Trabajo de Fin de Grado

Odometría visual aplicada a la localización de un robot con Kinect en Interiores

Visual odometry applied to the location of a robot with Kinect Indoor

Marcos Luis Díaz García

La Laguna, 4 de marzo de 2016

D. **Jonay Tomas Toledo**, con N.I.F. 78.698.554-Y profesor titular de universidad adscrito al departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"Odometría visual aplicada a la localización de un robot con Kinect en Interiores."

ha sido realizada bajo su dirección por D. **Marcos Luis Díaz García**, con N.I.F. 42.196.179-B.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 4 de marzo de 2016.

Agradecimientos

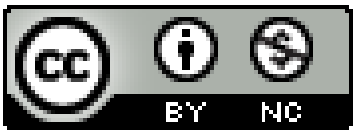
Agradecer en primera instancia a Jonay por ofrecerme la oportunidad de realizar el trabajo bajo las condiciones en las que estaba, fuera de la isla y con trabajo. Sin su paciencia y colaboración este trabajo no habría sido tan gratificante para mí.

Quiero agradecer a toda mi familia, a mi madre María Nieves García Martín, a mi padre Marcos Luis Díaz Martín y mis hermanos. Pero en especial a esa persona que me acompaña cada día y me tiene que aguantar muchas veces mi manera de ser, muchas gracias Lennimar Carlonia Soto García.

También agradecer a todos mis amigos especialmente a los compañeros de Amathink, por estar juntos y ayudarme en muchas ocasiones.

Gracias a todos.

Licencia



Resumen

El objetivo de este trabajo ha sido el desarrollar un sistema que permita medir la precisión de la odometría visual, en espacios interiores a través de sensores de profundidad, en un entorno Linux.

Se ha elegido la Kinect 1.0 desarrollada por Microsoft, para obtener los datos de profundidad del entorno que esta nos ofrece.

El desarrollo del software se hizo en un entorno Linux con el framework ROS, que nos proporciona todas las herramientas necesarias para poder interactuar con la Kinect 1.0, y poder obtener todos los datos necesarios para la obtención de la odometría visual.

Se ha elegido el paquete "freenect_launch" del framework ROS, que nos permite a través del driver "OpenNI" interactuar desde ROS con la Kinect 1.0.

Para la creación de la odometría visual, se utilizara el paquete rtabmap_ros que nos ofrece ROS Indigo, el cual nos permite subscribirnos al topic "odom", para poder utilizar la información a tiempo real que nos ofrece la Kinect 1.0.

El paquete software ROS resultante del cálculo de la odometría visual se programara en el lenguaje de programación Pyhton, el cual integrara todas las funcionalidades del sistema mejorando la precisión de la medida con técnicas estadísticas.

Palabras clave: Odometría, ROS, Kinect, *freenect_launch, rtabmap_ros.*

Abstract

The objective of the work has been to develop a system, which we can measure the accuracy of the visual odometry, indoors spaces and through depth sensors, in a Linux environment.

I have chosen the Kinect 1.0 developed by Microsoft, with the Kinect we could obtain depth data of the environment that this system offers us.

The software development was made in a Linux environment with the ROS framework, which provides us all the tools necessary to be able to interact with the Kinect 1.0, and get all the information about the visual odometry

I have been chosen the package "freenect_launch" of the ROS framework which allows us, through the "OpenNI" driver, to interact from ROS with the Kinect 1.0.

For the creation of the visual odometry, I have used the package rtabmap_ros that ROS indigo offers us, which enable us to subscribe to topic "odom", where we can use the information in real time that Kinect 1.0 gives us.

This resulting software ROS package of the visual odometry will be programmed in a scripting language Pyhton that will include all the features of the system, improving accuracy with statistical techniques.

Keywords: *Odometry, ROS, Kinect, freenect_launch, rtabmap_ros.*

Índice General

Capítulo 1. Introducción **5**

1.1 Justificación y contextualización. 5

1.2 Objetivos y competencias. 6

1.3 Planificación. 7

1.4 Herramientas utilizadas. 8

1.5 Productos obtenidos. 8

1.6 Descripción de la memoria 9

Capítulo 2. Estado del arte. **10**

2.1 Odometría visual con ICP. 10

2.2 Odometría visual con FOVIS. 10

2.3 Odometría visual con RGB-D. 11

2.4 Odometría visual con SLAM. 12

2.5 Comparación algoritmos. 12

Capítulo 3. Elementos de desarrollo. **14**

3.1 Odometría. 14

3.2 Robot Operating System (ROS). 14

3.3 Kinect 1.0. 16

Capítulo 4. **18**

Limitaciones del sistema. **18**

4.1 Precisión del sensor Kinect 1.0. 18

4.2 Alcance del sensor Kinect 1.0. 19

Capítulo 5. Fases del proyecto **21**

5.1 Análisis. 21

5.2 Preparación entorno de trabajo. 22

5.3 Desarrollo. 22
5.4 Documentación. 22

Capítulo 6. Desarrollo del proyecto. **23**
6.1 Creación paquete ROS. 23
6.2 Obtención de datos de profundidad. 24
6.3 Suscripción al topic *"rtabmap"*. 24
6.4 Creación de un Subscriber Node. 25
6.5 Recepción de mensajes del tipo Odometry. 25
6.6 Creación algoritmo para el cálculo de la odometría. 26
6.7 Comprobación del entorno gráfico. 27
6.8 Corrección del error en los cálculos de la odometría. 28
6.9 Calculo de la trayectoria recorrida en un circuito. 28
6.10 Análisis de los datos obtenidos. 30

Capítulo 7. Presupuesto **32**
7.1 Costes de personal 32
7.2 Costes de Hardware 33
7.3 Costes de Software 33
7.4 Coste total 33

Capítulo 8. Conclusiones y trabajos futuros. **34**

Capítulo 9. Summary and Conclusions **35**

Apéndice A. Título del Apéndice 1 **36**
A.1. Algoritmo OdometryFinal.py 36

Bibliografía **42**

Índice de figuras

Figura 2.1. Diferencia de dos Point Cloud.....10
Figura 2.2. Imagen RGBD.....11
Figura 2.3. Funcionamiento algoritmo RGB-D Mapping.....11
Figura 2.4. cálculo error.....12
Figura 2.5. Grafica error en la trayectoria.....13
Figura 3.1. Flujo de mensajes ROSCORE.....15
Figura 3.1. Comunicación entre Nodes.....15
Figura 3.3. Kinect 1.0.....16
Figura 3.4. Hardware Kinect 1.0.....17
Figura 4.1. Modos de distancia por la Kinect.....19
Figura 4.2. Características de los modos de distancia de la Kinect.....20
Figura 6.1. Diferencia de dos puntos en un plano 3d.....26
Figura 6.2. Herramienta RTAB-Map.....27
Figura 6.3. Circuito en una sala interior.....29
Figura 6.4. Carro de transporte.....29
Figura 6.5. Datos obtenidos en el recorrido del circuito.....30
Figura 6.6. Recorrido del circuito en 3D.....31

Índice de tablas

Tabla 1.1. Descomposición del TFG en tareas principales.....7

Tabla 2.1. Error en la trayectoria estimada con respecto a la trayectoria real.....13

Tabla 7.1. Relación costes personal.....32

Tabla 7.2. Relación costes Hardware.....33

Tabla 7.3. Relación de coste total.....33

Capítulo 1. Introducción

1.1 Justificación y contextualización.

Hoy en día, cuando utilizamos la odometría en robótica hablamos de estimación de posiciones, ya que saber la posición cierta de un robot móvil es sencillamente imposible. Esto es así porque los métodos que utilizamos para calcular esa posición no tienen una precisión absoluta. Nuestra única forma de obtener información es a través de los sensores que el robot posee. Sobre esta información que nos proporciona los sensores utilizaremos modelos matemáticos y cálculos más o menos complejos, pero nada puede remediar que estos datos que obtenemos de los sensores tengan errores en las medidas.

La reciente aparición en el mercado del sensor Kinect impulsa líneas de investigación limitadas hasta hoy por la tecnología. Este dispositivo desarrollado por Microsoft está diseñado para controlar videojuegos, pero a la vez presenta características que resultan muy interesantes para aplicaciones de robótica móvil:

- Sensores compactos y ligeros que proporciona imágenes RGB-D
- Frecuencia de trabajo de 30Hz.
- Rango de trabajo de la detección aceptable, desde 1.2 hasta 3.5 metros.
- Bajo coste, entorno a los 100€.

El framework ROS nos proporcionara a través de sus librerías, herramientas (visualizadores, GUIs, etc.) y aplicaciones (R, tf, rviz, etc.), todo lo necesario para desarrollar del software.

El objetivo principal del proyecto que abordamos consiste en la interacción de los sistemas ROS y Kinect, para poder calcular la odometría visual en interiores, y poder reducir a través de técnicas estadísticas el error que nos ofrecen los sensores de profundidad.

1.2 Objetivos y competencias.

Esta sección describe los objetivos generales y específicos de este proyecto.

Los objetivos y competencias generales propias del Trabajo Fin de Grado son los siguientes:

- Analizar el problema desde un punto de vista teórico y práctico.
- Establecer una planificación adecuada a la carga de trabajo para un proyecto de estas características.
- Desarrollar un software, a partir de la información analizada y la propia reflexión sobre el problema.
- Documentar todo el proceso de realización del proyecto en la medida de lo posible, así como elaborar la memoria final del proyecto.

Los objetivos específicos que tratamos en este trabajo son:

- Instalación y configuración del framework ROS Indigo.
- Instalación y configuración de la Kinect en el entorno de trabajo ROS.
- Crear un paquete ROS que permita la obtención de los datos del sensor RGB-D de la Kinect.
- Implementar un algoritmo estadístico que permita reducir los errores de medición de los sensores de la Kinect.
- Documentar todo el ciclo de vida de la aplicación.

1.3 Planificación.

En este apartado se detallan en forma de tablas los aspectos más importantes de la planificación del proyecto.

Las tareas principales del proyecto se detallan en la siguiente tabla.

Objetivo	Resultado	Tarea	Actividad	Días
1. Desarrollo de un sistema que calcule la odometría de una Kinect en interiores a través del framework ROS.	1.1 Sistema del cálculo de una trayectoria en metros recorrida con una Kinect.	1.1.1 Estudio, análisis y planificación del sistema.	1.1.1.1 Identificación de objetivos y metas factibles.	2 días.
			1.1.1.2 Desarrollo de un método para el seguimiento de la ejecución del proyecto.	2 días.
			1.1.1.3 Determine todas las tareas necesarias.	1 día.
		1.1.2 Instalación de los sistemas.	1.1.2.1 Instalación y configuración del framework ROS.	4 días.
			1.1.2.2 Instalación y configuración de los drivers *"libfreenect"*.	3 días.
			1.1.2.3 Inicialización del paquete ROS odometry.	2 días.
		1.1.3 Programación y pruebas.	1.1.3.1 Creación del código en Python para el cálculo de la trayectoria recorrida por la Kinect.	20 días.
			1.1.3.2 Creación de un recorrido físico.	5 días.
			1.1.3.3 Comprobación de los datos obtenidos con los físicos del recorrido.	6 días.
			1.1.3.4 Mejorar la precisión de los datos obtenidos a través de técnicas estadísticas.	10 días.
		1.1.4 Elaboración de la documentación.	1.1.4.1 Generación de toda la documentación necesaria para la memoria del TFG.	6 días.

Tabla 1.1. Descomposición del TFG en tareas principales

1.4 Herramientas utilizadas.

Para la realización del Trabajo Fin de Grado se han utilizado las siguientes herramientas Hardware:

- Kinect V1.0
- Portátil Lenovo: Intel i7 3630, 16GB Ram, 1 TB Disco Duro, Gráfica Nvidia Geforce GTX 660M.

Las herramientas Software son las siguientes:

- Sistema Operativo Ubuntu 14.04 LTS (Trusty).
- IDE Geany
- Framework ROS Indigo Igloo
- Driver Kinect *OpenNI*.
- Ros Package *rtbmap_ros*.
- Latex, para la memoria del proyecto.

1.5 Productos obtenidos.

Durante la realización de este Trabajo Fin de Grado (TFG) se han obtenido como resultado los siguientes productos principales:

La memoria del TFG contiene toda la información relacionada con el TFG.

La aplicación software generada durante el proyecto, así como toda la documentación necesaria sobre su diseño e implementación y su correcto funcionamiento.

1.6 Descripción de la memoria.

- **Capítulo 1:** Se hace una breve introducción de este trabajo fin de grado, dando a conocer sus objetivos, la metodología de desarrollo y las herramientas utilizadas durante el desarrollo del trabajo.

- **Capítulo 2:** Se estudia que tipos de algoritmos sobre la odometría hay desarrollados y cual es funcionamiento, haciendo por último una comparación entre ellos, para ver cuál es más eficiente.

- **Capítulo 3:** Se analizan todos los elementos de desarrollo que van a interactuar en el proyecto, para el cálculo de la odometría visual con la Kinect en interiores.

- **Capítulo 4:** Se estudia las limitaciones que tienen el sistema tanto por la parte del software ROS, como por la parte del hardware Kinect principalmente en el sensor de profundidad.

- **Capítulo 5:** Se estudian todas las fases de desarrollo que se van a contemplar en el desarrollo del proyecto, terminando con la memoria final del TFG.

- **Capítulo 6:** Se describe todo el proceso de desarrollo del proyecto, paso a paso, explicando como punto importante la creación del script final y el análisis de los datos obtenidos, en el cálculo de la distancia total en un circuito medido previamente.

- **Capítulo 7:** Se detalla el coste total del proyecto a través de un presupuesto.

- **Capítulo 8:** Se muestran las conclusiones finales del trabajo así como los posibles próximos pasos para continuar con el mismo.

Capítulo 2.
Estado del arte.

2.1 Odometría visual con ICP.

ICP [1] es un algoritmo empleado para minimizar la diferencia entre una serie de puntos del PC (Point cloud, nube de puntos del sensor de profundidad). ICP se utiliza a menudo para reconstruir superficies 2D o 3D a partir de diferentes análisis, consiguiendo unir las capturas tomadas desde distintos puntos de vistas (especialmente empleado cuando la odometría a través de la rueda no es fiable por el terreno resbaladizo).

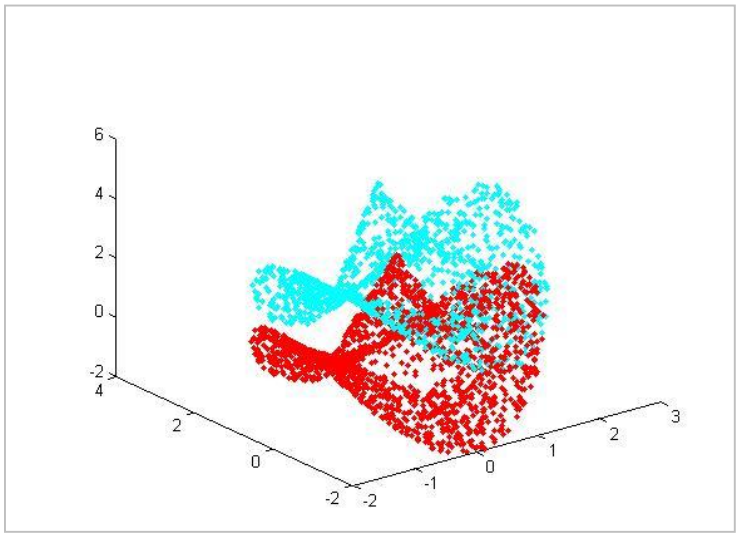


Figura 2.1. Diferencia de dos Point Cloud.

2.2 Odometría visual con FOVIS.

Fovis [2] es un algoritmo que busca puntos característicos en la imagen y trata de encontrar coincidencias entre ambos, para así conseguir el movimiento realizado. A diferencia del algoritmo ICP que utiliza solo la nube de puntos, Fovis añade también información visual de las cámaras, pero solo imágenes en blanco y negro.

2.3 Odometría visual con RGB-D.

El algoritmo RGB-D analiza todos los grados de libertad de la imagen, juntado los valores de color con los de profundidad para dar una solución de odometría en tiempo real. [3]

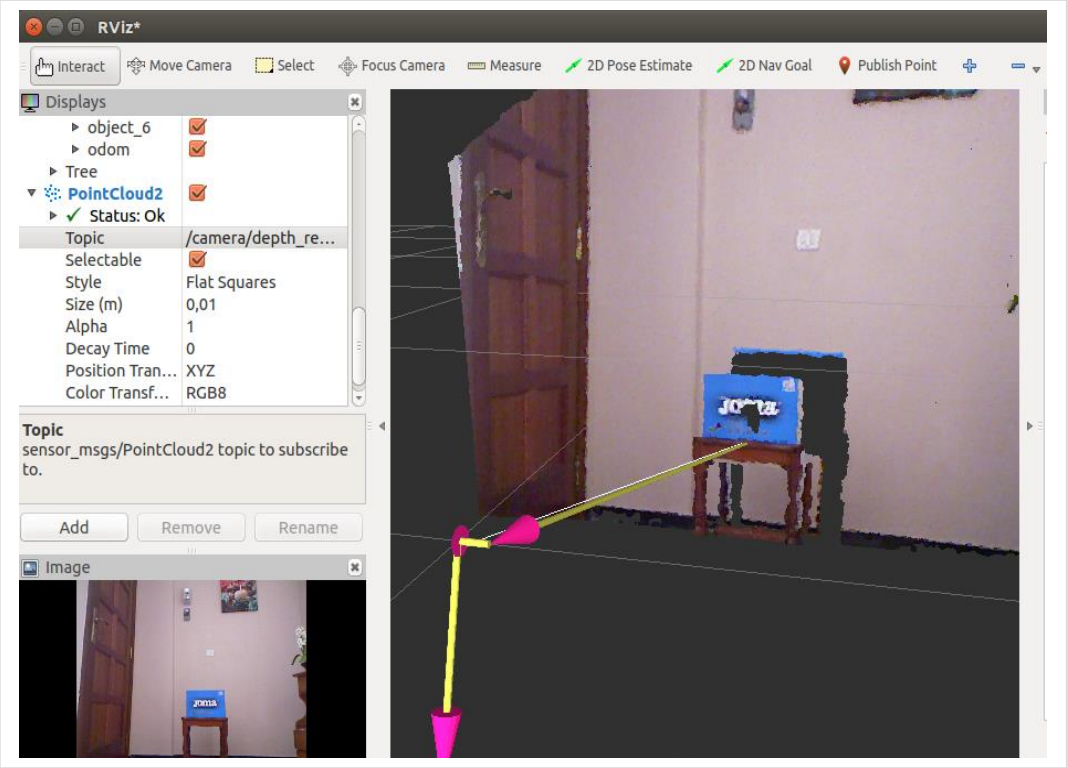


Figura 2.2. Imagen RGBD.

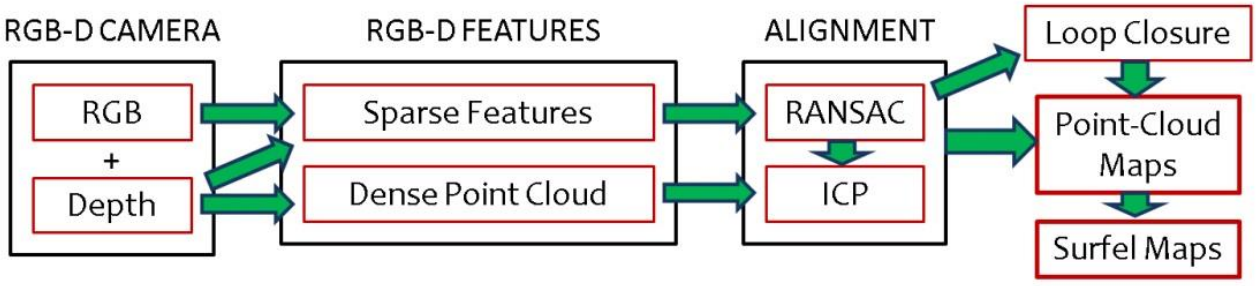


Figura 2.3. Funcionamiento algoritmo RGB-D Mapping.

2.4 Odometría visual con SLAM.

El Slam (Simultaneous localization and mapping) es una técnica que permite construir un mapa de un entorno desconocido en el que se encuentra el robot, a la vez estima su trayectoria al desplazarse dentro de este entorno.

2.5 Comparación algoritmos.

En este apartado se compararán el error en la trayectoria de los distintos algoritmos descritos anteriormente, también se utilizará una combinación previamente analizada entre los distintos algoritmos para tratar de reducir el error en el cálculo del trayecto.

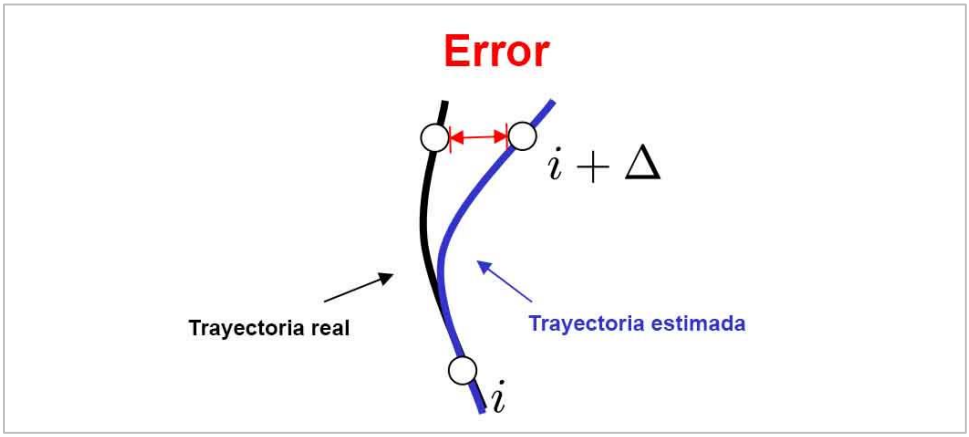


Figura 2.4. cálculo error.

Se ha recogido información de la odometría en unos Dataset [4] para poder comprar todos los algoritmos con el mismo recorrido. Dentro de estos Dataset tenemos toda la información correspondiente del recorrido en una trayectoria con los distintos algoritmos de odometría visual.

Como se puede apreciar en la tabla X hay tres distintos recorridos, room1, room2, desk. Dentro de estos entornos se analizan los distintos algoritmos, también se realiza una media para saber cuál es el algoritmo que menor cantidad de error en la trayectoria produce.

Dataset	Room1	Room2	Desk1	Media
ICP	0.396m	1.298m	**0.058m**	0.584m
RGB-D	**0.138m**	**0.261m**	0.094m	**0.164m**
FOVIS	0.799m	0.273m	0.221m	0.431m
FOVIS/ICP	1.062m	1.225m	0.284m	0.857m
FOVIS/RGB-D	**0.137m**	0.582m	0.094m	0.271m
ICP+RGB-D	0.234m	**0.256m**	0.069m	**0.186m**
FOVIS/ICP+RGB-D	0.231m	0.309m	**0.068m**	0.202m

Tabla 2.1. Error en la trayectoria estimada con respecto a la trayectoria real.

En el análisis de los resultados obtenidos podemos observar, que el algoritmo que mejor se adapta a los entornos estudiados es el RGB-D, dando una medía error aproximada de 16 cm en los cálculos de los tres Dataset. En cuanto a la combinación de algoritmos se puede observar que ICP+RGBD-D es la combinación que menor cantidad de error produce en el cálculo de la odometría.

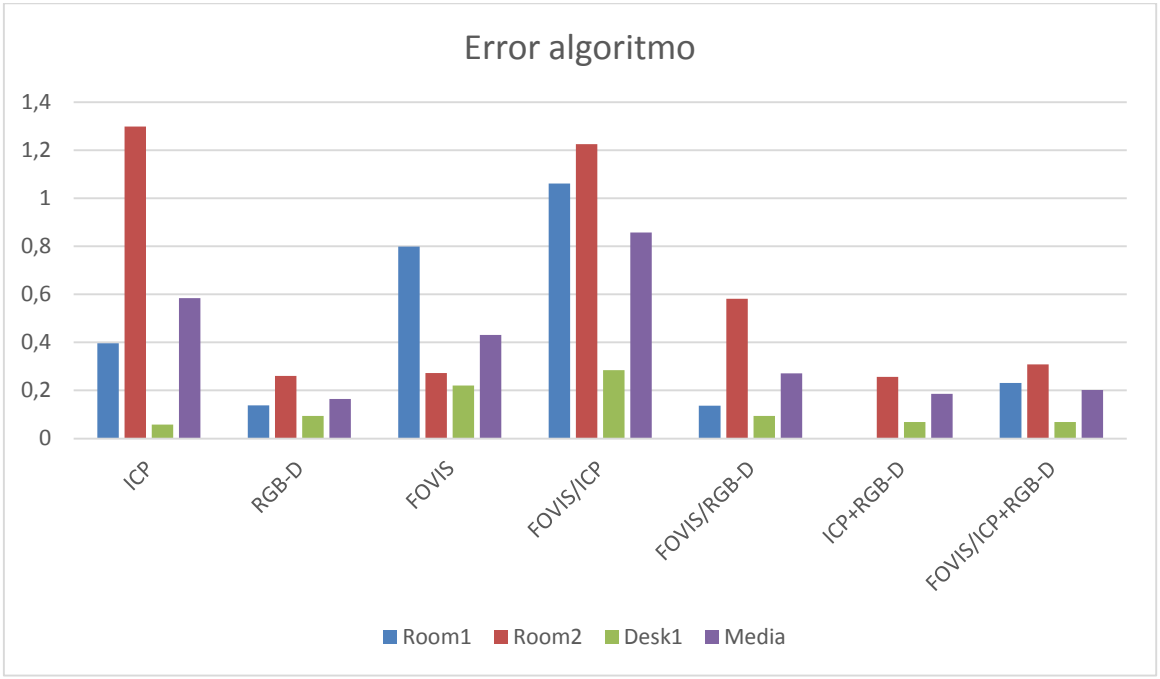


Figura 2.5. Grafica error en la trayectoria.

Capítulo 3.
Elementos de desarrollo.

3.1 Odometría.

Técnica que permite estimar el cambio de posición de un sistema, a partir del uso de datos provenientes de sensores (encoders, magnetómetros, giróscopos, cámaras, etc.). Se utiliza a menudo para calcular la posición relativa de robots móviles. Bajo estos conceptos se hace posible la implementación de los requerimientos vinculados a la localización y el control del robot. Por otro lado, la odometría visual consiste en el proceso de estimación de la posición y la orientación del robot a través del análisis de imágenes tomadas con cámaras asociadas. En otras palabras, se estudia el efecto que produce el movimiento sobre las imágenes obtenidas por las cámaras a bordo del robot. Algunos factores que podrían dificultar el proceso son la falta de iluminación, escenas con escasa textura o con muchos objetos en movimiento. Por ello es frecuente el uso de cámaras de profundidad RGBD para realizar la odometría visual en robots móviles. Dicho concepto permite la implementación de los requerimientos vinculados al mapeo y visualización. [5]

3.2 Robot Operating System (ROS).

ROS (Robot Operating System) provee librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots. ROS provee abstracción de hardware, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes y más. ROS está bajo la licencia open source, BSD. [6]

Tiene dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente y ros-pkg, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o en inglés

stacks) que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

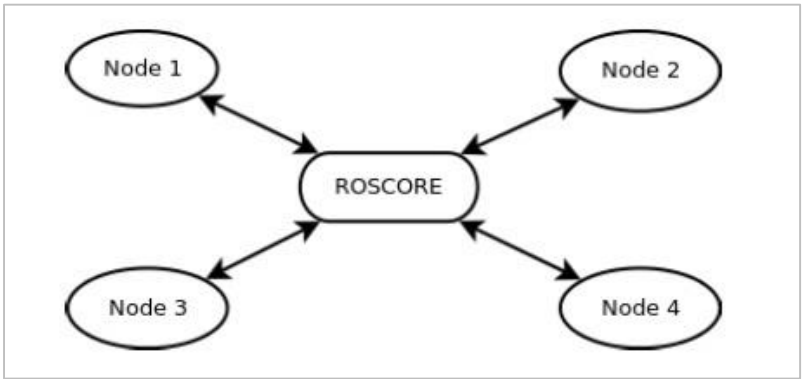


Figura 3.1. Flujo de mensajes ROSCORE.

Organiza el software según una arquitectura de grafos en la que los nodos son porciones ejecutables de código y éstos se comunican entre sí a través de 'topics'. Estos últimos definen la interfaz con la que se comunican dichos nodos. A su vez, los nodos se agrupan en 'packages' (colecciones mínimas de código reusable), y éstos en 'stacks' que permiten compartir el código fácilmente.

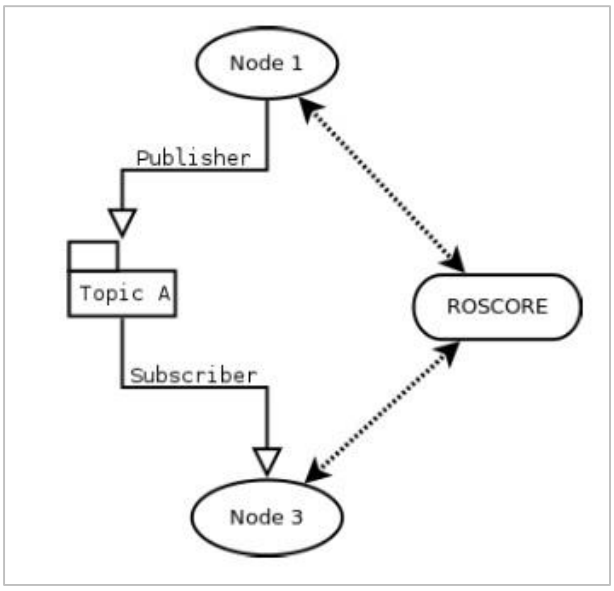


Figura 3.2. Comunicación entre Nodes.

ROS provee las siguientes características: Abstracción de hardware, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes.

3.3 Kinect 1.0.

El dispositivo Kinect (ver Figura 3.3), distribuido en Europa desde Noviembre del 2010, ha revolucionado en gran medida el mundo de la investigación relacionado con la percepción del entorno, tanto por sus prestaciones como por su reducido coste. Lanzado inicialmente como periférico para la videoconsola XBOX 360, se está afianzando en la comunidad robótica dado el amplio abanico de aplicaciones en las que se puede emplear. Desde Junio de 2011 se cuenta con una API oficial de Microsoft [7], si bien, desde poco después de su aparición, la comunidad Open Source hizo público drivers no oficiales como Freenect o CLNUI.

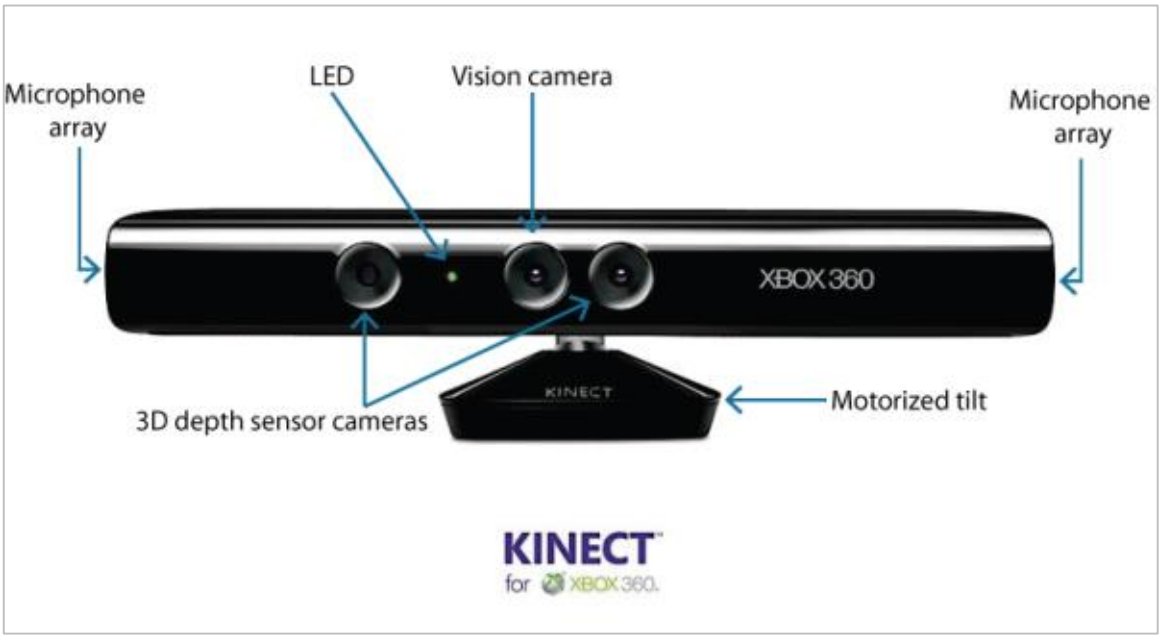


Figura 3.3. Kinect 1.0.

Kinect está provisto de una cámara RGB, un sensor de profundidad, una matriz de micrófonos y una base motorizada que le permite un movimiento de

cabeceo (Ver Figura 3.4). Además, posee una serie de giróscopos que aportan información sobre su orientación. La cámara RGB tiene un flujo de datos de 8 bits por canal, siendo capaz de proporcionar imágenes a una frecuencia de 30Hz para resolución VGA (640x480 píxeles). Por su parte, el sensor de profundidad, al que también se le denomina cámara de rango, está compuesto por un proyector de luz infrarroja combinado con un sensor CMOS monocromo. En este caso el flujo de datos es de 11 bits, manteniendo igualmente una resolución VGA (640x480 píxeles) y una frecuencia de muestreo de 30 Hz. El campo de visión de ésta cámara es de 58º en horizontal y 45º en vertical. El rango de trabajo reportado por el fabricante va desde los 1.2m hasta los 3.5m, aunque en las pruebas realizadas en este trabajo, se ha comprobado que el sensor es capaz de detectar objetos poco reflectantes situados a 0.5 metros.

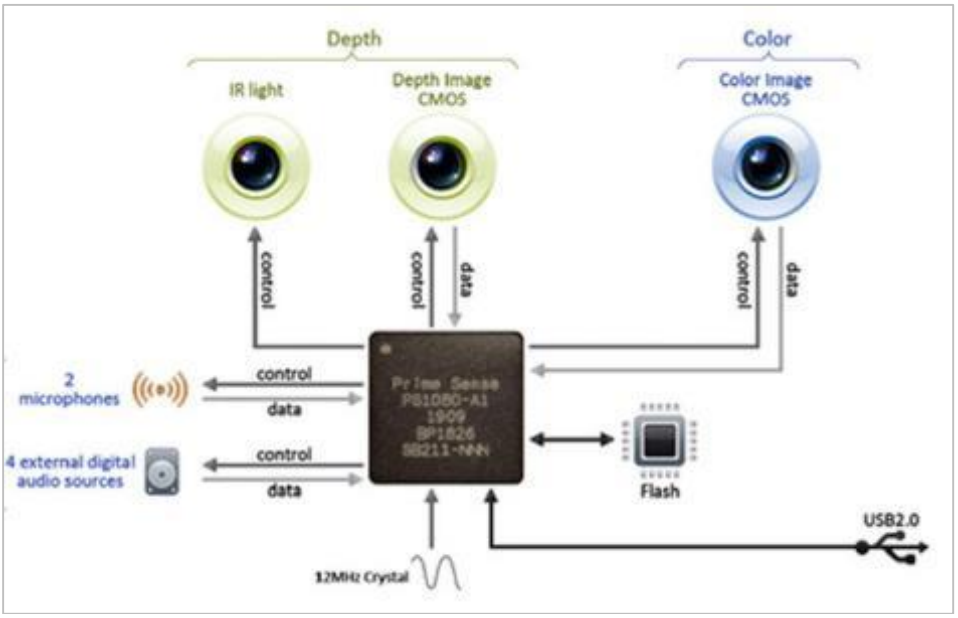


Figura 3.4. Hardware Kinect 1.0.

Capítulo 4.

Limitaciones del sistema.

4.1 Precisión del sensor Kinect 1.0.

Es uno de los puntos más importantes a tener en cuenta en el desarrollo del sistema del caculo de odometría visual.

La Kinect tiene varias limitaciones de precisión en ciertas regiones de la escena, las cuales no se puede estimar o si se estima los valores son datos no aceptables (Lost Odometry, pérdida de odometría). Estas limitaciones de precisión vienen condicionadas tanto por el hardware de la Kinect, como por la propia naturaleza de la escena.

Las características más importantes de las limitaciones de precisión son las siguientes:

- La nube de puntos no cubre de forma continua toda la superficie de los objetos de la escena, por lo tanto hay partes de los pixeles de las imágenes de profundidad que tiene que ser interpolados. Esto implica que el valor de profundidad de un pixel determinado tiene asociado un margen de error. Este error es mayor cuanto más alejado este el objeto de la escena. Esto ocurre así, porque la potencia del haz de luz del infrarrojo se atenúa en el trayecto recorrido de la escena.
- En condiciones ideales de reposo de la Kinect, se detectan oscilaciones en lecturas para un mismo punto, lo cual genera un pequeño margen de error en las distancias.
- Con los problemas descritos anteriormente se derivan también una imprecisión en los bordes de los objetos. Dado que la estimación de profundidad de un objeto considera más de un pixel, unas veces se tomara la profundidad de la parte del objeto más cercano y otras la del más lejano.

- Otra limitación muy importante a tener en cuenta, es la medición en la escena de los objetos cóncavos o reflectantes. Estos objetos pueden causar reflexiones dobles o inter-reflexiones, las cuales pueden hacer que no se puedan estimar los valores en ese punto, generando una pérdida de la odometría.

Las zonas de profundidad que no se pueden estimar, aparecerán como zonas negras en la imagen de profundidad.

4.2 Alcance del sensor Kinect 1.0.

El rango de trabajo reportado por el fabricante va desde los 1.2m hasta los 3.5m, aunque en las pruebas realizadas en este trabajo, se ha comprobado que el sensor es capaz de detectar objetos poco reflectantes situados a 0.5 metros.

Actualmente el sensor de profundidad de la Kinect 1.0, trabaja con dos modos de profundidad, default mode y near mode [8], como se puede apreciar en la Figura 4.1.

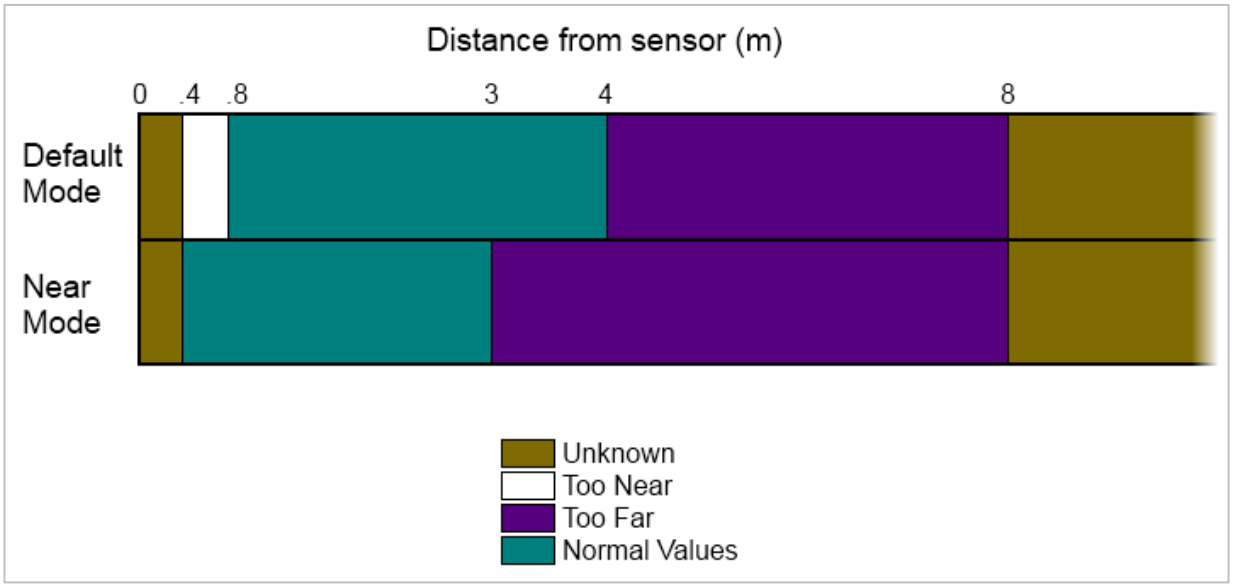


Figura 4.1. Modos de distancia por la Kinect.

La gran diferencia entre los modos que nos ofrece la Kinect 1.0, la podemos observar en la siguiente figura [9].

Mode	Depth & Player	Center Hip Joint	Other 19 Joints
Default	Yes	Yes	Yes
Near	Yes	Yes	No, for v1.0

Figura 4.2. Características de los modos de distancia de la Kinect.

Capítulo 5.
Fases del proyecto

Para llevar a cabo este proyecto, lo dividiremos en cuatro fases, que a su vez contendrán todas las tareas a realizar.

5.1 Análisis.

En esta primera fase se aprenderán los conceptos básicos de ROS para poder desarrollar un programa en esta plataforma, y también otros conceptos específicos de robótica que son necesarios para el proyecto a desarrollar.

El análisis del proyecto comenzará con el aprendizaje de conceptos específicos de robóticas necesarios para el desarrollo del proyecto, entre los cuales están:

- Odometría: la odometría es el uso de datos de sensores de movimiento, para estimar los cambios en la posición del tiempo. La odometría es utilizada por algunos robots, ya sea en las piernas o ruedas, para estimar (no determinar) su posición relativa a un punto de partida. Este método es sensible a los errores debidos a la integración de las mediciones de velocidad en el tiempo, para obtener estimaciones de posición. [10]
- Encoder: el encoder es un sensor basado en la utilización de un haz infrarrojo para detectar posición y velocidad. Consiste en un emisor y un receptor del haz infrarrojo, y en medio, una rueda agujereada que al girar interrumpirá este haz. Si se cuentan las interrupciones del haz, sabiendo cuantos agujeros tiene el disco, se sabrá cuantas vueltas dio el motor, y cuál es la pasión actual del eje. [11]
- Odometría visual: se puede definir la odometría visual como el proceso mediante el cual se determina la posición y la orientación de una cámara o de un sistema de cámaras mediante el análisis de una secuencia de imágenes adquiridas, sin ningún conocimiento previo del entorno. [12]

Después de tener claro los conceptos específicos de robótica necesarios para el proyecto, se empieza una fase de aprendizaje del framework ROS todo su entorno y paquetes necesarios para el desarrollo del software.

En este punto de aprendizaje ROS se nos hace indiscutible, http://wiki.ros.org/, la cual nos aporta todos los tutoriales y conceptos necesarios para el proyecto.

5.2 Preparación entorno de trabajo.

La fase de preparación del entorno de trabajo consta de la instalación y configuración del sistema operativo Ubuntu 14.04 Trusty Tahr [13]. Sobre este sistema operativo se instalará el framework ROS Indigo Igloo [14], en el cual se configurará el paquete *freenect_launch* que utiliza el driver OpenNi, para poder obtener los datos de la Kinect desde ROS.

Una vez obtenidos todos los datos de los sensores de la Kinect, ya se procederá al desarrollo de la aplicación para el cálculo de la odometría visual.

5.3 Desarrollo.

Dentro de esta fase se realizarán todas las tareas de programación y otras tareas necesarias para alcanzar los objetivos del proyecto. Este apartado será ampliamente explicado en el capítulo 6, Desarrollo del proyecto.

5.4 Documentación.

Fase a realizar a lo largo de todo el proyecto, cuya importancia será mayor una vez alcanzados los objetivos.

En esta fase se realiza el desarrollo de una memoria borrador, se establece un control de versiones para la memoria y se desarrolla la memoria final del proyecto.

Capítulo 6.
Desarrollo del proyecto.

6.1 Creación paquete ROS.

Los paquetes en ROS son el elemento principal de su arquitectura. Los paquetes pueden contener los procesos de ROS en tiempo de ejecución (nodos), conjunto de datos, archivos de configuración o cualquier elemento que necesitemos. Los paquetes son elemento de construcción más pequeño y el elemento de ejecución de ROS.

Todo el software de ROS está organizado en paquetes. El objeto de los paquetes es que cada paquete pueda ofrecer una funcionalidad determinada de manera que el software se pueda construir y reutilizar. Lógicamente nosotros podemos crear nuevos paquetes, para desarrollar nuestra funcionalidad del cálculo de odometría. Además los paquetes pueden tener dependencias entre sí.

La creación de un paquete es sencilla, pero antes debemos de tener configurada correctamente la variable de entorno ROS_PACKAGE_PATH y un entorno de trabajo donde vamos a crear nuestros paquetes. [15]

Una vez configurado nuestro entorno procedemos a crear un paquete dentro del directorio creado “catkin_ws/src/”, con el comando “catkin_create_pkg”.

En nuestro caso, necesitamos la librería “rtabmap_ros”, que nos aporta todos los métodos necesarios para la obtención de los datos, que copiaremos dentro del directorio “catkin_ws/src/”.

Por ultimo compilamos todo con el comando “catkin_make”, lo cual terminará de crear todo el árbol de directorios que ROS necesita para generar los ejecutables del proyecto.

6.2 Obtención de datos de profundidad.

Una vez configurado todo el entorno de trabajo que nos proporciona ROS a través de sus paquetes, vamos a iniciar el servicio que provee el paquete de librerías OpenNi [16], encargado de publicar todos los datos obtenidos por la Kinect 1.0.

Para iniciar este servicio utilizamos la herramienta "*freenect_launch*", la cual nos permite obtener los datos de las cámaras de la Kinect. "*freenect_launch*" es un paquete basado en "openni_launch". Para ejecutar este paquete utilizamos el comando "*roslaunch*", que es una herramienta que nos permite ejecutar múltiples archivos ROS de manera local o remota a través de SSH.

Su sintaxis sería:

"roslaunch freenect_launch freenect.launch".

Una vez iniciado este comando ya podemos empezar a analizar los datos obtenidos por la Kinect a través de "*rtabmap*"

6.3 Suscripción al topic "*rtabmap*".

Para poder suscribirnos a los datos que nos ofrece el paquete "rtabmap_ros" debemos ejecutar el siguiente comando:

"*roslaunch rtabmap_ros rgbd_mapping.launch rviz:=true rtabmapviz:=false*"

Al ejecutar este comando el paquete "rtabmap_ros" empieza a generar Topics, buses sobre el cual los nodos de ROS se intercambian mensajes, que podemos visualizar a través del siguiente comando:

"*rostopic list -v*"

El topic al cual nos vamos a suscribir es *"/rtabmap/odom/pose/pose*", este topic nos ofrece los datos relacionados con la posición actual de la Kinect.

6.4 Creación de un Subscriber Node.

Una vez suscritos al tópico anterior, necesitamos crear un Subscriber Node [17] para poder utilizar y analizar todos estos datos de odometría que nos ofrece "*rtabmap_ros*".

Para la creación de este nodo se ha optado por utilizar el lenguaje de programación Pyhton. La principal función es la de escuchar los datos de posicionamiento que nos ofrece "*rtabmap_ros*".

El primer paso para la creación de este nodo es crear dentro de nuestro entorno la carpeta /script, que almacenara el archivo .py donde estará programado este Subscriber Node. El siguiente paso es importar dentro del archivo .py la librería rospy con la siguiente sintaxis:

"*import rospy*".

Una vez realizado estos pasos, ya podemos utilizar todas las variables que creamos necesarias de ROS dentro del archivo .py.

6.5 Recepción de mensajes del tipo Odometry.

Para poder trabajar con los mensajes que nos ofrece el nodo "*rtabmap_ros*", en cuanto a la odometría, necesitamos importar la librería Odometry que se encuentra dentro "*nav_msgs.msg*". Esto se haría con la siguiente sintaxis:

"*from nav_msgs.msg import Odometry*"

Una vez realizada esta operación dentro del script, ya podemos trabajar con los mensajes de tipo Odometry.

6.6 Creación algoritmo para el cálculo de la odometría.

Una vez configuradas todas las librerías dentro del script, ya podemos proceder a programar el algoritmo que capture los datos de odometría y no los muestre por pantalla.

Se necesita crear una función llamada "*listener*", en la cual se van a inicializar el node ROS con la siguiente sintaxis:

"rospy.init_node('estimated_trajectory_listener', anonymous=True)"

Después debemos subscribirnos al topic /rtabmap, para poder capturar todos los mensajes necesarios y poder llamar a la función que nos realice todas estas operaciones, la sintaxis seria la siguiente:

"rospy.Subscriber('/rtabmap/odom', Odometry, callbackOdom)"

Con esta línea nos subscribimos al topic *"/rtabmap/odom"* y llamamos a la función "callbackOdom", en la cual se obtendrán y calcularán los datos de odometría. Dentro de esta función se creara una variable "*position*", que contendrá el objeto "*odom_data*", el cual nos ofrece las variables *"position.x, position.y, position.z"*, que nos ubica la Kinect en una posición concreta. Para poder determinar la trayectoria recorrida por la Kinect en un mapa establecido, tenemos que medir en tiempo real la distancia que hay entre dos puntos del plano XYZ [18].

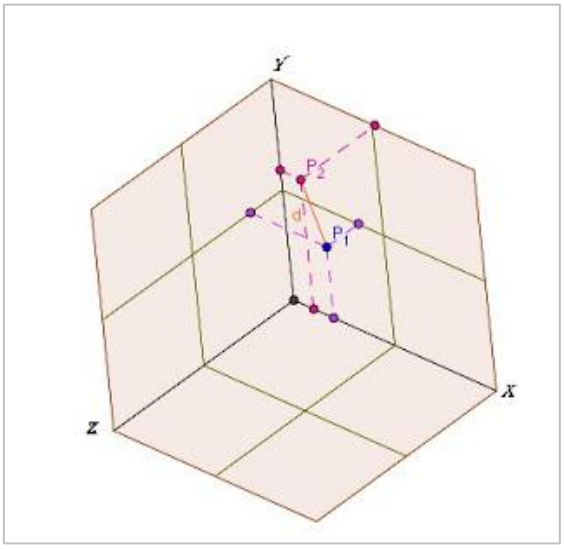


Figura 6.1. Diferencia de dos puntos en un plano 3d

Para poder calcular esta distancia entre dos puntos, se utilizara la siguiente formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

El valor que nos devuelve esta fórmula es la distancia entre dos puntos, por lo tanto para poder calcular la trayectoria recorrida, necesitamos ir sumando todas estas pequeñas distancias para calcular la medida final del recorrido.

6.7 Comprobación del entorno gráfico.

Después de capturar los datos de la odometría y calcular el trayecto, es necesario a través de la herramienta R que nos ofrece el paquete rtabmap, comprobar que nos genera el mapa de la trayectoria recorrida.

Para realizar la comprobación ejecutamos el siguiente comando:

"roslaunch rtabmap_ros rgbd_mapping.launch rtabmap_args:="--delete_db_on_start"".

Una vez ejecutado este comando, se nos abrirá el programa R que nos permite ver tanto los mapas generados, como datos sobre la odometría que son fundamentales para el desarrollo del proyecto.

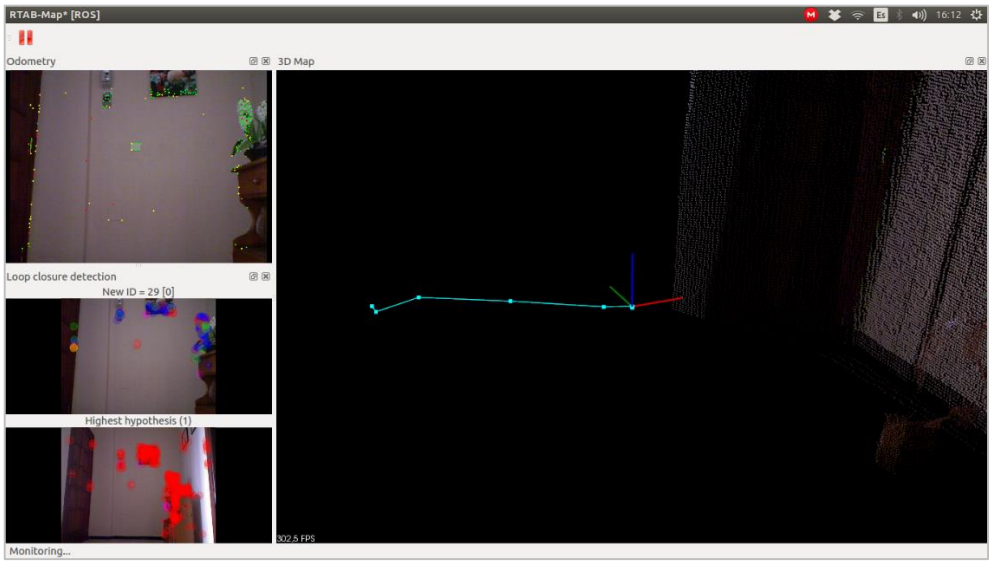


Figura 6.2. Herramienta RTAB-Map.

6.8 Corrección del error en los cálculos de la odometría.

Después de hacer un recorrido y comprobar la trayectoria real con los datos que nos calcula el script, comprobamos que hay una gran cantidad de errores en las mediciones, llegando aproximadamente a 1 metro por cada 10 metros recorridos. Este error se produce con la Kinect, ya que estando sin movimiento, nos ofrece posiciones distintas de casi 5 cm de diferencia, y al sumar las distancias vamos añadiendo error continuamente en la medición de la trayectoria recorrida.

Una vez detectado este error se procede a minimizar, ya que no se puede eliminar, la distancia de error que nos ofrece la Kinect con su sensor de profundidad. La medida adoptada para minimizar este error, es calcular la media de un cantidad de posiciones obtenidas en tiempo real, esto se logra a través de una cola circular por eje x, y, z, en la cual vamos añadiendo los datos de posición y en tiempo real vamos obteniendo las media de la posiciones obtenida. Sabiendo que la Kinect trabaja a 30Hz, 30 imágenes por segundo, se decide aplicar un tamaño de cola de 30 posiciones, una vez comprobado este tamaño se minimiza a la mitad el error obtenido anteriormente, pero todavía sigue añadiendo un error importante, por lo tanto después de ejecutar distintos tamaños de cola, se comprueba que el mejor tamaño, es el doble de las imágenes que toma por segundo, 60 posiciones para cada media. Con este tamaño de cola podemos comprobar que el error se hace casi de aproximadamente 1-2 cm para un recorrido de 10 metros.

6.9 Calculo de la trayectoria recorrida en un circuito.

En este punto se procede a realizar un circuito medido dentro de una sala interior.

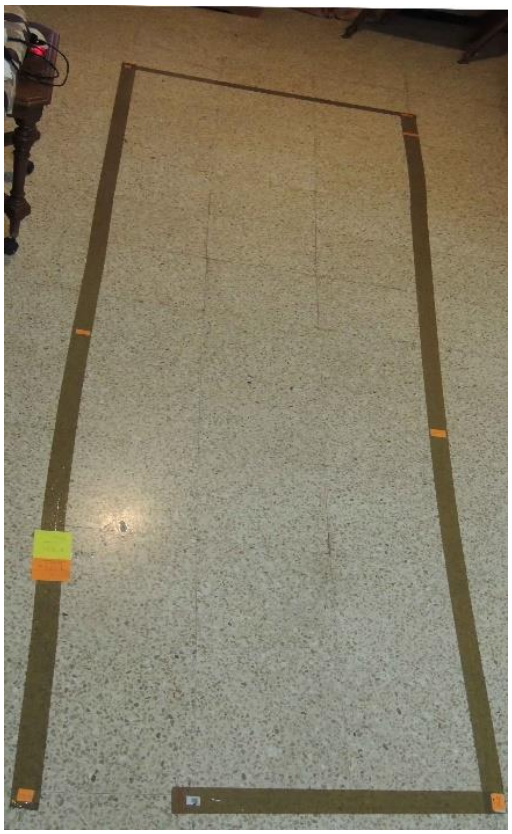


Figura 6.3. Circuito en una sala interior.

Para poder desplazar la Kinect y cometer la menor cantidad de errores en los movimientos, se monta en una mesa con ruedas en la cual también estará el portátil que calculará todos los datos.

Figura 6.4. Carro de transporte.

Después de tener todo el circuito listo, se inicia el proceso de medición de la trayectoria con la Kinect y poder comprobar los datos obtenidos. Se compararán los cálculos sin procesamiento estadístico y con la cola circular de las medias obtenidas.

6.10 Análisis de los datos obtenidos.

Al finalizar los dos recorridos de las trayectorias, obtenemos los siguientes resultados.

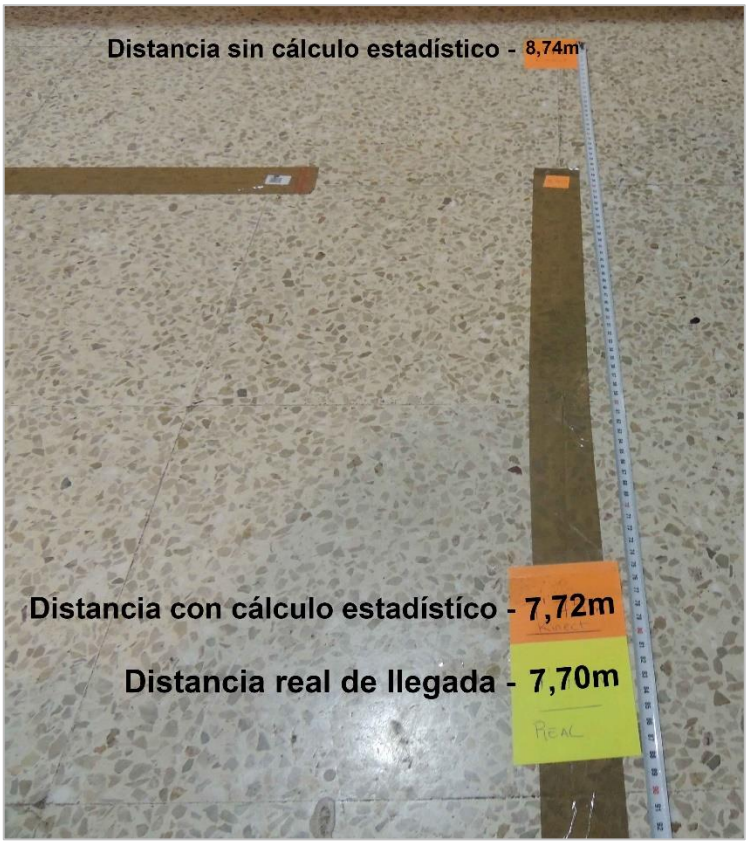


Figura 6.5. Datos obtenidos en el recorrido del circuito.

Como se puede apreciar en la imagen hay una gran diferencia sobres los resultados obtenidos en los dos modos de ejecución del proyecto. Minimizando

a casi 2 cm de error el modo con cálculo estadístico, cola circular con las medias de las posiciones, en distancias de 10 metros.

También con la herramienta que nos ofrece rtabmap, podemos visualizar el recorrido del mapa en 3d y la nube de puntos que genera el paquete.

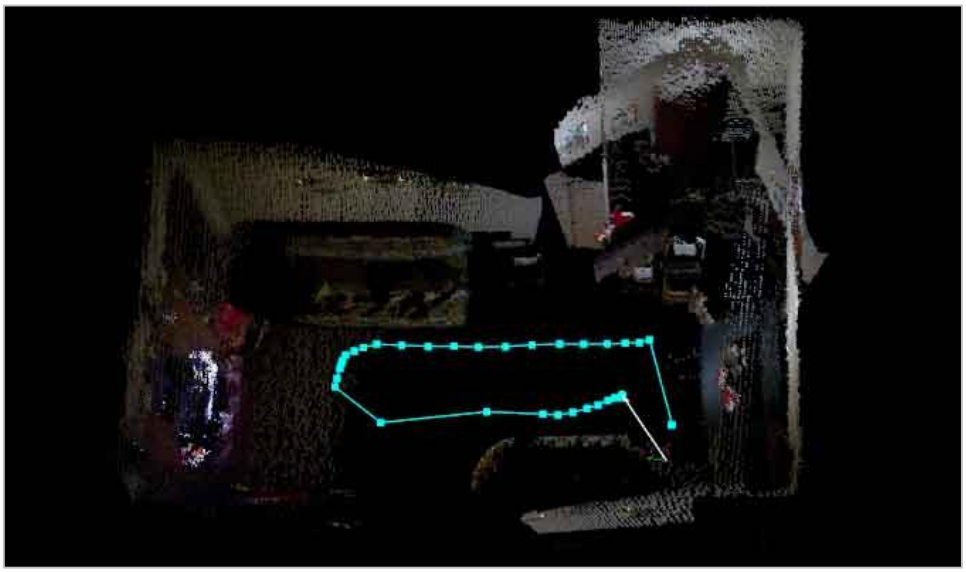


Figura 6.6. Recorrido del circuito en 3D.

Por último se realiza un video con el recorrido en los distintos modos, viendo cómo se calcula a tiempo real la distancia de la trayectoria recorrida.

https://youtu.be/hQJ53zWe6aw

Capítulo 7. Presupuesto

En este capítulo estableceremos en presupuesto sobre el coste total del proyecto en un entorno real de trabajo. Para realizar este presupuesto dividiremos el mismo en tres partes principales: costes de personal, costes de Hardware y costes del Software.

7.1 Costes de personal

Los costes del personal se establecerán según el tiempo de trabajo a las diferentes tareas realizadas y al coste aproximado de una hora de trabajo para un ingeniero informático que es de 15 euros.

En la tabla 7.1 podemos observar con mayor detalle el conjunto de tareas realizadas durante el trabajo, el tiempo aproximado y el coste del mismo.

Tarea	Horas	Costes (€)
Análisis y estudio de la tecnología utilizada (ROS, Kinect, Odometría).	80	1200€
Configuración y pruebas en el Framework ROS.	180	2700€
Programación en Pyhton del script que realice los cálculos del proyecto.	120	1800€
Redacción de la memoria.	60	900€
Total	440	6600€

Tabla 7.1. Relación costes personal.

7.2 Costes de Hardware

En la tabla 7.2 poder observar los costes de los componentes Hardware utilizados durante el desarrollo del proyecto.

Componentes	Costes (€)
Kinect 1.0	70€
Portatil Intel i-7, 8Gb Ram	950€
Carro montaje Kinect	50€
Total	1070€

Tabla 7.2. Relación costes Hardware.

7.3 Costes de Software

En cuanto a los costes del software en el desarrollo del proyecto, siempre se han utilizado software con licencias libres, por lo tanto no incrementa el coste al total del proyecto.

7.4 Coste total

En la tabla 7.3 podemos observar el coste total del proyecto fruto de la suma de los costes de personal y Hardware.

Componentes	Costes (€)
Costes personal	6600€
Costes Hardware	1070€
Total	7670€

Tabla 7.3. Relación de coste total.

Capítulo 8. Conclusiones y trabajos futuros.

En este trabajo se ha llevado a cabo el estudio de la odometría visual en interiores con los sensores que nos provee la Kinect, terminando en un diseño e implementación de un sistema que nos calcula el recorrido en un circuito previamente medido.

Durante la realización del trabajo, se han analizado y estudiado varios algoritmos de odometría para comprobar su eficiencia, dentro de estos algoritmos nos centramos en el RGB-D, con el cual se ha trabajado para el desarrollo del script que calcule las medidas de una trayectoria en un recorrido de un circuito.

Para desarrollar esta aplicación, en primer lugar fue necesaria la formación en el Framework ROS, el cual nos aporta todas las herramientas necesarias para poder interactuar con la Kinect.

El principal inconveniente encontrado ha sido el error que añade en las mediciones el sensor de profundidad de la Kinect. Para poder solucionar este error se ha hecho necesario analizar múltiples escaneos, y aplicar métodos estadísticos para aumentar la precisión de los cálculos.

Como culminación una vez eliminado gran parte del error en la medición que nos añade la Kinect, se ha realizado el recorrido por el circuito analizando los datos obtenidos.

Quedan como líneas abiertas para continuar este trabajo en el futuro:

- Minimizar aún más el error que nos produce la Kinect, con otras medidas estadísticas.
- Contemplar la mezcla de algoritmos para el cálculo del recorrido.

Capítulo 9.
Summary and Conclusions

In this work, I have been carried out the investigation of the visual odometry indoors with Kinect sensors, ending in a design and the implementation of a system that will enable us to calculate the route in a circuit previously measured.

During the conduct of the work, I have been analysed and studied several odometry algorithms to verify its effectiveness, within these algorithms I focused in the RGB-D, which I have worked for the development of the script to calculate the measures of the route in a circuit.

In order to develop this application, first it was necessary the formation in the ROS Framework, which provides us all the tools necessary to be able to interact with the Kinect.

The main disadvantage was the mistake that is made in the measurements in the Kinect depth sensor. To fix this problem it's necessary to analyze different scans, and implement statistical methods to increase the accuracy of calculation.

The culmination, after removing the error of measurements in the Kinect sensor, it has made the route in the circuit analyzing data obtained.

There are different possibilities to continue de work in the future:

- Minimize as much as possible the Kinect sensor error, with others statistical methods
- Complete some algorithms to better calculate of the route.

Apéndice A.
Título del Apéndice 1

A.1. Algoritmo OdometryFinal.py

```
"""********************************************************************************
*
*
* Marcos Luis Díaz García
*
*
* 25/02/2016
*
*
* Conjunto de funciones que permiten calcular la distancia de un recorrido con los
sensores de profundidad de las Kinect en el entorno ROS.
*
*
********************************************************************************/"""
#!/usr/bin/env python
import rospy
from nav_msgs.msg import Odometry
from rtabmap_ros.msg import Info
import math
import pygame
import os
import sys
from pylab import *
from collections import deque
from time import time
```

```
from turtle import *
from numpy import arange

class CircularBuffer(deque):
    def ___init___(self, size=0):
        super(CircularBuffer, self).___init___(maxlen=size)
    @property
    def average(self):  # TODO: Make type check for integer or floats
        return sum(self)/len(self)

"""
    Variables globales
"""
TamArray = 2
Cx = CircularBuffer(size=TamArray)
Cy = CircularBuffer(size=TamArray)
Cz = CircularBuffer(size=TamArray)
camina = False
MapFinish = False
i = 1
antx = 0
anty = 0
antz = 0

distance= 0.0
start_time=0
dtotal=0.0
```

```
def rango (X,tam):
    X1 = arange(X-tam, X+tam, 0.1)
    Y.append(X1[0])
    Y.append(X1[len(X1)-1])
    return Y

def load_array():
    rospy.Subscriber('/rtabmap/odom', Odometry, callbackOdom)
    position = Odometry.pose.pose.position
    orientation = Odometry.pose.pose.orientation
    Cx.append(position.x)
    Cy.append(position.y)
    Cz.append(position.z)

def callbackOdom(odom_data):
    #Variable initialization
    global dtotal
    global distance
    global start_time
    global camina
    global i
    global antx
    global anty
    global antz
    global MapFinish
```

```
        position = odom_data.pose.pose.position
        orientation = odom_data.pose.pose.orientation

        #si no lost odometry
        if position.x != 0.0 and position.x != 0.0 and position.x != 0.0:

            distance = math.sqrt(((((Cx.average)) - antx)**2) + ((((Cy.average))
- anty)**2) + ((((Cz.average)) - antz)**2))
            antx = Cx.average
            anty = Cy.average
            antz = Cz.average
            if i > TamArray:
                dtotal = distance + dtotal

            Cx.append(position.x)
            Cy.append(position.y)
            Cz.append(position.z)

            if dtotal >= 0.2:
                start_time = time()
                camina = True

            os.system('clear')
            print "iteracion: ", i
            print "Distancia recorrida: ", distance
            print "Distancia total: ", dtotal
            print "Xant: ",(antx)
            print "X: ",position.x
```

```
        print "Yant: ",(anty)
        print "Y: ",position.y
        print "Zant: ", (antz)
        print "Z: ",position.z
        print "Camina:", camina
        i = i + 1
    else:
        print "Lost odometry"

def myhook():
    print "shutdown time!"

def callbackLoop(info_data):
    loopC = info_data.loopClosureId
    if loopC > 0:
        print "Mapa Cerrado !!!!!"
        print "Se han recorrido %5.4f. metros"% dtotal
        MapFinish = True
        rospy.signal_shutdown(myhook)

def listener():
    """
        Subscribe to the topic: /rtabmap/odom
                Data type: nav_msgs/Odometry
                Method called: callback(Odometry)
    """
    rospy.init_node('estimated_trajectory_listener', anonymous=True)
```

```
    rospy.Subscriber('/rtabmap/odom', Odometry, callbackOdom)
    rospy.Subscriber('/rtabmap/info', Info, callbackLoop)
    # Mantiene a python activo mientras el RosNode no se pare
    rospy.spin()

if ___name___ == '___main___':
    #Variable initialization
    listener()
```

Bibliografía

[1] Efficient Variants of the ICP Algorithm, https://graphics.stanford.edu/papers/fasticp/fasticp_paper.pdf

[2] Libfovis, https://fovis.github.io/

[3] Peter Henry. RGB-D Mapping: Using Kinect-Style Depth Cameras for Dense 3D Modeling of Indoor Environments.

[4] Dataset Download, http://vision.in.tum.de/data/datasets/rgbd-dataset/download.

[5] Arqhys Arquitectura, http://www.arqhys.com/contenidos/quees-odometria.html

[6] Wiki ROS. http://wiki.ros.org/es.

[7] Kinect (página official). http://www.xbox.com/es-ES/kinect

[8] Kinect for developers, http:// www.kinectfordevelopers.com /es /2012 /03 /01 / diferencias-entre-kinect-xbox-360-y-kinect-for-windows/

[9] KinectSDK, http://elbruno.com/2012/02/05/kinectsdk-un-par-de-detalles-sobre-el-nuevo-near-mode-y-sobre-como-funciona-con-el-sensor-de-profundidad/

[10] Wikipedia, https://es.wikipedia.org/wiki/Odometría

[11] Encoder, https://sites.google.com/site/proyectosroboticos/encoder

[12] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments," The Int. Journal of Robotics Research, 2012.

[13] Ubuntu 14.04. *http://www.ubuntu.com/download.*

[14] ROS Indigo. *http://wiki.ros.org/indigo/Installation/Ubuntu*

[15] Creación de paquetes en ros, http://wiki.ros.org/ROS/Tutorials/catkin/CreatingPackage

[16] WikiRos, http://wiki.ros.org/openni_launch

[17] ROS,
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)

[18] Puntos en el espacio, http://distanciapuntos enelespacio .blogspot.com.es/2010/04/como-calcular-la-distancia-entre-dos_19.html

[19] C. Audras, A. I. Comport, M. Meilland, and P. Rives, "Real-time dense RGB-D localisation and mapping," in Australian Conf. on Robotics and Automation, (Monash University, Australia), December 2011.