la implementación actual se encarga de activar la marca (o *flag*) correspondiente y asignar los valores de radio (distancia de los considerados vecinos) y número de vecinos.

@param radDist - máxima distancia, en metros, entre cada punto y aquellos que se considerarán “vecinos”.

@param noOfNeighbours - mínimo número de vecinos para que un punto se considere “acompañado” y conservarlo.

Métodos privados auxiliares

■ void printConfiguration()

Muestra, mediante texto en la consola de ejecución, la lista de parámetros disponibles y sus valores de configuración.

■ PointCloudT::ConstPtr fetchPointCloud (std::string topicName, int timeout = 20)

Obtiene la última nube de puntos publicada en el topic especificado (la más reciente). No comprueba la marca de tiempo puesto que se deja tal responsabilidad al método que lo invoca.

@param topicName - nombre del topic del que obtener la nube de puntos.

@param timeout - tiempo límite a esperar por la nube de puntos.

@return un puntero constante a la nube de puntos obtenida o nulo si no fue posible obtenerla.

■ void apply_filters(PointCloudT::ConstPtr pointCloud1_in, PointCloudT::Ptr &pointCloud1_out)

Aplica, sobre la nube de puntos especificada, los distintos filtros activos guardando el resultado en la otra nube de puntos pasada como parámetro.

@param pointCloud1_in - nube de puntos entrante para ser filtrada.

@param pointCloud1_out - dirección en la que almacenar la nube de puntos resultante.

■ void trimPreviousCloud(PointCloudT::Ptr &pointCloud1_out, double x = 0.75, double y = 0.9, double z = 0.9)

Descarta los puntos más alejados del centro de la nube según los valores “x,y,z” entendidos como el “tanto por uno” de puntos a conservar en cada coordenada.

@param pointCloud1_out - nube de puntos entrante para ser recortada

@param x - proporción (sobre uno) a mantener respecto al valor de 'x'

@param y - proporción (sobre uno) a mantener respecto al valor de 'y'

@param z - proporción (sobre uno) a mantener respecto al valor de 'z'

■ **double round(double value, int noDecimals)**

Redondea un número real hasta el número de decimales especificado.

@param value - número a redondear.

@param noDecimals - cantidad de decimales a conservar.

■ **void printMatrix4f(Eigen::Matrix4f &tMatrix)**

Imprime la TF especificada almacenada en forma matriz de tipo “Matrix4f”.

@param tMatrix - matriz “Matrix4f” con los valores de la TF.

■ **bool generate_tf(Eigen::Matrix4f &mat, double roll, double pitch, double yaw)**

Genera una TF orientada según las rotaciones “roll,pitch,yaw” especificadas y centrada en la posición espacial (0, 0, 0).

@param mat - variable de salida en la que almacenar la TF generada.

@param roll - rotación a aplicar sobre el eje X.

@param pitch - rotación a aplicar sobre el eje Y.

@param yaw - rotación a aplicar sobre el eje Z.

■ **void setOdomStatus(int status)**

Actualiza el código de estado del algoritmo en función del parámetro especificado. Actualmente se trata como un valor entero si bien podrían utilizarse otros tipos o estructuras de datos.

@param status - código que se almacenará en la variable “odomStatus”.

Tratamiento de matrices

■ **double matToDist(Eigen::Matrix4f t)**

Calcula la distancia, en metros, recorrida según el movimiento descrito como una TF

@param t - TF sobre la que realizar el cálculo.

■ **void matToXYZ(Eigen::Matrix4f t, double& x, double& y, double& z)**

Extrae los valores individuales de “X,Y,Z” de una TF representada como Matrix4f.

@param t - TF de la cual obtener los valores.

@param x - variable de salida para almacenar el valor de X.
@param y - variable de salida para almacenar el valor de Y.
@param z - variable de salida para almacenar el valor de Z.

■ void matToRPY(Eigen::Matrix4f t, double& roll, double& pitch, double& yaw)

Extrae los valores individuales de “R,P,Y” (rotaciones) de una TF representada como Matrix4f.

@param t - TF de la cual obtener los valores.

@param roll - variables de salida para almacenar el valor de giro roll (alabeo).

@param pitch - variables de salida para almacenar el valor de giro pitch (cabeceo).

@param yaw - variables de salida para almacenar el valor de giro yaw (guiñada).

■ void filter_resulting_TF(myTransf *targetTF)

Realiza el filtrado de la TF especificada mediante diferentes pasos y criterios (según la implementación). Actualmente se anulan los valores cercanos a cero (considerados ruido), se cambia el sistema de coordenadas desde la cámara al robot (cuyo movimiento es el que realmente se busca) y se finaliza con un último redondeo orientado de valores cercanos a cero para eliminar valores residuales tras los cálculos.

@param targetTF - TF sobre la cual realizar el filtrado (modificando la variable original).

Métodos conversores entre estructuras

■ myTransf eigenToTransform(Eigen::Matrix4f tMatrix)

Convierte la matriz de tipo “Matrix4f” en la TF de tipo “myTransf” correspondiente.

@param tMatrix - matriz original de la TF.

@return la TF resultante, almacenada como “myTransf” y representando el mismo movimiento o posición y orientación.

■ bool fill_in_answer(my_odometry::odom_answer &res)

Método para cargar, según la información actual, la estructura de datos correspondiente al mensaje que el nodo publicará como actualización. Incluye el código de estado del algoritmo y las posiciones y orientaciones global y relativa.

@param res - estructura en la cual almacenar la información a enviar.

@return Verdadero siempre que se llegue al final del método sin que interrumpa una excepción.

Métodos para lanzar el algoritmo

- **Eigen::Matrix4f process2CloudsICP(PointCloudT::Ptr &cloud_initial,
PointCloudT::Ptr &cloud_final, double *final_score_out=0)**

Configura y lanza el algoritmo ICP aplicado sobre las dos nubes de puntos especificadas para obtener la TF que describe la posición relativa entre ellas.

@param cloud_initial - nube de puntos inicial tomada en el instante considerado “de partida”.

@param cloud_final - segunda nube de puntos final, tomada tras realizar el presunto movimiento.

@return la TF que representa la posición relativa entre las dos escenas, almacenada como matriz “Matrix4f”.

- **Eigen::Matrix4f process2CloudsICP(PointCloudT::Ptr &cloud_initial,
PointCloudT::Ptr &cloud_final, Eigen::Matrix4f &hint, double *final_score_out=0)**

Sobrecarga del método del mismo nombre con el añadido de incluir una “pista” para que el algoritmo comience su búsqueda desde ella.

@param cloud_initial - nube de puntos inicial tomada en el instante considerado “de partida”.

@param cloud_final - segunda nube de puntos final, tomada tras realizar el presunto movimiento.

@param hint - TF que sugiere una solución “probable” de la cual parte el algoritmo para afinar la búsqueda.

@return la TF que representa la posición relativa entre las dos escenas, almacenada como matriz “Matrix4f”.

Manejadores de servicios y lanzador

- **bool update_odometry(my_odometry::odom_update_srv::Request &req,
my_odometry::odom_update_srv::Response &res)**

Manejador del servicio que ordena una nueva estimación de movimiento al algoritmo de odometría utilizando la última nube almacenada y la nueva que sea capturada.

@param req - estructura vacía para mantener el formato de los manejadores de servicios.

@param res - variable de respuesta en la que almacenar los datos. Almacena las TFs globa y relativa, tanto en estructuras “myTransf” como “Odometry”, así como el código de estado del algoritmo.

@return Verdadero si no se produjo ningún error.

- **bool get_last_status(my_odometry::statusMsg::Request &req,
my_odometry::statusMsg::Response &res)**

Manejador del servicio que permite solicitar el código de estado del algoritmo.

@param req - estructura vacía para mantener el formato de los manejadores de servicios.

@param res - variable de respuesta en la que almacenar el valor solicitado. Almacena el código de estado del algoritmo.

@return Verdadero si no se produjo ningún error.

- **bool reset_globals(my_odometry::emptyRequest::Request &req,
my_odometry::emptyRequest::Response &res)**

Manejador del servicio que permite reiniciar los valores de odometría para recobrar la posición y orientación iniciales (matriz identidad: posición en el origen de coordenadas y sin rotaciones). Se ocupa de borrar también la última nube de puntos conocida, pues es anterior al reinicio desde la nueva posición inicial mencionada.

@param req - estructura vacía para mantener el formato de los manejadores de servicios.

@param res - variable de respuesta en la que almacenar la misma. Almacena un valor booleano correspondiente al valor de retorno del presente método.

@return Verdadero si no se produjo ningún error.

Métodos de callback

- **void cameraTF_callback(const tf::tfMessageConstPtr& newTF)**

Método desencadenado al recibir una TF (vía topic) para actualizar la posición de la cámara, es decir, la TF “fixed_camera_transform”. El mensaje debe contener la TF que describe la posición y orientación relativas de la cámara respecto al centro de coordenadas del robot. Es necesaria para poder calcular el movimiento con respecto al robot en vez de hacerlo respecto a la cámara.

@param newTF- TF que describe la nueva situación relativa de la cámara.

- **void PCL_callback(const PointCloudT::ConstPtr & cloud_msg)**

Método desencadenado al recibir una nueva nube de puntos (vía topic). Si el modo “manual” está desactivado (en favor del automático), se lanza un nuevo cálculo de odometría mediante la invocación del método “common_callback_routine”.

@param cloud_msg - nueva nube de puntos recibida.

- **void STR_callback(const std_msgs::String::ConstPtr & nextTopic)**

Método desencadenado al recibir una cadena de texto correspondiente a un topic donde leer una nueva nube de puntos. Como respuesta obtiene trata de obtener una nube de puntos del topic especificado e invoca al método “common_callback_routine” para estimar el último movimiento y la nueva posición.

@param nextTopic - nuevo nombre de topic recibido.

- **void common_callback_routine(PointCloudT::ConstPtr & pointCloud1_aux, std::string nextTopic)**

Este método es invocado por otros callbacks para realizar acciones comunes para el cálculo de nuevas estimaciones de odometría a partir de las dos últimas nubes de puntos obtenidas.

@param pointCloud1_aux - nueva nube de puntos para filtrar y sobre la cual aplicar el algoritmo.

@param nextTopic - nombre del topic del cual obtener nuevas nubes de puntos si la recibida es demasiado antigua.

6.6.2. Clase *myTransf*

Datos Miembro

Los datos miembro de esta clase están implementados en la clase padre “tf::transform” y no pertenecen, por tanto, al alcance del proyecto actual. Aún así es necesario decir que se trata de una TF formada por dos miembros, a saber: una matriz de dimensión 3 para definir la orientación (**Matrix3x3 tf::Transform::m_basis**) y un vector de 3 posiciones para determinar la posición (u origen) de la TF (**Vector3 tf::Transform::m_origin**).

Funciones Miembro

No se incluyen aquí los métodos pertenecientes a la clase padre “tf::transform” cuya implementación no fue parte de este desarrollo y pueden ser consultados en su propia documentación de referencia⁴.

Métodos conversores de TFs

- **tf::Transform getTransform()**

Método para convertir un objeto “myTransf” (sobre el que se invoca el método) en su versión equivalente “tf::Transform”

@return la TF original almacenada como “tf::Transform”

- **nav_msgs::Odometry transformToOdometry(ros::Time stamp = ros::Time(0), std::string frameID = “my_odom_tf”, std::string childID = “base_link”)**

Convierte la TF sobre la que se invoca en una estructura de tipo “Odometry” describiendo la misma TF. Se trata de una estructura más orientado a la publicación de movimientos y estimaciones odométricas generalizada en el entorno ROS.

@param stamp - marca de tiempo para incluir en la estructura. Se puede usar “0” como tiempo desconocido o irrelevante.

⁴La referencia de tf::transform puede encontrarse en “ http://mediabox.grasp.upenn.edu/roswiki/doc/unstable/api/tf/html/c%2B%2B/classtf_1_1Transform.html ”

@param frameID - nombre para identificar la TF de referencia en el entorno ROS global.
@param childID - nombre para identificar la propia TF en el entorno ROS global.
@return la estructura resultante que expresa la TF inicial y los datos adicionales.

- **tfMessage transformToTFMsg(ros::Time stamp = ros::Time(0), std::string frameID = “my_odom_tf”, std::string childID = “base_link”)**
Convierte la TF sobre la que se invoca en una estructura de tipo “tfMessage” describiendo la misma TF. Se trata de una estructura más orientado a la publicación de TFs como tales en el entorno ROS.
@param stamp - marca de tiempo para incluir en la estructura. Se puede usar “0” como tiempo desconocido o irrelevante.
@param frameID - nombre para identificar la TF de referencia en el entorno ROS global.
@param childID - nombre para identificar la propia TF en el entorno ROS global.
@return la estructura resultante que expresa la TF inicial y los datos adicionales.

Métodos para extraer, publicar o mostrar por pantalla las TFs

- **void publishOdom(ros::Publisher &odom_publisher)**
Publica un mensaje ROS de tipo “Odometry” al entorno global con la información de la TF sobre la que se invoca y usando el canal “odom_publisher” especificado.
@param odom_publisher - objeto en el cual depositar el mensaje para su envío a través de su topic asociado.
- **void printTransform()**
Muestra por pantalla, en la consola de ejecución, la información de la TF “myTransf” sobre la que se invoca
- **void broadcastTransform(std::string tfChannel, std::string tfParent=“map”)**
Publica la TF sobre la que se invoca al entorno ROS a través de un objeto “TransformBroadcaster” para que sea visible por el resto de los nodos.
@param tfChannel - nombre para identificar la propia TF en el entorno ROS global.
@param tfParent - nombre para identificar la TF de referencia en el entorno ROS global.
- **double transformToDistance()**
Calcula la distancia, en metros, recorrida según el movimiento descrito por la TF sobre la que se invoca
@param t - TF sobre la que realizar el cálculo.

■ `double transformToRotation(double accuracy)`

Calcula la cantidad de rotación correspondiente al movimiento descrito por la TF sobre la que se invoca. Se calcula este en base a la componente 'w' de la rotación (descrita como cuaternión) y su diferencia con '1'.

@param t - TF sobre la que realizar el cálculo.

Métodos para modificar TFs

■ `void changeCoordinates()`

Corrige los valores originales de la TF para adaptarlos al convenio ROS que dice:

“Los sistemas de coordenadas 3D en ROS siempre siguen la regla de la mano derecha, con X hacia delante, Y hacia la izquierda y Z hacia arriba.”⁵.

Dado que las nubes de puntos siempre utilizan Z para la profundidad e Y para la altura, el cambio es de (X, Y, Z) a (Z, X, Y)

■ `void round_near_zero_values(double accuracy, double rotationAccuracy)`

Realiza el redondeo de los valores cercanos a '0' (aquellos cuya diferencia en valor absoluto sea menor que “accuracy” o, en el caso de las rotaciones, “rotationAccuracy”) que son considerados “ruido”.

@param accuracy - la diferencia mínima entre '0' y cada valor para que éste no sea considerado ruido

@param rotationAccuracy - la diferencia mínima entre '0' y cada rotación para que ésta no sea considerada ruidosa

■ `void get_robot_relative_tf(myTransf &fixedTF)`

Realiza una rotación descrita por “fixedTF” sobre la TF desde la que se invoca. Se interpreta que “fixedTF” describe la relación en el espacio de dos sistemas de coordenadas fijos entre sí de tal manera que el movimiento de uno permite calcular el movimiento del otro desde su propio sistema de referencia.

@param fixedTF - la TF fija que describe la relación entre los dos sistemas de coordenadas (orientada del original al que se pretende obtener).

■ `void rotate_tf(double roll, double pitch, double yaw)`

Realiza una rotación de la TF sobre la que es invocado y de forma acorde a los giros R,P,Y especificados

@param roll - rotación a aplicar respecto al eje X

@param pitch - rotación a aplicar respecto al eje Y

@param yaw - rotación a aplicar respecto al eje Z

⁵Tomado de “<http://ros.org/wiki/tf/Overview/Transformations>”, apartado 2.

Métodos para crear TFs

■ **bool generate_tf(double roll, double pitch, double yaw)**

Genera una TF en 6 dimensiones que describe la posición en el origen (0, 0, 0) y una orientación dada por los valores de giro “R,P,Y” especificados.

@param roll - rotación a aplicar respecto al eje X

@param pitch - rotación a aplicar respecto al eje Y

@param yaw - rotación a aplicar respecto al eje Z

6.7. Configuraciones y parámetros

Con la intención de dar el máximo control al usuario final, la mayoría de los parámetros del código han sido externalizados como parámetros de la aplicación. A continuación se muestra una lista de las variables que se cargan durante la inicialización de cada uno de los dos nodos.

6.7.1. Parámetros del nodo interfaz configurables durante el lanzamiento

Se muestra, debajo, la lista de parámetros configurables en `my_adaptor` que se encuentran implementados en forma de variables globales declaradas en el fichero “ctrl_interface.py” y se incluyen sus parámetros por defecto.

Nombre del fichero de traducción y valores de espera máximos para las conexiones

- `propertyConfigFile = “propertyConfigFile.yaml”`
- `topicTimeOut = 3`
- `serviceTimeOut = 3`

Direcciones de los servicios para leer del controlador mediante la interfaz

- `getStrSrv = “get/StringProperty”`
- `getIntSrv = “get/IntProperty”`
- `getFloatSrv = “get/FloatProperty”`
- `getBoolSrv = “get/BoolProperty”`
- `getDispImgSrv = “get/TopicLocation”`

Direcciones de los servicios para modificar el controlador mediante la interfaz

- `setStrSrv = “set/StrProperty”`
- `setIntSrv = “set/IntProperty”`
- `setFloatSrv = “set/FloatProperty”`
- `setBoolSrv = “set/BoolProperty”`

- **setLocationSrv = “set/TopicLocation”**

Los timeouts indican el tiempo exacto, en milisegundos, que esperarán servicios o topics por un mensaje o conexión nuevos desde que comienza la escucha hasta que asume que ha fallado y se cancela la operación. Si el tiempo especificado es demasiado escaso, el nodo renunciará a mensajes que, considera, llegan tarde aunque en realidad no haya ningún problema. Si es demasiado alto (o infinito, si se establecer a cero), el nodo se bloqueará durante ese tiempo cada vez que un mensaje o señal falle sin llegar.

Las direcciones de los servicios establecen las llamadas que deberán hacer otros nodos que deseen comunicarse con la interfaz con el objetivo de leer o modificar parámetros del controlador.

Todo ello se encuentra al principio del fichero “ctrl_interface.py”, comenzando por una lista de nombres clave del tipo “KEY_NOMBRE_PARAMETRO” que almacenan el nombre **externo** del parámetro (dentro del servidor de parámetros). Inmediatamente debajo de la lista de claves se encuentran los valores por omisión bajo el nombre “DEFAULT_NOMBRE_PARAMETRO”. Las dos listas anteriores sólo pueden ser modificadas por el programador en tiempo de desarrollo, pero marcan el procedimiento para implementar configuraciones adicionales para la inicialización del nodo.

6.7.2. Parámetros para configurar el comportamiento de la odometría

Los parámetros que se listan ahora están incluidos en el fichero *my_odometry.hpp* para integrarlos con la clase *odometryComm*, pero podrían haberse implementado en un fichero adicional exclusivo de parámetros. En cualquier caso completan el listado de datos miembro de dicha clase y se presentan a continuación con su valor por defecto y su significado.

Parámetros del algoritmo ICP

- **int maxIterations** es el máximo número de iteraciones que realizará el algoritmo ICP en búsqueda de un estimación.
- **double maxDistance** define la distancia máxima esperable entre los puntos correspondientes de las dos nubes analizadas.
- **double epsilon** es un criterio de parada basado en la diferencia máxima entre el resultado de la última iteración y el de la anterior.
- **double euclideanDistance** es un criterio de parada que depende de la suma de distancias euclidianas entre puntos correspondientes elevadas al cuadrado.
- **int maxRansacIterations** determina el máximo número de iteraciones del algoritmo RANSAC que se ejecuta en cada iteración de ICP.
- **double ransacInlierThreshold** define la distancia máxima entre puntos para que sean considerados “útiles” por el algoritmo RANSAC.
- **double ICPMinScore** dicta la puntuación mínima de una estimación de ICP para que ésta pueda considerarse válida..

- **bool doDownsampleFiltering** activa el filtrado por submuestreo
- **double leafSize** determina la precisión de la nube de puntos. Su valor es inversamente proporcional a la cantidad de puntos.
- **bool doDepthFiltering** activa el filtrado por distancia.
- **double minDepth, maxDepth** son la distancia mínima y máxima de los puntos respecto a la cámara para el filtrado por distancia.
- **bool doNoiseFiltering** activa el filtrado de ruido en las nubes de puntos.
- **int neighbours** es el mínimo número de vecinos para que un punto se considere “acompañado” y conservarlo.
- **double radius** define la máxima distancia, en metros, entre cada punto y aquellos que se considerarán “vecinos”

Nombres de publicación de topics

- **std::string inputStrRequest_topic** - recibe un texto con topic del que deberá leer la siguiente nube de puntos
- **std::string inputPCL_topic** - lee las nubes de puntos publicadas en su nombre para realizar estimaciones
- **std::string cameraTF_topic** - permite recibir una TF nueva para la cámara respecto al robot
- **std::string outputGlobalOdometry_topic** - publica la TF global con la posición y orientación estimada del robot
- **std::string outputRelativeOdometry_topic** - publica la TF relativa del último movimiento
- **std::string outputAlignedCloud_topic** - publica la nube alineada
- **std::string outputInitialCloud_topic** - publica la nube de partida
- **std::string outputFilteredCloud_topic** - publica la nube inicial filtrada
- **std::string outputOdometryAnswer_topic** - publica el estado completo del algoritmo

Parámetros para tratamiento de nubes y TFs

- **bool manualMode** activa el modo manual si el valor es verdadero, es decir, que el algoritmo sólo captura y estima un nuevo movimiento si así se lo solicita otro nodo o usuario.
- **bool ignoreTime** permite ignorar las marcas de tiempo de las nubes de puntos cuando es verdadero.
- **double measureAccuracy, rotationAccuracy** definen respectivamente la precisión lineal, en metros, y la rotación esperada de la cámara. Si un valor es menor que la precisión de la cámara para medirlo se asume que debe ser ruido.

- **double cloud_trim_x, cloud_trim_y, cloud_trim_z** son las porciones (sobre '1') de cada escena que deberán seguir en el rango de visión tras un movimiento. Dicho de otro modo, “1-cloud_trim” es la parte que se considera “margen” de la nube anterior y se elimina al compararla con la siguiente.
- **double kinectRoll,kinectPitch, kinectYaw** son los valores de giro “alabeo, cabeceo y guiñada” que determinan la orientación de la cámara respecto al robot.