Trabajo de Fin de Grado

Sistema de posicionamiento para un drone mediante odometría visual

Positioning system for a drone using visual odometry

José Oliver Martínez Novo

La Laguna, 7 de julio de 2015

D. **Jonay Tomás Toledo**, con N.I.F. 78.698.554-Y profesor contratado Doctor del 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:

"Sistema de posicionamiento para un drone mediante odometría visual"

ha sido realizada bajo su dirección por D. **José Oliver Martínez Novo**, con N.I.F. 43.826.914-T.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 7 de julio de 2015.

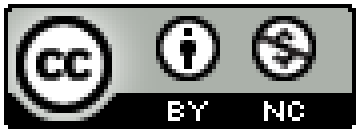
Agradecimientos

Jonay Tomás Toledo, por su total apoyo en este proyecto.

Compañeros de la promoción 2011-2015, por haber podido estar juntos, ayudándonos siempre, tanto en los buenos como en los malos momentos.

Quiero agradecer en especial a mi familia, a mi madre María Novo Paz, a mi hermano Juan Martínez Novo, y a mi pareja Netey Sosa Fleitas, por el apoyo, tanto emocional como económico, que me han brindado, de forma incondicional, durante todos estos años de universidad.

Licencia



Resumen

El objetivo de este trabajo es desarrollar un sistema de posicionamiento 3D para un vehículo aéreo no tripulado, cuyo espacio de trabajo estará en interiores, como naves industriales, donde no sea posible hacer uso del GPS para localizar al vehículo.

Se hará uso de la cámara Kinect 1.0 desarrollada por Microsoft, como dispositivo de bajo coste, para obtener los datos de profundidad del entorno.

Esto será posible bajo sistemas Linux gracias al driver de software libre "OpenNi", que permite tener acceso a las funciones de la cámara Kinect, y publicar las imágenes de profundidad.

El desarrollo de software usará la librería "DepthImage_To_LaserScan", que permite generar y publicar escaneos de la imagen de profundidad, y la librería "Laser_Scan_Matcher", que permite calcular odometría 2D a partir de los escaneos generados con la librería anterior.

La creación de una odometría 3D será el resultado de la modificación de la librería "Laser_Scan_Matcher", y de la creación de un programa que integre toda la funcionalidad del sistema y mejore la precisión usando técnicas estadísticas.

Palabras clave: *Odometría, Kinect, OpenNi, DepthImage_To_LaserScan, Laser_Scan_Matcher*

Abstract

The objective of this work is to develop a 3D positioning system for a drone, whose workspace is indoors, as warehouses, where it is not possible to use GPS to locate the vehicle.

It will be using the Kinect camera, developed by Microsoft, such as low cost device to obtain depth data of the environment.

This will be possible in Linux thanks to the open source driver "OpenNI" which allows access to the Kinect camera functions, and publish depth images.

The software development will use the library "DepthImage_To_LaserScan", that allow generate and publish scans of the image depth, and the library "Laser_Scan_Matcher", that allow calculate 2D odometry from scans generated by previous library.

Create a 3D odometry will be the result of make changes in the library "Laser_Scan_Matcher", and the creation a program that integrate all system functionality, and improves accuracy using statistics techniques.

Keywords: *Odometry, Kinect, OpenNi, DepthImage_To_LaserScan, Laser_Scan_Matcher*

Índice General

Índice de figuras

Índice de tablas

Capítulo 1.
Introducción

1.1 Sistemas de posicionamiento actuales

En la actualidad los sistemas de posicionamiento y guiado para vehículos no tripulados basan sus decisiones en los resultados obtenidos por diferentes tipos de dispositivos de medición y localización.

El uso del GPS como instrumento de localización facilita la tarea dado que es capaz de acotar la ubicación del objeto con márgenes de error centimétricos en el mejor de los casos, siempre y cuando se cuente con un dispositivo con dicha precisión que, por lo general, suelen ser de elevado coste.

1.2 Problemática

El verdadero problema aparece cuando se pretende trasladar la tecnología actual a dispositivos cuya misión sea navegar en espacios interiores donde no sea posible utilizar el GPS, siendo las distancias cortas y con obstáculos, como puede ocurrir en cualquier tipo de nave industrial o dependencia en oficinas, casas, etc.

1.3 Solución

Una vez descartado el uso del GPS la solución radica en encontrar algún sistema que permita al vehículo saber en todo momento donde se encuentra. Para ello es necesario conocer de forma precisa cuanto se ha desplazado en el espacio y cuál es su orientación por unidad de tiempo.

Dado que el vehículo es aéreo, es importante conocer en todo momento su orientación, ya que éste, puede permanecer perfectamente inmóvil sin desplazarse con respecto a los ejes, pero haberse girado.

1.4 Inconvenientes

Los drones, están formados por un número mayor o igual a tres, de motores que giran en sentidos opuestos por parejas para evitar la pérdida de control por el

giro de las hélices. Este fenómeno se corrige en los helicópteros normales con el conocido rotor de cola.

A pesar de disponer de motores de alta calidad que garanticen una precisión en cuanto al número de vueltas que giran por minuto, siempre existe un margen de error entre las velocidades de giro que, por pequeño que sea, se irá acumulado en el tiempo dando lugar a la deslocalización de drone por lo que se conoce como efecto de deriva.

1.5 Sistemas de medición actuales

Actualmente existen sistemas de medición laser pero suelen tener un precio poco asequible y los más económicos proporcionan medidas de distancia en un solo punto.

Existen dispositivos de medición laser de barrido de 360º, que trabajan a gran velocidad y proporcionan hasta 64 lecturas verticales en cada recepción de datos.

También existen los sensores de ultrasonido que, por lo general, son de bajo coste. Entre sus inconvenientes tenemos que tardan demasiado tiempo en completar el ciclo de (emisión - recepción), y que su señal se ve interferida por la frecuencia generada por los motores.

Capítulo 2. Antecedentes

2.1 Vuelo con control de visión

En Suiza, en el IDSC, Institute for Dynamic Systems and Control, [5], se ha hecho un trabajo basado en vuelo de precisión para interiores, donde, a través de 8 cámaras situadas en el techo se obtiene una precisión milimétrica para el vuelo.

Gracias a la información recibida por las ocho cámaras, un drone podría pasar por pasillos o zonas estrechas a gran velocidad.

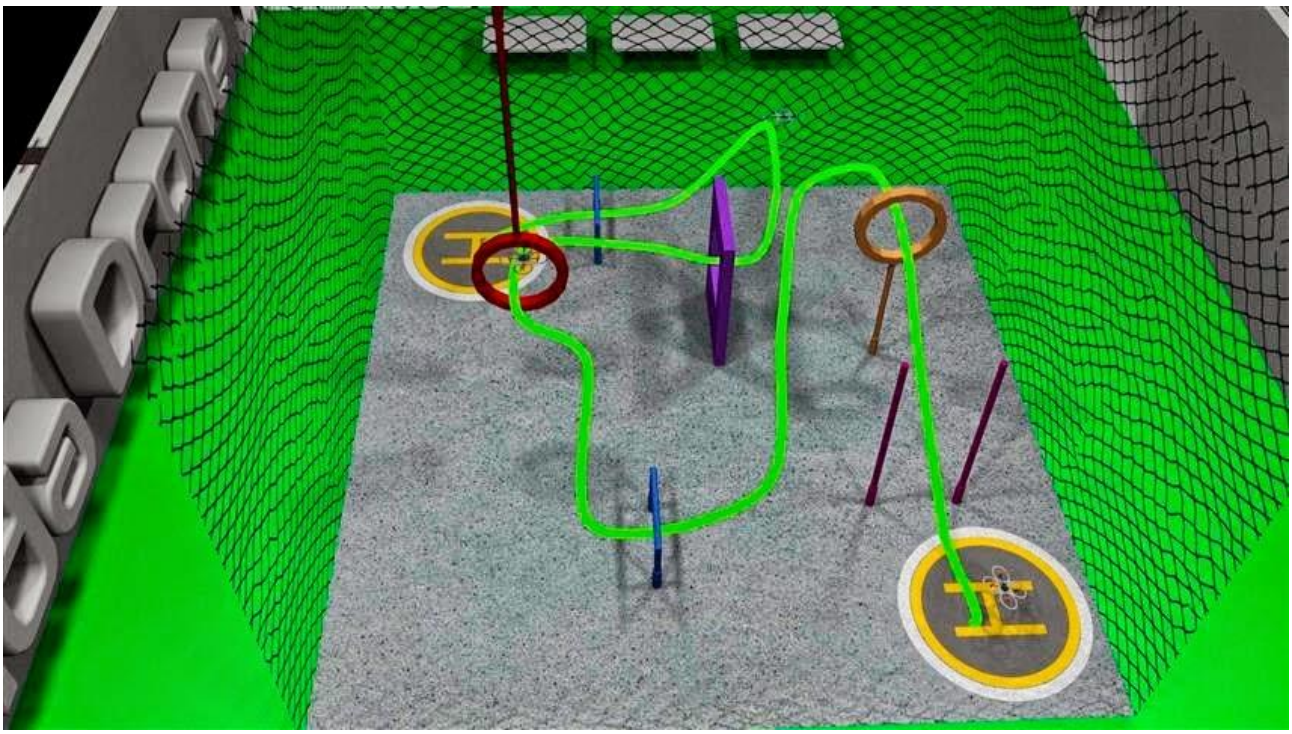


Figura 2.1. Circuito a recorrer por un mini drone

2.2 Vuelo colectivo sincronizado

Otras líneas de investigación se centran en el trabajo colaborativo en interiores, donde los drones son guiados por un ordenador central que es quien toma las decisiones permitiendo, de esta forma, reducir considerablemente la necesidad de capacidad de cómputo de los drones.

Las pruebas son realizadas en una habitación que, igual que el trabajo del punto anterior, también tiene un conjunto de sistemas de cámaras que permite conocer con milímetros de precisión donde esta cada uno de los drones en tiempo real.

Finalmente el vuelo coordinado se consigue mediante el uso de algoritmos de navegación en formación y técnicas de generación de trayectorias libres de colisión.

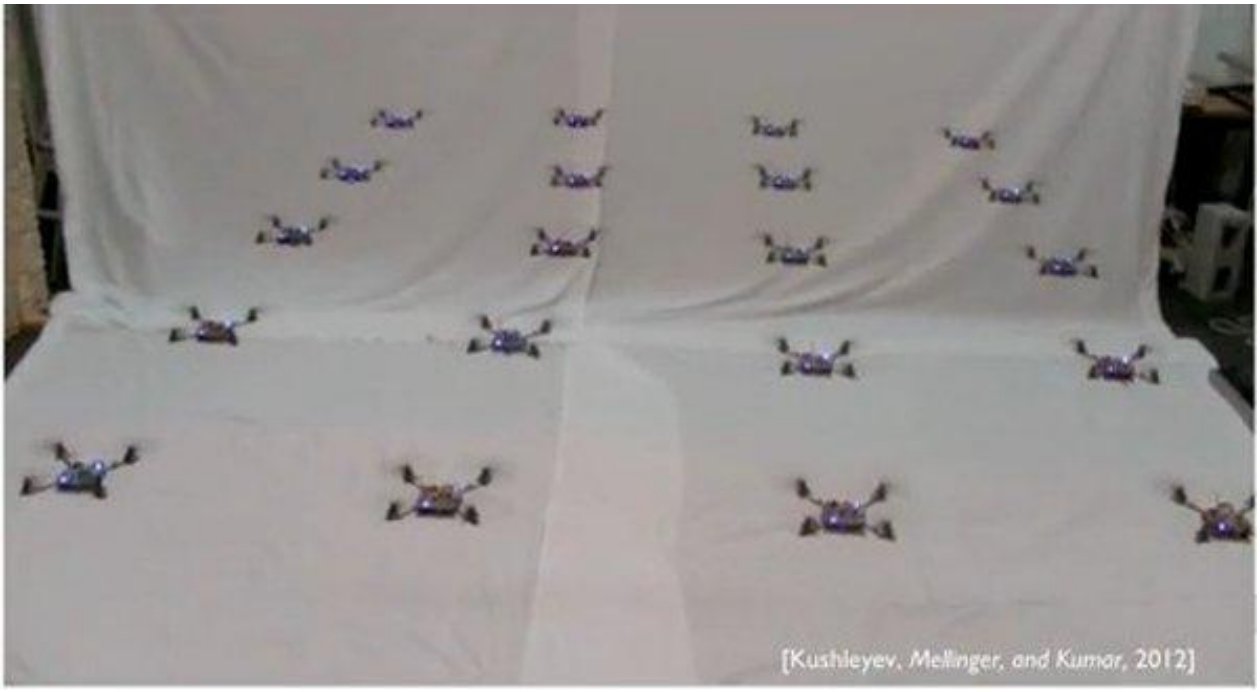


Figura 2.2. Escuadrón de drones coordinados

2.3 Odometría visual con RGB-D

Otra línea de desarrollo combina la información de profundidad con la de color, mediante una técnica llamada “RGB-D”, que analiza todos los grados de libertad de la imagen, juntando los valores de color con los de profundidad para dar una solución de odometría 3D en tiempo real.

Este tipo de desarrollo no es posible implementarlo en la placa Odroid U3, [2], dada la capacidad computacional necesaria para su ejecución.

Capítulo 3.
Elementos de desarrollo

Para emprender este tipo de proyectos en los que es necesario obtener información en tiempo real se necesita contar con hardware y herramientas software que permitan su desarrollo.

A continuación se describen las principales herramientas software y dispositivos necesarios.

3.1 Robot Operating System (ROS)

Es un sistema operativo que provee librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots.

Un sistema ROS [7], se compone de un número de nodos independientes, cada uno de los cuales se comunica con los demás nodos utilizando un modelo de (publicación / suscripción) de mensajería.

Los nodos publican o consumen información, y lo hacen a través de tópicos con nombres definidos. Estos tópicos se usan como canal de comunicación de mensajes entre los diferentes nodos.

Este modelo permite que la información capturada por cada nodo pueda ser consumida por el resto de nodos o, por otros sistemas de más alto nivel.

Los nodos en los sistemas ROS no tienen por qué ser basados en el mismo sistema o arquitectura, siendo posible tener una placa Arduino [8], publicando mensajes, y un teléfono Android [9], consumiéndolos con una aplicación.

Esto hace a los sistemas ROS altamente flexibles y adaptables a las necesidades del usuario.

Entre otras, ROS provee las siguientes características:

- Abstracción de hardware.
- Controladores de dispositivos.
- Librerías.
- Herramientas de visualización.
- Comunicación por mensajes.
- Administración de paquetes.

3.2 Odroid U3

ODROID es una serie de computadoras de una sola placa, y computadoras de tipo tablet creada por Hardkernel Co., Ltd., una empresa de hardware ubicada en Corea del Sur. A pesar de que el nombre 'ODROID' es un acrónimo de 'abierto' + 'Android', el hardware no es realmente abierto porque algunas partes del diseño no son publicadas por la empresa.

El modelo elegido para el desarrollo del proyecto es el U3, ya que, además de su reducido tamaño, 83 × 48 mm, y un peso de 48g incluido el disipador de calor cuenta, entre otros componentes, con un microprocesador QuadCore a 1.7GHz, slot para tarjeta microSD y modulo eMMC, puertos USB 2.0, conexión Ethernet 10/100, y es compatible con sistemas operativos Android/Ubuntu.

Figura 3.1. Placa Odroid U3

Para la instalación del sistema operativo, el fabricante recomienda el uso de módulos eMMC que actualmente ofrecen una capacidad de hasta 64GB y permiten sacar el máximo rendimiento al sistema. [2]

A continuación se muestra la ficha técnica de características.

CPU	1.7GHz **Exynos4412 Prime** Cortex-A9 Quad-core processor with PoP (Package on Package) 2Gbyte LPDDR2 880Mega Data Rate
PMIC	MAX77686 Power Management IC from MAXIM
HSIC USB 2.0 Hub	USB3503A Integrated USB 2.0-compatible hub / HSIC upstream port from SMSC/Microchip
HSIC Ethernet controller	LAN9730HSIC USB 2.0 to 10/100 Ethernet controller with HP Auto-MDIX from SMSC/Microchip
Audio CODEC	MAX98090 is a full-featured and high performance audio CODEC from MAXIM
Protection IC	NCP372 Over-voltage, Over-current, Reverse-voltage protection IC from OnSemi.
USB Load switch	NCP380 Protection IC for USB power supply from OnSemi.
HDMI conditioner	IP4791CZ12 HDMI transmitter interface protector with level shifter from NXP
HDMI connector	Standard Micro-HDMI, supports up to 1920 x 1080 resolution
Connectivity	USB Host x 3, Device x 1, Ethernet RJ-45, Headphone Jack
IO Ports	GPIO, UART, I2C, SPI(Board Revision 0.5 or higher)
Storage Slot	Micro-SD slot, eMMC module connector
DC Input	5V / 2A input, Plug specification is inner diameter 0.8mm and outer diameter 2.5mm

Tabla 3.1. Características técnicas de Odroid U3

3.3 ArduCopter - APM:Copter

Figura 3.2. Imagen promocional ArduCopter.

Arducopter [3], actualmente APM:Copter es un software controlador UAV multicopter, de código abierto, que ganó el Sparkfun 2013 y 2014, competición de vehículos autónomos. Un equipo de desarrolladores de todo el mundo están constantemente mejorando y perfeccionando el rendimiento y las capacidades de APM:Copter.

Incluye un planificador de vuelo que permite crear rutas marcando puntos en un mapa de Google Maps. También provee un simulador de vuelo que permite crear y

almacenar tanto rutas como misiones específicas que se pueden seleccionar en los menús desplegables que ofrece la interfaz gráfica.

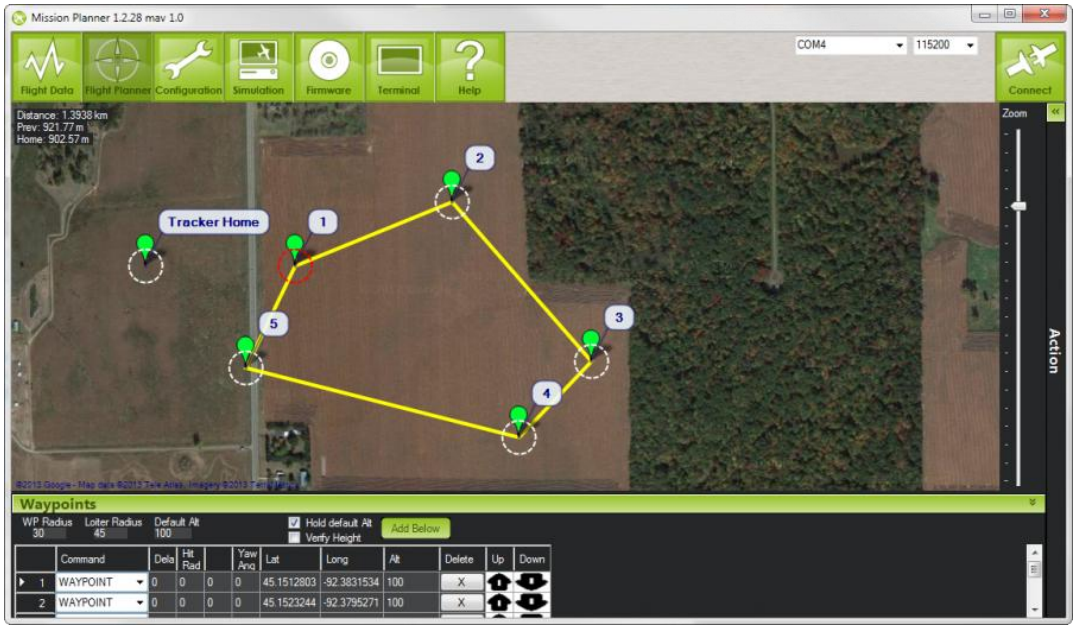


Figura 3.3. Planificador de rutas.

3.4 Kinect 1.0

Kinect 1.0 [1], es la primera versión del controlador de videojuegos desarrollado por Microsoft para la consola Xbox 360.

Figura 3.4. Cámara Kinect 1.0.

Entre otras características, dispone de un emisor de infrarrojos, un sensor de profundidad, cámara RGB 640x480 pixels y un array de micrófonos.

El sensor de profundidad está adaptado a la resolución de la cámara RGB, por lo que las imágenes de profundidad que ofrece son también una matriz de 640x480.

Ofrece un ángulo de visión horizontal de 57º y un ángulo vertical de 43º. Su rango de detección de profundidad oscila entre los 0.45 y los 4.5 metros.

Su frecuencia de trabajo es configurable, pero por defecto son 30Hz tanto para las imágenes RGB como para las de detección de profundidad.

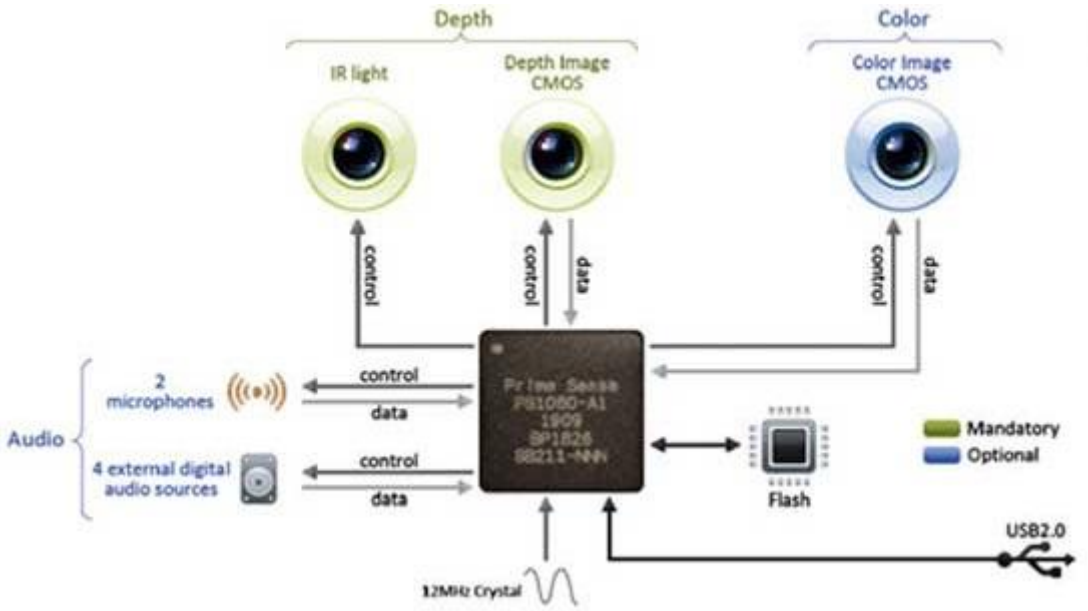


Figura 3.5. Cámara Kinect 1.0 - Esquema.

Capítulo 4.
Limitaciones del sistema

Además de los inconvenientes ya mencionados, cabe destacar la necesidad de capacidad de cómputo por parte de la placa Odroid, y la necesidad de precisión y alcance por parte del sensor kinect.

4.1 Capacidad de cómputo de Odroid U3

Centrándonos en su capacidad computacional concretamos que la placa Odroid U3 dispone de un procesador Exynos 4412 ARM Cortex-A9 Quad Core 1.7GHz, que a pesar de trabajar con 4 núcleos a una buena frecuencia, su rendimiento puede verse comprometido bajo condiciones de tiempo real.

Se comprueba en el laboratorio que el sistema en funcionamiento provoca un uso de CPU del 100% para los cuatro procesadores.

Esta sobre carga de trabajo crea retrasos en el envío y/o recepción de los mensajes provocando que la información final no se encuentre disponible en tiempo real.

4.2 Precisión del sensor kinect 1.0

En las pruebas de laboratorio realizadas durante el estudio inicial del proyecto se constató que el sensor de profundidad tenía un margen de error de un centímetro, concretamente, a 2.48 metros el margen de error fue de ±13mm.

Además se observó que en condiciones ideales de reposo, que se producían oscilaciones en las lecturas para un mismo punto.

Una vez observadas estas oscilaciones en su conjunto mediante un representador gráfico del sistema operativo, se detectan tramos del mapa generado que oscilan en un pequeño margen de distancias.

Por último indicar la baja potencia de la luz emitida por el emisor infrarrojo es interferida por la luz ambiental haciendo, en algunos casos, imposible que el sensor de profundidad pueda detectarla.

Todo ello complica bastante los cálculos para la obtención de un resultado óptimo dado que son necesarias más muestras y aplicar cálculo estadístico.

4.3 Alcance del sensor Kinect 1.0

Dado que la odometría basa sus cálculos en la cantidad de información que el sensor es capaz de detectar, se observa que, en condiciones de laboratorio, el alcance del sensor deja muchas zonas sin detectar provocando que haya mediciones erróneas.

Estás mediciones son descartadas a la espera de mediciones con un mayor número de datos que puedan ser procesadas.

A mayor número de mediciones erróneas, mayor es el tiempo transcurrido desde la última medición correcta y esto provoca posteriormente, en el desarrollo del proyecto, que los cálculos de odometría, aún siendo correctos, tengan un mayor error.

A continuación se muestra una imagen indicando los modos de funcionamiento del sensor kinect:

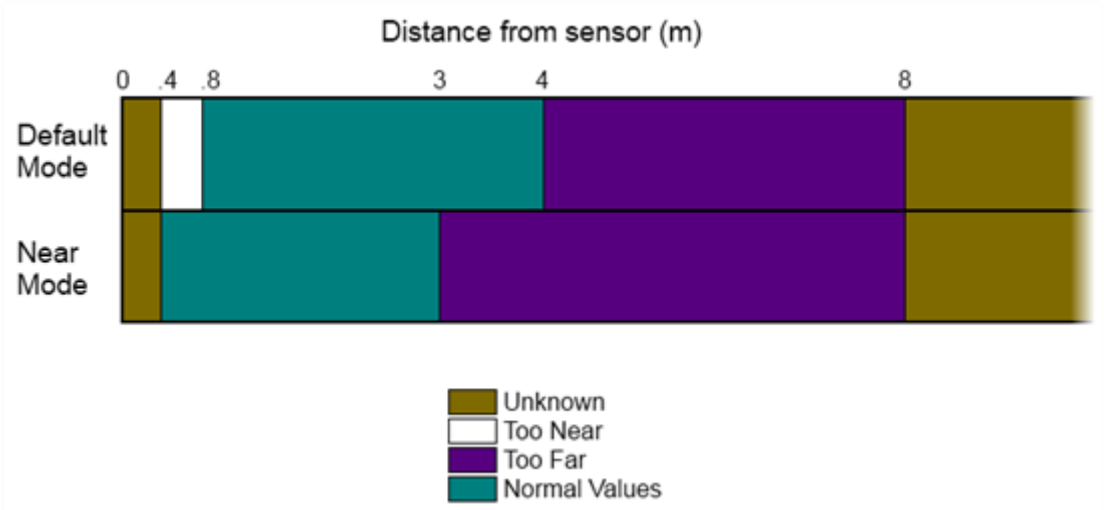


Figura 4.1. Modos de distancia admitidos por Kinect

Capítulo 5.
Objetivos

Para poder considerar viable la solución planteada, el sistema debe permitir principalmente tener acceso a la información en los siguientes aspectos:

5.1 Cálculo de distancias

Poder calcular la distancia entre el drone y el resto de objetos de su entorno con el menor margen de error posible, y un alcance que cubra su área de trabajo.

5.2 Corrección del efecto deriva

Poder detectar y corregir el efecto de deriva producido por el error en la velocidad de giro de los motores, en situaciones en las que el drone deba permanecer inmóvil, suspendido en el aire, por un cierto periodo de tiempo.

5.3 Localización del drone

Conocer en todo momento la posición del drone, bien por odometría, o bien por su representación en un mapa.

5.4 Cálculo del desplazamiento

Saber cuánto se ha desplazado en cada tramo de su trayectoria para escenarios en los que haya que realizar recorridos lineales como por ejemplo en el interior de los pasillos de estanterías de almacenes industriales, o bibliotecas, etc.

Ante todo, el verdadero reto es conseguir precisión en el sistema, y para ello será necesario contar con un buen sensor que ofrezca datos de gran precisión y velocidad, al mismo tiempo que una buena capacidad computacional para poder atender toda esa información en tiempo real.

Capítulo 6.
Fases del proyecto

6.1 Análisis

Mediante el desarrollo de este proyecto se pretende obtener una odometría 3D, en tiempo real, haciendo uso de herramientas software que sólo ofrecen odometría 2D.

El proceso comenzará con la obtención de las imágenes de profundidad publicadas por la librería "OpenNi" [10], la que es capaz de acceder a los datos del sensor de la cámara kinect, crear las imágenes con datos de profundidad y publicarlas.

Las imágenes de profundidad, son necesarias, pues interesa saber a qué distancia están las cosas que componen el entorno, para poder calcular cuánto se ha desplazado el dispositivo, con el análisis continuo de dichas imágenes.

Como procesar toda la imagen completa sería muy costoso, computacionalmente hablando, se hará uso de la librería "DepthImage_To_LaserScan" [11], con la que se podrá crear y procesar una fila de datos, "LaserScan", de esa imagen de profundidad.

Dado que el sensor kinect presenta un margen de error centimétrico, y oscilaciones en las mediciones para puntos fijos, se crearán tres "LaserScan", a diferentes alturas, para poder generar tres odometrías distintas, y de esta forma, poder obtener las componentes 2D que tengan mayor precisión de las tres odometrías.

Para poder obtener esta precisión en las mediciones de la odometría, será necesario modificar la librería "Laser_Scan_Matcher" [12], encargada del cálculo de la odometría 2D, para añadir los valores de las covarianzas de las componentes 2D en un nuevo tipo de dato que incluya la Pose 2D ya calculada y sus covarianzas.

Del mismo modo, para obtener la odometría 3D, se crearán tres escaneos verticales, también de tipo "LaserScan", que la librería interpretará como horizontales, pero si nos hacemos a la idea que se gira la cámara 90º, el movimiento que antes era (izquierda – derecha), ahora, con la cámara girada, se convierte en el movimiento (arriba - abajo), y con esto obtenemos la componente "z", de elevación, que nos faltaba para nuestro sistema 3D.

6.2 Configuración del sistema operativo

Para el desarrollo del proyecto se utilizará un PC de escritorio al que se instalará el sistema operativo Linux en su distribución "Ubuntu 14.04.2 TLS" disponible en [13].

Sobre el sistema operativo se instalará "ROS" en su versión "Indigo" que habilitará, entre otras, las características de publicación y subscripción de mensajes que se usará en todo el proyecto.

El proceso de instalación de Ros está disponible en [14].

6.3 Instalación de librerías

Entre otras librerías y paquetes que son necesarios instalar, haremos hincapié en aquellas de mayor relevancia para el proyecto.

- **OpenNi:** es un conjunto de librerías de código abierto desarrolladas para tener acceso a las características de la cámara kinect 1.0 y publicar imágenes de profundidad.
  Este paquete se puede encontrar en: [10].
- **DepthImage_To_LaserScan:** librería que provee los tipos de datos y mecanismos para poder crear un escaneo "LaserScan", de una imagen de profundidad.
  Esta librería está disponible en: [11].
- **Laser_Scan_Matcher:** esta librería es la encargada del cálculo de odometría mediante la comparación de escaneos laser consecutivos.
  Esta librería está disponible en: [12].

6.4 Obtención de los datos de profundidad

Inicialmente, tras haber configurado todo el sistema se creará un proyecto que haga uso de la librería "DepthImage_to_LaserScan" para trabajar con la imagen de profundidad para crear y publicar los escaneos laser de dicha imagen.

6.5 Publicación de múltiples escaneos

Una vez obtenida la imagen de profundidad, una matriz de 640x480 de tipo uint8[], que representan las distancias de cada "pixel" al sensor, se toman tres filas y tres columnas de dicha matriz.

Con esas filas y columnas se generarán seis mensajes distintos, de tipo "Laser_Scan", tres horizontales y tres verticales, pero todos ellos con la misma marca de tiempo, pues interesa que se calculen seis odometrías diferentes para poder obtener la mejor de ellas en cada caso.

Originalmente se pensó en tomar esas filas y columnas de forma contigua, para tener tres medidas de la misma zona con una separación de un pixel, a efectos de calcular una moda, o en su defecto, una media, y crear un "LaserScan" de mayor precisión. Posteriormente esta idea se descarta al hacer uso de una matriz de covarianza para aumentar la precisión de la odometría resultante.

6.6 Generación de múltiples odometrías 3D

Con los tres mensajes de datos horizontales se pretende obtener el mejor valor de odometría para las componentes:

- "x", que representa el movimiento de avance o retroceso.
- "y", que representa el movimiento de izquierda a derecha.
- "theta" que representa el giro o "guiñada", (yaw).

Con los otros tres mensajes que contienen datos verticales se pretende obtener el mejor valor de las componentes que faltan.

"z", que representa el movimiento arriba - abajo, y será el valor "y" en odometrías procedentes de escaneos verticales a los que se aplica un giro de 90º.

"Pitch", que representa el ángulo de alabeo, y será el valor de "theta" en odometrías procedentes de escaneos verticales a los que se aplica un giro de 90º.

Los escaneos verticales son tratados por las librerías como si fueran horizontales, pero como sus datos son columnas verticales de la imagen de profundidad, es como si implícitamente se aplicara un giro de 90º al sistema.

Por este motivo, se indica en la tabla siguiente la relación que existe entre las componentes de las odometrías según provengan de escaneos horizontales o verticales.

Horizontales Componentes	Verticales Componentes
X	X
Y	Z
Thetha	**Pitch**

Tabla 6.1. Relación de las componentes entre odometrías fruto de escaneos horizontales y verticales

Por último faltaría por obtener el ángulo de “alabeo”, para tener todas las componentes del sistema pero con los datos disponibles es imposible obtener este ángulo.

6.7 Cálculo de la odometría 3D resultante

A medida que se vayan obteniendo los datos de las odometrías 2D se hará un análisis de sus covarianzas para determinar el valor con mayor precisión para cada componente.

Por último se actualizarán los valores de la odometría local que se publicará como resultado del sistema en un tópico llamado “kinect_Odometry”, mediante un mensaje de tipo “Pose3D”.

6.8 Integración del sistema

Será necesario instalar todos los componentes software mencionados en este capítulo en la placa Odroid U3.

Se deberá disponer de una tarjeta MicroSD, a ser posible de clase 10, dado la capacidad de transferencia de datos entre la memoria y la MicroSD que es de 25MB/s.

Se aconseja usar un modulo eMMC como medio de almacenamiento ya que su velocidad de transferencia de datos es mucho mayor permitiendo aprovechar al máximo el rendimiento de la placa Odroid U3.

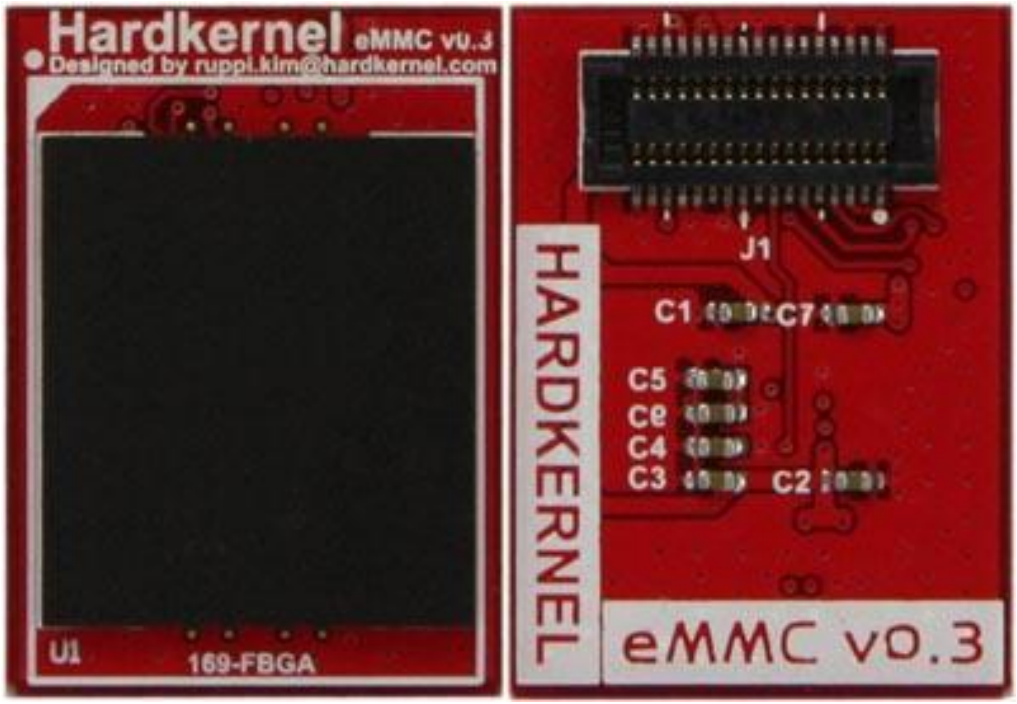


Figura 6.1. 16GB eMMC V5.0 for Odroid U3

Figura 6.2. 16GB eMMC V5.0 to MicroSD Adapter

Capítulo 7.
Cálculo de la odometría

El proceso del cálculo de odometría comienza con la imagen de profundidad obtenida por la librería "OpenNi", con los datos que ofrece la cámara Kinect.

Cada pixel es un número entero que indica la distancia entre ese pixel de la imagen y el sensor.

Un escaneo o "LaserScan", es un tipo de mensaje que contiene los datos de profundidad de una fila horizontal, de esa imagen, originalmente la central, pero puede ser otra, incluso más de una.

El proceso comienza cuando la librería "Laser_Scan_Matcher" obtiene el primer escaneo que coloca como escaneo base hasta recibir uno nuevo que comparará con el escaneo anterior.

En cada escaneo se identifican sucesiones de puntos, en forma de distancias. Esas sucesiones de distancias luego se buscarán en el escaneo base aplicando un umbral.

Localizadas las sucesiones de distancias en ambas imágenes se calculan las distancias entre todas las parejas de puntos para determinar los valores de avance o retroceso, desplazamiento lateral, y giro de guiñada, "yaw".

Una vez obtenidos los valores, por defecto, se publica un mensaje de tipo "Pose2D", indicando la odometría.

Después de publicar el mensaje de odometría el último escaneo que se comparó pasara a ser el escaneo base y el proceso se repetirá de nuevo a la espera de un nuevo escaneo entrante.

Recordar que la librería "Laser_Scan_Matcher" sólo ofrece cálculo de odometría 2D. El proceso para obtener una odometría 3D con esta librería se tratará con más detalle en el próximo capítulo dedicado al desarrollo del proyecto.

Capítulo 8.
Desarrollo del proyecto

En este capítulo se van a explicar todas las partes que componen el proyecto a nivel de software.

8.1 Creación del espacio de trabajo

El sistema "ROS", está configurado para atender solicitudes de compilación y ejecución de los paquetes que se encuentren bajo el directorio "catkin_ws".

En el caso de que se desee crear un paquete nuevo dentro de esta estructura, se usara el comando "**catkin_create_pkg <<nombre_paquete>>**", creándolo en "catkin_ws/src/…".

En nuestro caso, los paquetes con las dos librerías necesarias, "**DepthImage_To_LaserScan**" y "**Laser_Scan_Matcher**", se clonan directamente en el directorio "catkin_ws/src/".

Por último, se compila todo con el comando "**catkin_make**", lo que terminará de crear todo el árbol de directorios que "ROS" necesita para generar los ejecutables del proyecto.

Para más información sobre este tema consultar: [15].

8.2 Ejecución inicial de las librerías

Una vez creado el espacio de trabajo, hay que asegurarse de tener iniciados los servicios que provee el paquete de librerias "**OpenNi**", encargado de publicar las imágenes de profundidad.

La ejecución de este paquete se realiza mediante el comando "**roslaunch**" el cual permite ejecutar archivos de tipo "**.launch**" que son usados para configurar los parámetros de las aplicaciones en el momento de iniciar su ejecución.

Su sintaxis seria:

"**roslaunch openni_launch openni.launch**".

Una vez lanzado "OpenNi" en una consola lo dejaremos en ejecución durante los periodos de desarrollo y pruebas, reiniciándolo de ser necesario.

A continuación pasamos ejecutar la librería “DepthImage_To_LaserScan” haciendo uso del ejemplo que viene incorporado en la descarga, como se muestra en la siguiente imagen con datos precargados.

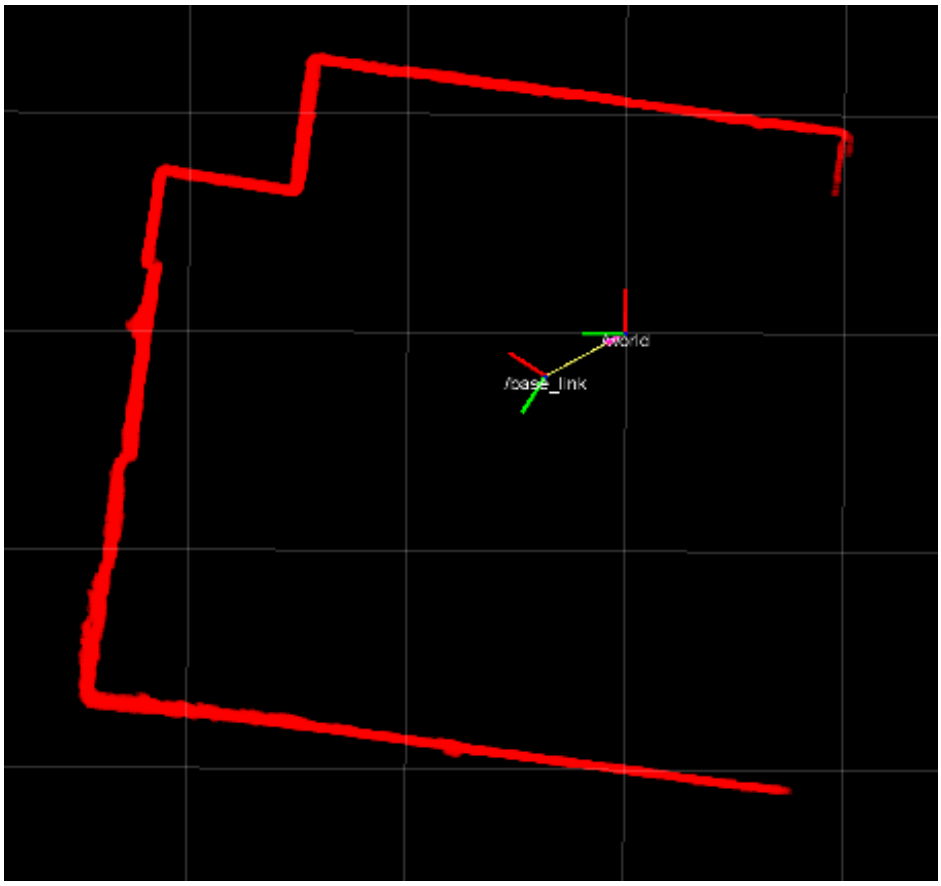


Figura 8.1. Demo de DepthImage to LaserScan con Rviz

8.3 Configuración de arranque del sistema

Dado que el sistema dependerá de la ejecución simultanea de varios programas, crearemos un archivo de tipo “.launch”, que permita configurar los parámetros de inicio de los programas.

Este archivo “.launch” deberá ir en la misma carpeta “src/” del paquete donde se encuentren los archivos “.cpp” a ejecutar.

Dentro del paquete “DepthImage_To_LaserScan”, por ser la primera de las dos librerías a utilizar, y a efectos de poder empezar a trabajar con la obtención de datos, se creará un archivo llamado “prueba.cpp” que será el programa principal y contendrá el núcleo de la funcionalidad del sistema.

El método “main” iniciará los procesos de publicación y subscrición a recepción de mensajes necesarios para que las aplicaciones se comuniquen entre sí.

8.4 Subscripción al tópico Camera_Depth_Image

Para poder obtener la imagen de profundidad hay que subscribirse a las publicaciones de "OpenNi" en el tópico "/camera/depth/image_rect_raw", que publica mensajes de tipo "sensor_msgs::Image".

Estos mensajes además de contener los campos de cabecera, ancho y alto de la imagen y su codificación, también contienen el campo "data" de tipo uint8[], que contiene los 640x480 valores de profundidad de la imagen expresados en milímetros.

La subscripción a este tópico se realiza creando un nuevo manejador de nodos y un objeto de subscripción a recepción de mensajes.

El siguiente ejemplo indica como subscribirse a dicho tópico para recibir las imágenes de profundidad:

```
ros::init(argc, argv, "listener");
ros::NodeHandle n;
ros::Subscriber sub = n.subscribe("camera/depth/image_rect_raw", 30, imgCallback);
```

El número 30 indica la cola de mensajes que se guardaran antes de empezar a descartarlos, e "imgCallback", el nombre del método local que recibirá los mensajes:

```
void imgCallback(const sensor_msgs::Image::Ptr& msg)
{
    …
}
```

8.5 Recepción de mensajes de tipo sensor_msgs::Image

En el momento de la recepción de una imagen de profundidad, el método "imgCallback" recibirá como parámetro un puntero de tipo "sensor_msgs::Image" que contiene los valores de profundidad expresados en milímetros.

Al ser un puntero y no necesitar toda la información que contiene, se hará una copia de esta información en variables locales.

Los datos necesarios se extraerán de esta imagen de profundidad, y serán tres filas horizontales de 640 valores uint8, y tres columnas verticales de 480 valores uint8.

8.6 Corrección del error en el cálculo de distancias

En las pruebas realizadas, haciendo uso de la herramienta de representación gráfica "rviz", se observa que las medidas tomadas por el sensor, situado a cierta distancia, y de forma perpendicular sobre una superficie plana, daban como resultado una línea curva simétrica al eje de visión.

Este hecho pone de manifiesto que el sensor Kinect está calculando las distancias respecto del eje "x" y no respecto de los ejes "xy", lo que provoca este efecto curvo en la representación de lo que debería ser una línea recta.

En la siguiente imagen se ve exactamente cuál es la distancia que está devolviendo el sensor y sirve de partida para modelar el problema de forma matemática.

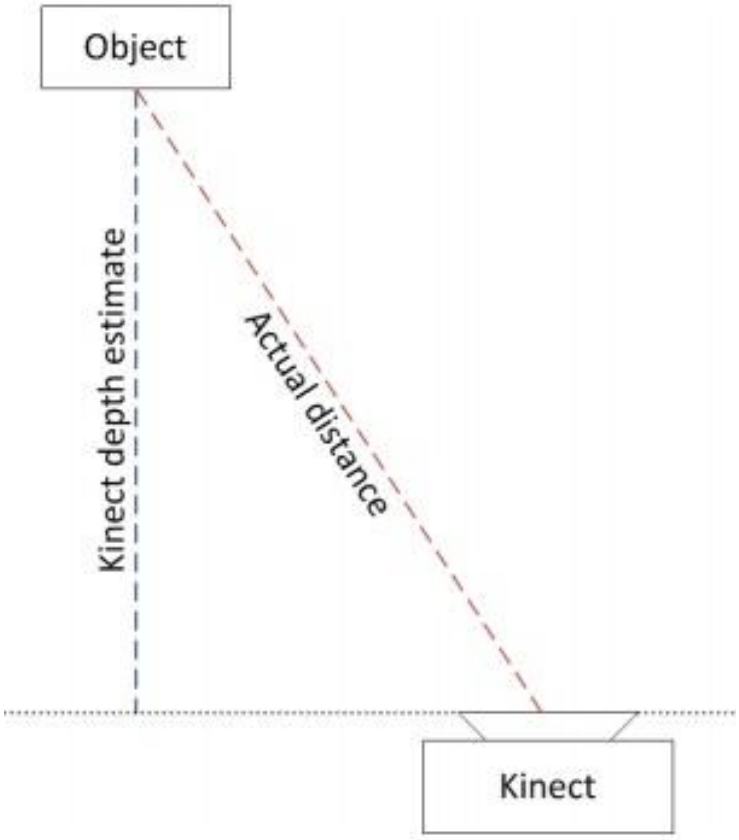


Figura 8.2. Estimación de distancias de Kinect

Como se puede observar, la distancia estimada es la que se obtiene del sensor, siendo la distancia marcada en la imagen como distancia actual la que sería correcta.

Para corregir esto tomaremos el triángulo de la imagen como base y los datos disponibles para calcular la distancia actual correcta.

La siguiente razón trigonométrica será la usada para resolver el problema:

$$\frac{a}{SenA} = \frac{b}{SenB} = \frac{c}{SenC}$$

Como la línea de distancia estimada forma un ángulo de 90º con la base, conocida la medida de esa línea, que es el valor que retorna el sensor, y conocido el ángulo que forma con la línea de distancia actual, que es la que queremos conocer, aplicando la formula se puede despejar y obtener su valor.

El ángulo que forma cada pixel del escaneo con el sensor va en función de la fila y columna que ocupe cada pixel en la matriz de 640x480.

Los cálculos hechos en el paso anterior habrá que repetirlos para corregir la desviación vertical de todos aquellos escaneos que estén por encima o por debajo de la fila central de la matriz de datos.

Del mismo modo, también habrá que hacer lo mismo para los escaneos verticales desplazados de la columna central de la imagen.

Teniendo en cuenta los ángulos de visión de la cámara que son: 57º horizontales y 43º verticales, y sabiendo la resolución de la imagen, es fácil calcular cuánto representa un grado en pixels de desviación desde el centro.

Los resultados son:

- **Desviación horizontal:** 640px / 57º = 11.23px/grado.
- **Desviación vertical:** 480px / 43º = 11.16px/grado.

8.7 Publicación de mensajes sensor_msgs::LaserScan

En el punto quinto de este capítulo hemos mencionado que no toda la información de la imagen de profundidad es necesaria, y que de ella sólo tomaremos tres filas y tres columnas que representan escaneos horizontales y verticales de la imagen.

Llamaremos escaneos a estas filas y columnas debido a que la siguiente librería que usaremos está preparada para recibir mensajes de tipo "LaserScan" que contienen **"una y sólo una fila horizontal de una imagen de profundidad",** con la que realiza el cálculo de la odometría.

En la imagen que aparece en el siguiente apartado, se muestra una representación gráfica de los seis escaneos, tres horizontales que contienen datos

de tres filas de la imagen, y otros tres verticales que contienen datos de tres columnas.

Para lograr poder trabajar con la librería obviando que sólo se le pueda pasar una fila horizontal como escaneo haremos dos cosas:

Primero daremos ID diferentes a cada escaneo "LaserHx" para los horizontales y "LaserVx" para los verticales donde "x" ira de 1 a 3 ambos inclusive.

La distribución quedará de la siguiente forma:

- LaserH1: Escaneo horizontal superior.
- LaserH2: Escaneo horizontal central.
- LaserH3: Escaneo horizontal inferior.
- LaserV1: Escaneo vertical izquierdo.
- LaserV2: Escaneo vertical central.
- LaserV3: Escaneo vertical derecho.

En segundo lugar trataremos los escaneos verticales como si fueran horizontales pero con el parámetro de ángulo de visión establecido a 43º, en vez de a 57º, y el escaneo en vez de contar con 640 valores, contará con 480.

En el archivo "pruebas.launch" habrá que hacer seis "remaps" del nombre del tópico "scan", que es donde la librería espera recibir mensajes, a nuevos nombres de tópicos para poder acceder a los resultados de cada escaneo de forma independiente.

En este caso, los nombres de los tópicos seguirán una nomenclatura análoga a los ID: "scanhx" y "scanvx", para los escaneos horizontales y verticales respectivamente, donde "x" irá de 1 a 3, ambos inclusive, indicando el mismo orden que en los ID.

A continuación se muestra un fragmento de código de esta configuración en el archivo "pruebas.launch".

```
<node pkg="laser_scan_matcher"    type="laser_scan_matcher_node" name="LaserH1">
      <param name="_use_odom" type="bool" value="false" />
      <param name="_use_imu" type="bool" value="false" />
      <remap from="scan" to="scanh1" />
   <remap from="pose2DCov" to="pose2D_H1" />
</node>
```

La ejecución de este nodo en concreto generará un tópico llamado "LaserH1" que se configura para que no use entradas de odometría, ni de una imu.

El primer remap se hace para indicar a la librería que los mensajes que espera recibir en el tópico “scan”, los recibirá en el tópico “scanh1”, que serán los escaneos de la primera fila horizontal.

El segundo remap se explicará más adelante cuando veamos como modificar la información que la librería “Laser_Scan_Matcher” publica para poder obtener una odometría 3D.

Para publicar el mensaje en el sistema se usará un objeto de tipo “sensor_msgs::LaserScan”, teniendo en cuenta que, **los valores que no se encuentren dentro de los parámetros máximo y mínimo, referidos a las distancias admitidas, no deberán incluirse en el array de datos del mensaje, “ranges[]”.**

Las siguientes líneas de código muestran cómo crear, configurar y publicar un mensaje de tipo “sensor_msgs::LaserScan”.

```
ros::Publisher m_ScanH1Pub;
m_ScanH1Pub = n.advertise<LaserScan>("scanh1", 30);
sensor_msgs::LaserScan scanMsgH1;
scanMsgH1.header.frame_id = "LaserH1";
scanMsgH1.header.stamp = ros::Time::now();
scanMsgH1.angle_min = -0.4974;
scanMsgH1.angle_max = 0.4974;
scanMsgH1.angle_increment = 0.994837674 / 640;
scanMsgH1.time_increment = (1 / 30) / 640;
scanMsgH1.range_min = 0.45;
scanMsgH1.range_max = 4.5;
for(int r = 0; r < 640; r++){
      scanMsgH1.ranges[r] = datos[r];
}
m_ScanH1Pub.publish(scanMsgH1);
```

Inicialmente se crea un objeto publicador, “m_ScanH1Pub”, al que se le indica que va a publicar mensajes de tipo “LaserScan” en el tópico “scanh1”.

Después se crea el objeto de tipo “LaserScan”, “scanMsgH1”, a publicar, y se le asignan los valores de ángulos en radianes y distancias en metros.

En el campo “ranges” irán los valores válidos de la fila o columna de la imagen de profundidad.

Finalmente se publica el mensaje.

8.8 Comprobación en entorno gráfico.

Para verificar la correcta aplicación de los cálculos se utilizan diferentes superficies planas para construir un entorno controlado.

Usaremos la herramienta de representación gráfica “Rviz” mediante la ejecución del siguiente comando que lanzará el entorno:

“rosrun rviz rviz”.

Una vez allí habrá que añadir tantos tópicos como escaneos queramos ver, en este caso seis.

A medida que vayamos añadiendo los tópicos “LaserH1,⋯,LaserV3”, irán apareciendo las líneas que representan las superficies detectadas.

Para la visualización del entorno que se muestra en la imagen ha sido necesario aplicar una transformación de 90º, (1,5708 Rad), en el archivo “pruebas.launch”, en los escaneos verticales.

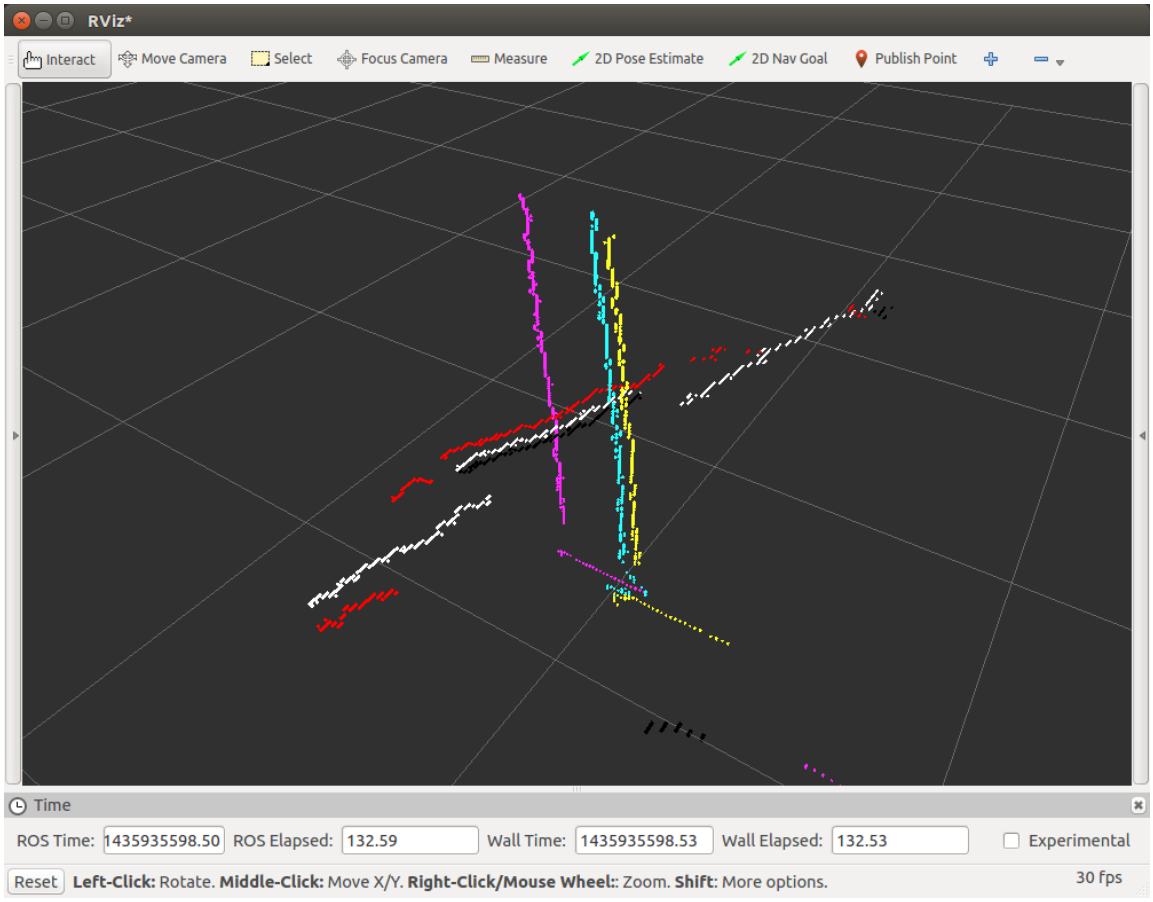


Figura 8.3. Representación gráfica de los escaneos

Una vez comprobados los datos habrá que deshacer las transformaciones en el archivo "pruebas.launch", debido a que la librería parece no procesar de forma correcta escaneos a los que se les ha aplicado una transformación de giro.

El siguiente fragmento de código muestra cómo aplicar las transformaciones en el el archivo "pruebas.launch":

```
    <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_V1" args="0
0 0 1,5708 0 0 1 base_link LaserV1 33" />
    <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_V2" args="0
0 0 1,5708 0 0 1 base_link LaserV2 33" />
    <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_V3" args="0
0 0 1,5708 0 0 1 base_link LaserV3 33" />
```

8.9 Adaptación 3D en Laser_Scan_Matcher

"Laser_Scan_Matcher" es una librería pensada para dar soluciones de odometría 2D, siendo el objetivo de este trabajo obtener una odometría 3D.

La librería tratará los escaneos verticales como si fueran horizontales, por lo que igualmente retorna las componentes "x", "y" y "theta", para dichos escaneos.

Si nos imaginamos que giramos la cámara 90º, lo que antes era el movimiento "y", (izquierda-derecha), se convierte en el movimento "z", (arriba-abajo).

Del mismo modo, con la cámara girada 90º, el que antes era el giro, "yaw", de (guiñada o deriva), se convierte en "pitch", o ángulo de cabeceo.

Se accederá al archivo principal de la librería "laser_scan_matcher.cpp" para modificarlo y crear un nuevo tipo de mensaje de odometría 2D que contenga las covarianzas de las predicciones, como se verá en el fragmento de código de ejemplo, al final de este apartado.

También será necesario indicar en el código que realice los cálculos usando una matriz de covarianza.

Esta matriz de covarianza se usará posteriormente para ver cuál de las odometrías ofrece mayor precisión, y lo haremos añadiendo la siguiente línea de código en el método "processScan" de la librería:

```
"input_.do_compute_covariance = true;"
```

Como se necesita tener acceso a esos valores de covarianza en el programa principal, justo antes de la publicación del mensaje por parte de la librería, modificaremos el código para que publique otro tipo de mensaje llamado "Pose2DWithCovariance" con el siguiente contenido:

geometry_msgs/Pose2D pose

float64[9] covariance

Este mensaje de tipo “Pose2DWithCovariance” se creará en un archivo de texto llamado “Pose2DWithCovariance.msg” que deberá estar ubicado en la carpeta:

“src/scan_tools/laser_scan_matcher/msg”.

La modificación se hará en el mismo método “processScan”, según el siguiente fragmento de código de ejemplo:

```
m_publisher = nh_.advertise<Pose2DWithCovariance>("pose2DCov", 5);
Pose2DWithCovariance::Ptr pose;
pose = boost::make_shared<Pose2DWithCovariance>();
pose->pose.x = pose_msg->x;
pose->pose.y = pose_msg->y;
pose->pose.theta = pose_msg->theta;
pose->covariance[0]=gsl_matrix_get(output_.cov_x_m,0,0);
…
pose->covariance[8]=gsl_matrix_get(output_.cov_x_m,2,2);
m_publisher.publish(pose);
```

En este ejemplo se crea un publicador “m_publisher”, que publicará mensajes de tipo “Pose2DWithCovariance” en el tópico “pose2DCov”.

Por este motivo veíamos el remap del archivo “.launch”, en el punto anterior que no se explicó:

```
<remap from="pose2DCov" to="pose2D_H1" />
```

Se indica un nombre de tópico específico para cada odometría, en este caso, “pose2D_H1”.

Esto indica que en ese tópico se recibirán los mensajes de odometría procedentes de los escaneos horizontales superiores de la imagen de profundidad.

De esta forma, desde el programa principal podremos subscribirnos a dichos tópicos y saber, en todo momento, qué odometría se recibe de las seis que se publican.

8.10 Recepción de mensajes Pose2DWithCovariance

En el método “main” del programa principal “prueba.cpp”, debemos crear un objeto de subscripción de mensajes al que pasaremos los siguientesparámetros:

- El nombre del tópico al que se va a subscribir,
- La cola de mensajes que almacena antes de empezar a descartarlos.
- El nombre del método local encargado de recibir y procesar cada mensaje.

El siguiente ejemplo muestra cómo subscribirse a la publicación de mensajes hecha en el punto anterior:

```
ros::Subscriber poseSubH1 = n.subscribe("pose2D_H1", 30, pcbH1);
```

La cabecera del método “poseCallbackH1 sería la siguiente:

```
void pcbH1(const Pose2DWithCovariance::ConstPtr& msg){…}
```

8.11 Cálculo de la mejor odometría

Debido a que los datos de odometría son acumulativos, para poder elegir, en cada iteración, los mejores valores posibles entre las diferentes odometrías, es necesario añadir cinco variables locales donde se irán acumulando los valores de cada componente de la odometría 3D final.

Los valores de las componentes de la odometría final se verán modificados por valores de mayor precisión de cada componente según la siguiente fórmula:

$$\boldsymbol{valorLocal} \mathrel{+}= (\boldsymbol{valorNuevo} - \boldsymbol{valorLocal})$$

Esta fórmula se aplicará a las componentes: (“x”, “y”, “z”, “yaw” (ángulo de guiñada), y “pitch” (ángulo de cabeceo)).

Se crearan seis vectores, a modo de colas, uno para cada una de las seis odometrías que se publican.

En cada recepción de un mensaje de odometría 2D, se coloca éste al final de su cola correspondiente, y se comprueba que todas las colas tengan al menos un mensaje para comenzar el proceso.

Las seis odometrías situadas en la posición inicial de cada cola son extraídas de dichas colas y copiadas en variables locales.

Tomaremos los valores de la posición [0][0] de las matrices de covarianza de las tres odometrías 2D horizontales para obtener aquella componente “x” con covarianza más próxima a cero, y cuyo valor pasará a modificar la componente “x” de la odometría 3D final.

El valor de la posición [1][1] de las matrices de covarianza de las odometrías 2D horizontales corresponderá a la componente “y”, y el valor de la posición [2][2] al ángulo Theta que representa el ángulo “yaw” en las odometrías 2D horizontales.

Para las odometrías fruto de escaneos verticales, el valor de la posición [0][0] de las matrices de covarianza corresponderá a la “x” que ya la tenemos por lo que este valor no lo usaremos.

La posición [1][1] corresponderá a la covarianza de la “y”, pero en las odometrías que se calcularon con escaneos verticales lo que antes representaba el desplazamiento (izquierda - derecha), ahora es el desplazamiento (arriba - abajo), ósea, la componente “z”.

Del mismo modo tomaremos el valor de la posición [2][2], covarianza para “Theta”, que se convierte en el ángulo “pitch”, en las odometrías que se calcularon con escaneos verticales.

8.12 Creación del mensaje Pose3D

Una vez actualizados los datos locales, la odometría 3D está lista para ser publicada en un nuevo tópico llamado “kinect_Odometry” con nuevos mensajes de tipo “Pose3D”.

De igual forma que se creó el mensaje “Pose2DWithCovariance”, crearemos un nuevo mensaje de tipo “Pose3D” con el siguiente contenido:

```
float64 x
float64 y
float64 z
float64 yaw
float64 pitch
```

Este nuevo mensaje se creará en un archivo de texto llamado “Pose3D.msg” que deberá estar ubicado en la carpeta:

“src/depthimage_to_laserscan-indigo-devel/msg”.

La creación y publicación de este mensaje de odometría final en 3D se hará según el siguiente ejemplo, donde las componentes de desplazamiento se mostrarán en centímetros, y los ángulos de giro en grados:

```
float m_DisplaceX = 0.0, m_DisplaceY = 0.0, m_DisplaceZ = 0.0;
float m_YawAngle = 0.0, m_PitchAngle = 0.0;
ros::Publisher m_KnOdom;
m_KnOdom = n.advertise<Pose3D>("kinect_Odometry", 30);
Pose3D odom;
…
odom.x = m_DisplaceX * 100.0;
odom.y = m_DisplaceY * 100.0;
odom.z = m_DisplaceZ * 100.0;
odom.yaw = (m_YawAngle*180)/M_PI;
odom.pitch = (m_PitchAngle*180)/M_PI;
m_KnOdom.publish(odom);
```

Capítulo 9.
Conclusiones y líneas futuras

En este trabajo final de grado se ha diseñado e implementado un sistema de odometría visual basado en las características de detección de profundidad del sensor Kinect.

Estas características han sido usadas para obtener seis escaneos por cada imagen de profundidad obtenida, y generar sus respectivas odometrías, lo cual ha sobrepasado la capacidad de procesamiento de la placa "Odroid U3", para ofrecer una odometría 3D en tiempo real.

La poca precisión, y el reducido alcance de visión del sensor kinect, han hecho necesario usar múltiples escaneos y aplicar estadística, para aumentar la precisión de los cálculos, lo que ha aumentado considerablemente el tiempo de respuesta del sistema.

El principal inconveniente encontrado ha sido la poca capacidad de procesamiento por parte de la placa Odroid U3, ya que, aún con el error y oscilaciones en las mediciones de la cámara, y pese a su escaso alcance de visión, sería posible obtener una buena odometría empleando otro tipo de micro placa con un procesador como los de los ordenadores de escritorio.

Esto permitiría usar aun más escaneos, y mediante estadística, y un sistema de selección, aumentar considerablemente la precisión actual y en tiempo real.

Para continuar con este trabajo, se recomienda volver a implementar el sistema de escaneos adyacentes, que se comenta en el punto 6.5, y añadir al mensaje "Pose2DWithCovariance", el valor "_nValid", que representa el número de puntos que la librería "Laser_Scan_Matcher" pudo reconocer para calcular la odometría, para los casos en los que las covarianzas sean idénticas.

If above problems of processing power for devices of reduced size are resolved, it is concluded that the project is viable based on the good results obtained in the trials, especially in movements (forward - backward) and rotation, which is where most precision has the system.

Capítulo 10.
Summary and Conclusions

In this final degree work it is designed and implemented a visual odometry system, based on the features of depth detection of the kinect sensor.

These features have been used to get six scans, of each depth-image obtained, and generate their poses, which has exceeded the processing capacity of the "Odroid U3", to get 3D pose, in real time.

The inaccuracy and the limited vision scope, of the kinect sensor, have made it necessary to use multiple scans, and apply statistics to increase the accuracy of calculations, which has greatly increased the response time of the system.

Main problem encountered has been the low processing capacity by ODROID U3 because, even with the error and variations in the measurements of the camera, and despite its limited scope of vision, it would be possible to get a good pose using another type of micro motherboards with processors like the desktop computers.

This would allow use even more scans, and using statistics, and a selection system greatly increase the current accuracy, and in real time.

To continue this work, you are recommended to implement the system of adjacent scans, discussed in section 6.5, and add in the ″Pose2DWithCovariance″ message, the value ″_nValid″, representing the number of points that the library ″Laser_Scan_Matcher″ recognized, to calculate the odometry, for cases where the covariances be same.

Capítulo 11. Presupuesto

En este capítulo se exponen los presupuestos estimativos del coste de un desarrollo como el expuesto en este trabajo.

Se detallan por separado el presupuesto de componentes del de desarrollo. Los precios aplicados a la parte de desarrollo son orientativos.

11.1 Presupuesto de componentes

Descripción	Uds.	Precio	Total
Cámara Kinect 1.0	1	55,00	55,00
Placa Odroid U3	1	65,00	65,00
Módulo eMMC 16GB para Odroid U3	1	35,00	35,00

Tabla 11.1. Presupuesto de componentes

11.2 Presupuesto de desarrollo

Descripción	Uds.	Precio	Total
Estudio, análisis y planificación	60	55,00	3.300.00
Configuración de los sistemas	40	35,00	1.400.00
Programación y pruebas	160	45,00	7.200.00
Elaboración de la documentación	20	35,00	700,00
Gastos varios	1	120,00	120,00

Tabla 11.2. Presupuesto de desarrollo

11.3 Presupuesto general del proyecto

Descripción	Total
Componentes del desarrollo	155,00
Desarrollo del proyecto	12.720.00
Total	**12.875.00**

Tabla 11.3. Presupuesto general del proyecto

Apéndice A. Algoritmos

A.1. Algoritmo pruebas.cpp

```
   /*****************************************************************************
   *
   * Fichero .h pruebas.h
   *
   *****************************************************************************
   *
   * AUTORES Oliver Martínez Novo
   *
   * FECHA 04/02/2015
   *
   * DESCRIPCION El conjunto de métodos del fichero se encargan de recibir la
información de la cámara, generar los escaneos que posteriormente publicara, recibirá
las odometrías de dichos escaneos, tomará la mejor y la publicará en un nuevo tópico.
   *
   *
   *****************************************************************************/
/*
ALPHA representa la relación entre el ángulo de visión
vertical de la cámara y la altura de la imagen de profundidad.
Es una constante utilizada en el cálculo de la fila que hay que
examinar en función del ángulo de cabeceo.
*/
#define ALPHA 0.001563524122
#include <depthimage_to_laserscan/DepthImageToLaserScanROS.h>
#include "geometry_msgs/Pose2D.h"
#include <laser_scan_matcher/Pose2DWithCovariance.h>
#include <depthimage_to_laserscan/Pose3D.h>
#include <ros/console.h>
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <vector>
#include <map>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
```

```
using namespace depthimage_to_laserscan;
using namespace laser_scan_matcher;
vector<Pose2DWithCovariance> m_PosesH1;
vector<Pose2DWithCovariance> m_PosesH2;
vector<Pose2DWithCovariance> m_PosesH3;
vector<Pose2DWithCovariance> m_PosesV1;
vector<Pose2DWithCovariance> m_PosesV2;
vector<Pose2DWithCovariance> m_PosesV3;
ros::Publisher m_ScanH1Pub;
ros::Publisher m_ScanH2Pub;
ros::Publisher m_ScanH3Pub;
ros::Publisher m_ScanV1Pub;
ros::Publisher m_ScanV2Pub;
ros::Publisher m_ScanV3Pub;
ros::Publisher m_KinectOdometry;
ros::Time m_Time;
unsigned short int* m_NewRow;
unsigned short int m_Read1[640];
int m_NumRow = 0;
float m_DisplaceX = 0.0;
float m_DisplaceY = 0.0;
float m_DisplaceZ = 0.0;
float m_YawAngle = 0.0;
float m_PitchAngle = 0.0;

/*    Calcula la fila correspondiente a analizar
      en función del ángulo que tenga el dispositivo. */
void getFila(const sensor_msgs::Image::Ptr& msg){
      double inputAngleDeg = 0.0;
      m_NumRow = 239 - nearbyint(((inputAngleDeg*M_PI)/180)/ALPHA);
      m_NewRow = &((unsigned short int*)msg->data.data())[640*m_NumRow];
}

// Los valores 0 no se incluyen en el array
void publishScan(const sensor_msgs::Image::Ptr& msg)
{
      m_Time = ros::Time::now();
      sensor_msgs::LaserScan scanMsgH1;
      sensor_msgs::LaserScan scanMsgH2;
      sensor_msgs::LaserScan scanMsgH3;
      sensor_msgs::LaserScan scanMsgV1;
      sensor_msgs::LaserScan scanMsgV2;
      sensor_msgs::LaserScan scanMsgV3;
      unsigned int num_readings = 640;
      double laser_frequency = 30;
      int value = INT_MIN;
      float temp = 0.0;
      float angulo = 0.0;
      float distancia = 0.0;
      scanMsgH1.header.frame_id = "LaserH1";
      scanMsgH1.header.stamp = m_Time;
      scanMsgH1.angle_min = -0.4974;
      scanMsgH1.angle_max = 0.4974;
      scanMsgH1.angle_increment = 0.994837674 / num_readings;
      scanMsgH1.time_increment = (1 / laser_frequency) / (num_readings);
      scanMsgH1.range_min = 0.45;
      scanMsgH1.range_max = 4.5;
      scanMsgH1.ranges.resize(num_readings);
      scanMsgH1.intensities.resize(num_readings);
      m_NewRow = &((unsigned short int*)msg->data.data())[(640*(m_NumRow - 56))];
      for(int r = 0; r < 640; r++)
```

```
{
	if((float)m_NewRow[r] > 0.0)
	{
		angulo = ((fabs(21.5 - (0.089583333 * 56)))*M_PI)/180.0;
		distancia = m_NewRow[r] / sin(M_PI - (angulo + (M_PI/2)));
		temp = distancia;
		angulo = ((fabs(28.5 - (0.0890625 * r)))*M_PI)/180.0;
		distancia = temp / sin(M_PI - (angulo + (M_PI/2)));
		scanMsgH1.ranges[r] = (float)distancia/1000.0;
	}
}
m_ScanH1Pub.publish(scanMsgH1);
scanMsgH1.ranges.clear();
scanMsgH2.header.frame_id = "LaserH2";
scanMsgH2.header.stamp = m_Time;
scanMsgH2.angle_min = -0.4974;
scanMsgH2.angle_max = 0.4974;
scanMsgH2.angle_increment = 0.994837674 / num_readings;
scanMsgH2.time_increment = (1 / laser_frequency) / (num_readings);
scanMsgH2.range_min = 0.45;
scanMsgH2.range_max = 4.5;
scanMsgH2.ranges.resize(num_readings);
scanMsgH2.intensities.resize(num_readings);
m_NewRow = &((unsigned short int*)msg->data.data())[(640*m_NumRow)];
for(int r = 0; r < 640; r++)
{
	if((float)m_NewRow[r] > 0.0)
	{
		angulo = ((fabs(28.5 - (0.0890625 * r)))*M_PI)/180.0;
		distancia = m_NewRow[r] / sin(M_PI - (angulo + (M_PI/2)));
		scanMsgH2.ranges[r] = (float)distancia/1000.0;
	}
}
m_ScanH2Pub.publish(scanMsgH2);
scanMsgH2.ranges.clear();
scanMsgH3.header.frame_id = "LaserH3";
scanMsgH3.header.stamp = m_Time;
scanMsgH3.angle_min = -0.4974;
scanMsgH3.angle_max = 0.4974;
scanMsgH3.angle_increment = 0.994837674 / num_readings;
scanMsgH3.time_increment = (1 / laser_frequency) / (num_readings);
scanMsgH3.range_min = 0.45;
scanMsgH3.range_max = 4.5;
scanMsgH3.ranges.resize(num_readings);
scanMsgH3.intensities.resize(num_readings);
m_NewRow = &((unsigned short int*)msg->data.data())[(640*(m_NumRow + 56))];

for(int r = 0; r < 640; r++)
{
	if((float)m_NewRow[r] > 0.0)
	{
		angulo = ((fabs(21.5 - (0.089583333 * 56)))*M_PI)/180.0;
		distancia = m_NewRow[r] / sin(M_PI - (angulo + (M_PI/2)));
		temp = distancia;
		angulo = ((fabs(28.5 - (0.0890625 * r)))*M_PI)/180.0;
		distancia = temp / sin(M_PI - (angulo + (M_PI/2)));
		scanMsgH3.ranges[r] = (float)distancia/1000.0;
	}
}
m_ScanH3Pub.publish(scanMsgH3);
scanMsgH3.ranges.clear();
```

```
num_readings = 480;
scanMsgV1.header.frame_id = "LaserV1";
scanMsgV1.header.stamp = m_Time;
scanMsgV1.angle_min = -0.3752;
scanMsgV1.angle_max = 0.3752;
scanMsgV1.angle_increment = 0.750491578 / num_readings;
scanMsgV1.time_increment = (1 / laser_frequency) / (num_readings);
scanMsgV1.range_min = 0.45;
scanMsgV1.range_max = 4.5;
scanMsgV1.ranges.resize(num_readings);
scanMsgV1.intensities.resize(num_readings);
m_NewRow = &((unsigned short int*)msg->data.data())[319];
for(int r = 0; r < 480; r++)
{
	if((float)m_NewRow[r*640] > 0.0)
	{
		angulo = ((fabs(21.5 - (0.089583333 * r)))*M_PI)/180.0;
		distancia = m_NewRow[r*640] / sin(M_PI - (angulo + (M_PI/2)));
		temp = distancia;
		angulo = ((fabs(28.5 - (0.0890625 * 56)))*M_PI)/180.0;
		distancia = temp / sin(M_PI - (angulo + (M_PI/2)));
		scanMsgV1.ranges[r] = (float)distancia/1000.0;
	}
}
m_ScanV1Pub.publish(scanMsgV1);
scanMsgV1.ranges.clear();
scanMsgV2.header.frame_id = "LaserV2";
scanMsgV2.header.stamp = m_Time;
scanMsgV2.angle_min = -0.3752;
scanMsgV2.angle_max = 0.3752;
scanMsgV2.angle_increment = 0.750491578 / num_readings;
scanMsgV2.time_increment = (1 / laser_frequency) / (num_readings);
scanMsgV2.range_min = 0.45;
scanMsgV2.range_max = 4.5;
scanMsgV2.ranges.resize(num_readings);
scanMsgV2.intensities.resize(num_readings);
m_NewRow = &((unsigned short int*)msg->data.data())[263];
for(int r = 0; r < 480; r++)
{
	if((float)m_NewRow[r*640] > 0.0)
	{
		angulo = ((fabs(21.5 - (0.089583333 * r)))*M_PI)/180.0;
		distancia = m_NewRow[r*640] / sin(M_PI - (angulo + (M_PI/2)));
		scanMsgV2.ranges[r] = (float)distancia/1000.0;
	}
}
m_ScanV2Pub.publish(scanMsgV2);
scanMsgV2.ranges.clear();
scanMsgV3.header.frame_id = "LaserV3";
scanMsgV3.header.stamp = m_Time;
scanMsgV3.angle_min = -0.3752;
scanMsgV3.angle_max = 0.3752;
scanMsgV3.angle_increment = 0.750491578 / num_readings;
scanMsgV3.time_increment = (1 / laser_frequency) / (num_readings);
scanMsgV3.range_min = 0.45;
scanMsgV3.range_max = 4.5;
scanMsgV3.ranges.resize(num_readings);
scanMsgV3.intensities.resize(num_readings);
m_NewRow = &((unsigned short int*)msg->data.data())[375];
for(int r = 0; r < 480; r++)
{
```

```
			if((float)m_NewRow[r*640] > 0.0)
			{
				angulo = ((fabs(21.5 - (0.089583333 * r)))*M_PI)/180.0;
				distancia = m_NewRow[r*640] / sin(M_PI - (angulo + (M_PI/2)));
				temp = distancia;
				angulo = ((fabs(28.5 - (0.0890625 * 56)))*M_PI)/180.0;
				distancia = temp / sin(M_PI - (angulo + (M_PI/2)));
				scanMsgV3.ranges[r] = (float)distancia/1000.0;
			}
		}
		m_ScanV3Pub.publish(scanMsgV3);
		scanMsgV3.ranges.clear();
}

void getDepth(const sensor_msgs::Image::Ptr& msg)
{
		getFila(msg);
		publishScan(msg);
}

Pose2DWithCovariance  computePose2DWithCovariance(const  Pose2DWithCovariance::ConstPtr&
msg)
{
		Pose2DWithCovariance aux;
		aux.pose.x = msg->pose.x;
		aux.pose.y = msg->pose.y;
		aux.pose.theta = msg->pose.theta;
		aux.covariance[0] = msg->covariance[0];
		aux.covariance[1] = msg->covariance[1];
		aux.covariance[2] = msg->covariance[2];
		aux.covariance[3] = msg->covariance[3];
		aux.covariance[4] = msg->covariance[4];
		aux.covariance[5] = msg->covariance[5];
		aux.covariance[6] = msg->covariance[6];
		aux.covariance[7] = msg->covariance[7];
		aux.covariance[8] = msg->covariance[8];
		return aux;
}

void processOdom()
{
		Pose3D odom;
		float minValue;
		int numPose;
		if((m_PosesH1.size() > 0) && (m_PosesH2.size() > 0) && (m_PosesH3.size() > 0) &&
(m_PosesV1.size() > 0) && (m_PosesV2.size() > 0) && (m_PosesV3.size() > 0))
		{
			Pose2DWithCovariance h1 = m_PosesH1[0];
			m_PosesH1.erase(m_PosesH1.begin());
			Pose2DWithCovariance h2 = m_PosesH2[0];
			m_PosesH2.erase(m_PosesH2.begin());
			Pose2DWithCovariance h3 = m_PosesH3[0];
			m_PosesH3.erase(m_PosesH3.begin());
			Pose2DWithCovariance v1 = m_PosesV1[0];
			m_PosesV1.erase(m_PosesV1.begin());
			Pose2DWithCovariance v2 = m_PosesV2[0];
			m_PosesV2.erase(m_PosesV2.begin());
			Pose2DWithCovariance v3 = m_PosesV3[0];
			m_PosesV3.erase(m_PosesV3.begin());

			// Cálculo del desplazamiento escaneos horizontales
```

```
for(int r = 0; r < 3; r++)
{
	minValue = 10000.0;
	numPose = 0;
	if(fabs(h1.covariance[r*4]) < minValue)
	{
		minValue = fabs(h1.covariance[r*4]);
		numPose = 1;
	}
	if(fabs(h2.covariance[r*4]) < minValue)
	{
		minValue = fabs(h2.covariance[r*4]);
		numPose = 2;
	}
	if(fabs(h3.covariance[r*4]) < minValue)
	{
		minValue = fabs(h3.covariance[r*4]);
		numPose = 3;
	}

	if(numPose == 1)
	{
		if(r == 0)
		{
			m_DisplaceX += (h1.pose.x - m_DisplaceX);
		}
		else if(r == 1)
		{
			m_DisplaceY += (h1.pose.y - m_DisplaceY);
		}
		else if(r == 2)
		{
			m_YawAngle += (h1.pose.theta - m_YawAngle);
		}
	}
	else if(numPose == 2)
	{
		if(r == 0)
		{
			m_DisplaceX += (h2.pose.x - m_DisplaceX);
		}
		else if(r == 1)
		{
			m_DisplaceY += (h2.pose.y - m_DisplaceY);
		}
		else if(r == 2)
		{
			m_YawAngle += (h2.pose.theta - m_YawAngle);
		}
	}
	else if(numPose == 3)
	{
		if(r == 0)
		{
			m_DisplaceX += (h3.pose.x - m_DisplaceX);
		}
		else if(r == 1)
		{
			m_DisplaceY += (h3.pose.y - m_DisplaceY);
		}
		else if(r == 2)
```

```
			{
				m_YawAngle += (h3.pose.theta - m_YawAngle);
			}
		}
	}

	// Cálculo del desplazamiento escaneos verticales
	for(int r = 0; r < 3; r++)
	{
		minValue = 10000.0;
		numPose = 0;
		if(fabs(v1.covariance[r*4]) < minValue)
		{
			minValue = fabs(v1.covariance[r*4]);
			numPose = 1;
		}
		if(fabs(v2.covariance[r*4]) < minValue)
		{
			minValue = fabs(v2.covariance[r*4]);
			numPose = 2;
		}
		if(fabs(v3.covariance[r*4]) < minValue)
		{
			minValue = fabs(v3.covariance[r*4]);
			numPose = 3;
		}
		if(numPose == 1)
		{
			if(r == 1)
			{
				m_DisplaceZ += (v1.pose.y - m_DisplaceZ);
			}
			else if(r == 2)
			{
				m_PitchAngle += (v1.pose.theta - m_PitchAngle);
			}
		}
		else if(numPose == 2)
		{
			if(r == 1)
			{
				m_DisplaceZ += (v2.pose.y - m_DisplaceZ);
			}
			else if(r == 2)
			{
				m_PitchAngle += (v2.pose.theta - m_PitchAngle);
			}
		}
		else if(numPose == 3)
		{
			if(r == 1)
			{
				m_DisplaceZ += (v3.pose.y - m_DisplaceZ);
			}
			else if(r == 2)
			{
				m_PitchAngle += (v3.pose.theta - m_PitchAngle);
			}
		}
	}
	odom.x = (int)(m_DisplaceX * 100.0);
```

```
            odom.y = (int)(m_DisplaceY * 100.0);
            odom.z = (int)(m_DisplaceZ * 100.0);
            odom.yaw = (int)((m_YawAngle*180)/M_PI);
            odom.pitch = (int)((m_PitchAngle*180)/M_PI);
            m_KinectOdometry.publish(odom);
      }
}

void chatterCallback(const sensor_msgs::Image::Ptr& msg)
{
      getDepth(msg);
}

void poseCallbackH1(const Pose2DWithCovariance::ConstPtr& msg)
{
      m_PosesH1.push_back(computePose2DWithCovariance(msg));
      processOdom();
}

void poseCallbackH2(const Pose2DWithCovariance::ConstPtr& msg)
{
      m_PosesH2.push_back(computePose2DWithCovariance(msg));
      processOdom();
}

void poseCallbackH3(const Pose2DWithCovariance::ConstPtr& msg)
{
      m_PosesH3.push_back(computePose2DWithCovariance(msg));
      processOdom();
}

void poseCallbackV1(const Pose2DWithCovariance::ConstPtr& msg)
{
      m_PosesV1.push_back(computePose2DWithCovariance(msg));
      processOdom();
}

void poseCallbackV2(const Pose2DWithCovariance::ConstPtr& msg)
{
      m_PosesV2.push_back(computePose2DWithCovariance(msg));
      processOdom();
}

void poseCallbackV3(const Pose2DWithCovariance::ConstPtr& msg)
{
      m_PosesV3.push_back(computePose2DWithCovariance(msg));
      processOdom();
}

int main(int argc, char **argv)
{
      ros::init(argc, argv, "listener");
      ros::NodeHandle n;
      ros::Subscriber    sub    =    n.subscribe("camera/depth/image_rect_raw",    1000,
chatterCallback);
      m_ScanH1Pub = n.advertise<sensor_msgs::LaserScan>("scanh1", 30);
      m_ScanH2Pub = n.advertise<sensor_msgs::LaserScan>("scanh2", 30);
      m_ScanH3Pub = n.advertise<sensor_msgs::LaserScan>("scanh3", 30);
      m_ScanV1Pub = n.advertise<sensor_msgs::LaserScan>("scanv1", 30);
      m_ScanV2Pub = n.advertise<sensor_msgs::LaserScan>("scanv2", 30);
      m_ScanV3Pub = n.advertise<sensor_msgs::LaserScan>("scanv3", 30);
```

```
    ros::Subscriber poseSubH1 = n.subscribe("pose2D_H1", 30, poseCallbackH1);
    ros::Subscriber poseSubH2 = n.subscribe("pose2D_H2", 30, poseCallbackH2);
    ros::Subscriber poseSubH3 = n.subscribe("pose2D_H3", 30, poseCallbackH3);
    ros::Subscriber poseSubV1 = n.subscribe("pose2D_V1", 30, poseCallbackV1);
    ros::Subscriber poseSubV2 = n.subscribe("pose2D_V2", 30, poseCallbackV2);
    ros::Subscriber poseSubV3 = n.subscribe("pose2D_V3", 30, poseCallbackV3);
    m_KinectOdometry = n.advertise<Pose3D>("kinect_Odometry", 30);
    ros::spin();
    return 0;
}
```

A.2. Algoritmo laser_scan_matcher.cpp

```
/*********************************************************************************
*
* Fichero .h laser_scan_matcher.h
*
*********************************************************************************
*
* AUTORES Copyright (c) 2011, Ivan Dryanovski, William Morris
* All rights reserved.
*
* FECHA 2011
*
*  DESCRIPCION Librería encargada de publicar la odometría 2D, así como las
modificaciones necesarias para la obtención de odometrías 3D mediante publicación de
nuevos tipos de mensajes
*
*
*********************************************************************************/
/*
 * Copyright (c) 2011, Ivan Dryanovski, William Morris
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 *     * Redistributions of source code must retain the above copyright
 *       notice, this list of conditions and the following disclaimer.
 *     * Redistributions in binary form must reproduce the above copyright
 *       notice, this list of conditions and the following disclaimer in the
 *       documentation and/or other materials provided with the distribution.
 *     * Neither the name of the CCNY Robotics Lab nor the names of its
 *       contributors may be used to endorse or promote products derived from
 *       this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
```

```
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/*  This package uses Canonical Scan Matcher [1], written by
 *  Andrea Censi
 *
 *  [1] A. Censi, "An ICP variant using a point-to-line metric"
 *  Proceedings of the IEEE International Conference
 *  on Robotics and Automation (ICRA), 2008
 */

#include <laser_scan_matcher/laser_scan_matcher.h>
#include <pcl_conversions/pcl_conversions.h>

namespace scan_tools
{

LaserScanMatcher::LaserScanMatcher(ros::NodeHandle nh, ros::NodeHandle nh_private):
  nh_(nh),
  nh_private_(nh_private),
  initialized_(false),
  received_imu_(false),
  received_odom_(false),
  received_vel_(false)
{
  ROS_INFO("Starting LaserScanMatcher");

  // **** init parameters

  initParams();

  // **** state variables

  f2b_.setIdentity();
  f2b_kf_.setIdentity();
  input_.laser[0] = 0.0;
  input_.laser[1] = 0.0;
  input_.laser[2] = 0.0;

  // **** publishers

  if (publish_pose_)
  {
      //pose_publisher_  = nh_.advertise<geometry_msgs::Pose2D>("pose2D", 5);
      pose_covariance_publisher_                                          =
nh_.advertise<laser_scan_matcher::Pose2DWithCovariance>("pose2DCov", 5);
  }

  if (publish_pose_stamped_)
  {
    pose_stamped_publisher_                                               =
nh_.advertise<geometry_msgs::PoseStamped>("pose_stamped", 5);
  }

  // *** subscribers
```

```
  if (use_cloud_input_)
  {
    cloud_subscriber_ = nh_.subscribe(
      "cloud", 1, &LaserScanMatcher::cloudCallback, this);
  }
  else
  {
    scan_subscriber_ = nh_.subscribe(
      "scan", 1, &LaserScanMatcher::scanCallback, this);
  }

  if (use_imu_)
  {
    imu_subscriber_ = nh_.subscribe(
      "imu/data", 1, &LaserScanMatcher::imuCallback, this);
  }
  if (use_odom_)
  {
    odom_subscriber_ = nh_.subscribe(
      "odom", 1, &LaserScanMatcher::odomCallback, this);
  }
  if (use_vel_)
  {
    vel_subscriber_ = nh_.subscribe(
      "vel", 1, &LaserScanMatcher::velCallback, this);
  }
}

LaserScanMatcher::~LaserScanMatcher()
{
  ROS_INFO("Destroying LaserScanMatcher");
}

void LaserScanMatcher::initParams()
{
  if (!nh_private_.getParam ("base_frame", base_frame_))
    base_frame_ = "base_link";
  if (!nh_private_.getParam ("fixed_frame", fixed_frame_))
    fixed_frame_ = "world";

  // **** input type - laser scan, or point clouds?
  // if false, will subscrive to LaserScan msgs on /scan.
  // if true, will subscrive to PointCloud2 msgs on /cloud

  if (!nh_private_.getParam ("use_cloud_input", use_cloud_input_))
    use_cloud_input_= false;

  if (use_cloud_input_)
  {
    if (!nh_private_.getParam ("cloud_range_min", cloud_range_min_))
      cloud_range_min_ = 0.1;
    if (!nh_private_.getParam ("cloud_range_max", cloud_range_max_))
      cloud_range_max_ = 50.0;
    if (!nh_private_.getParam ("cloud_res", cloud_res_))
      cloud_res_ = 0.05;

    input_.min_reading = cloud_range_min_;
    input_.max_reading = cloud_range_max_;
  }

  // **** keyframe params: when to generate the keyframe scan
```

```
    // if either is set to 0, reduces to frame-to-frame matching

    if (!nh_private_.getParam ("kf_dist_linear", kf_dist_linear_))
      kf_dist_linear_ = 0.10;
    if (!nh_private_.getParam ("kf_dist_angular", kf_dist_angular_))
      kf_dist_angular_ = 10.0 * (M_PI / 180.0);

    kf_dist_linear_sq_ = kf_dist_linear_ * kf_dist_linear_;

    // **** What predictions are available to speed up the ICP?
    // 1) imu - [theta] from imu yaw angle - /imu topic
    // 2) odom - [x, y, theta] from wheel odometry - /odom topic
    // 3) vel - [x, y, theta] from velocity predictor - see alpha-beta predictors -
/vel topic
    // If more than one is enabled, priority is imu > odom > vel

    if (!nh_private_.getParam ("use_imu", use_imu_))
      use_imu_ = true;
    if (!nh_private_.getParam ("use_odom", use_odom_))
      use_odom_ = true;
    if (!nh_private_.getParam ("use_vel", use_vel_))
      use_vel_ = false;

    // **** How to publish the output?
    // tf (fixed_frame->base_frame),
    // pose message (pose of base frame in the fixed frame)

    if (!nh_private_.getParam ("publish_tf", publish_tf_))
      publish_tf_ = true;
    if (!nh_private_.getParam ("publish_pose", publish_pose_))
      publish_pose_ = true;
    if (!nh_private_.getParam ("publish_pose_stamped", publish_pose_stamped_))
      publish_pose_stamped_ = false;

    // **** CSM parameters - comments copied from algos.h (by Andrea Censi)

    // Maximum angular displacement between scans
    if              (!nh_private_.getParam              ("max_angular_correction_deg",
input_.max_angular_correction_deg))
      input_.max_angular_correction_deg = 45.0;

    // Maximum translation between scans (m)
    if (!nh_private_.getParam ("max_linear_correction", input_.max_linear_correction))
      input_.max_linear_correction = 0.50;

    // Maximum ICP cycle iterations
    if (!nh_private_.getParam ("max_iterations", input_.max_iterations))
      input_.max_iterations = 10;

    // A threshold for stopping (m)
    if (!nh_private_.getParam ("epsilon_xy", input_.epsilon_xy))
      input_.epsilon_xy = 0.000001;

    // A threshold for stopping (rad)
    if (!nh_private_.getParam ("epsilon_theta", input_.epsilon_theta))
      input_.epsilon_theta = 0.000001;

    // Maximum distance for a correspondence to be valid
    if                (!nh_private_.getParam               ("max_correspondence_dist",
input_.max_correspondence_dist))
      input_.max_correspondence_dist = 0.3;
```

```
    // Noise in the scan (m)
    if (!nh_private_.getParam ("sigma", input_.sigma))
      input_.sigma = 0.010;

    // Use smart tricks for finding correspondences.
    if (!nh_private_.getParam ("use_corr_tricks", input_.use_corr_tricks))
      input_.use_corr_tricks = 1;

    // Restart: Restart if error is over threshold
    if (!nh_private_.getParam ("restart", input_.restart))
      input_.restart = 0;

    // Restart: Threshold for restarting
    if             (!nh_private_.getParam             ("restart_threshold_mean_error",
input_.restart_threshold_mean_error))
      input_.restart_threshold_mean_error = 0.01;

    // Restart: displacement for restarting. (m)
    if (!nh_private_.getParam ("restart_dt", input_.restart_dt))
      input_.restart_dt = 1.0;

    // Restart: displacement for restarting. (rad)
    if (!nh_private_.getParam ("restart_dtheta", input_.restart_dtheta))
      input_.restart_dtheta = 0.1;

    // Max distance for staying in the same clustering
    if (!nh_private_.getParam ("clustering_threshold", input_.clustering_threshold))
      input_.clustering_threshold = 0.25;

    // Number of neighbour rays used to estimate the orientation
    if              (!nh_private_.getParam              ("orientation_neighbourhood",
input_.orientation_neighbourhood))
      input_.orientation_neighbourhood = 20;

    // If 0, it's vanilla ICP
    if              (!nh_private_.getParam             ("use_point_to_line_distance",
input_.use_point_to_line_distance))
      input_.use_point_to_line_distance = 1;

    // Discard correspondences based on the angles
    if (!nh_private_.getParam ("do_alpha_test", input_.do_alpha_test))
      input_.do_alpha_test = 0;

    // Discard correspondences based on the angles - threshold angle, in degrees
    if              (!nh_private_.getParam              ("do_alpha_test_thresholdDeg",
input_.do_alpha_test_thresholdDeg))
      input_.do_alpha_test_thresholdDeg = 20.0;

    // Percentage of correspondences to consider: if 0.9,
    // always discard the top 10% of correspondences with more error
    if (!nh_private_.getParam ("outliers_maxPerc", input_.outliers_maxPerc))
      input_.outliers_maxPerc = 0.90;

    // Parameters describing a simple adaptive algorithm for discarding.
    //  1) Order the errors.
    //  2) Choose the percentile according to outliers_adaptive_order.
    //     (if it is 0.7, get the 70% percentile)
    //  3) Define an adaptive threshold multiplying outliers_adaptive_mult
    //     with the value of the error at the chosen percentile.
    //  4) Discard correspondences over the threshold.
```

```
    //  This is useful to be conservative; yet remove the biggest errors.
    if               (!nh_private_.getParam               ("outliers_adaptive_order",
input_.outliers_adaptive_order))
      input_.outliers_adaptive_order = 0.7;

    if                (!nh_private_.getParam                ("outliers_adaptive_mult",
input_.outliers_adaptive_mult))
      input_.outliers_adaptive_mult = 2.0;

    // If you already have a guess of the solution, you can compute the polar angle
    // of the points of one scan in the new position. If the polar angle is not a
monotone
    // function of the readings index, it means that the surface is not visible in the
    // next position. If it is not visible, then we don't use it for matching.
    if (!nh_private_.getParam ("do_visibility_test", input_.do_visibility_test))
      input_.do_visibility_test = 0;

    // no two points in laser_sens can have the same corr.
    if               (!nh_private_.getParam               ("outliers_remove_doubles",
input_.outliers_remove_doubles))
      input_.outliers_remove_doubles = 1;

    //    If    1,    computes    the    covariance    of    ICP    using    the    method
http://purl.org/censi/2006/icpcov
    if (!nh_private_.getParam ("do_compute_covariance", input_.do_compute_covariance))
      input_.do_compute_covariance = 0;

    // Checks that find_correspondences_tricks gives the right answer
    if (!nh_private_.getParam ("debug_verify_tricks", input_.debug_verify_tricks))
      input_.debug_verify_tricks = 0;

    // If 1, the field 'true_alpha' (or 'alpha') in the first scan is used to compute
the
    //  incidence  beta,  and  the  factor  (1/cos^2(beta))  used  to  weight  the
correspondence.");
    if (!nh_private_.getParam ("use_ml_weights", input_.use_ml_weights))
      input_.use_ml_weights = 0;

    // If 1, the field 'readings_sigma' in the second scan is used to weight the
    // correspondence by 1/sigma^2
    if (!nh_private_.getParam ("use_sigma_weights", input_.use_sigma_weights))
      input_.use_sigma_weights = 0;
  }

  void LaserScanMatcher::imuCallback(const sensor_msgs::Imu::ConstPtr& imu_msg)
  {
    boost::mutex::scoped_lock(mutex_);
    latest_imu_msg_ = *imu_msg;
    if (!received_imu_)
    {
      last_used_imu_msg_ = *imu_msg;
      received_imu_ = true;
    }
  }

  void LaserScanMatcher::odomCallback(const nav_msgs::Odometry::ConstPtr& odom_msg)
  {
    boost::mutex::scoped_lock(mutex_);
    latest_odom_msg_ = *odom_msg;
    if (!received_odom_)
    {
```

```
      last_used_odom_msg_ = *odom_msg;
      received_odom_ = true;
    }
  }

  void    LaserScanMatcher::velCallback(const    geometry_msgs::TwistStamped::ConstPtr&
twist_msg)
  {
    boost::mutex::scoped_lock(mutex_);
    latest_vel_msg_ = *twist_msg;

    received_vel_ = true;
  }

  void LaserScanMatcher::cloudCallback (const PointCloudT::ConstPtr& cloud)
  {
    // **** if first scan, cache the tf from base to the scanner

    std_msgs::Header cloud_header = pcl_conversions::fromPCL(cloud->header);

    if (!initialized_)
    {
      // cache the static tf from base to laser
      if (!getBaseToLaserTf(cloud_header.frame_id))
      {
        ROS_WARN("Skipping scan");
        return;
      }

      PointCloudToLDP(cloud, prev_ldp_scan_);
      last_icp_time_ = cloud_header.stamp;
      initialized_ = true;
    }

    LDP curr_ldp_scan;
    PointCloudToLDP(cloud, curr_ldp_scan);
    processScan(curr_ldp_scan, cloud_header.stamp);
  }

  void    LaserScanMatcher::scanCallback    (const    sensor_msgs::LaserScan::ConstPtr&
scan_msg)
  {
    // **** if first scan, cache the tf from base to the scanner

    if(!initialized_)
    {
      createCache(scan_msg);    // caches the sin and cos of all angles

      // cache the static tf from base to laser
      if (!getBaseToLaserTf(scan_msg->header.frame_id))
      {
        ROS_WARN("Skipping scan");
        return;
      }

      laserScanToLDP(scan_msg, prev_ldp_scan_);
      last_icp_time_ = scan_msg->header.stamp;
      initialized_ = true;
    }

    LDP curr_ldp_scan;
```

```
  laserScanToLDP(scan_msg, curr_ldp_scan);
  processScan(curr_ldp_scan, scan_msg->header.stamp);
}

void LaserScanMatcher::processScan(LDP& curr_ldp_scan, const ros::Time& time)
{
  ros::WallTime start = ros::WallTime::now();

  // CSM is used in the following way:
  // The scans are always in the laser frame
  // The reference scan (prevLDPcan_) has a pose of [0, 0, 0]
  // The new scan (currLDPScan) has a pose equal to the movement
  // of the laser in the laser frame since the last scan
  // The computed correction is then propagated using the tf machinery

  prev_ldp_scan_->odometry[0] = 0.0;
  prev_ldp_scan_->odometry[1] = 0.0;
  prev_ldp_scan_->odometry[2] = 0.0;

  prev_ldp_scan_->estimate[0] = 0.0;
  prev_ldp_scan_->estimate[1] = 0.0;
  prev_ldp_scan_->estimate[2] = 0.0;

  prev_ldp_scan_->true_pose[0] = 0.0;
  prev_ldp_scan_->true_pose[1] = 0.0;
  prev_ldp_scan_->true_pose[2] = 0.0;

  input_.laser_ref  = prev_ldp_scan_;
  input_.laser_sens = curr_ldp_scan;
  input_.do_compute_covariance = 1;

  // **** estimated change since last scan

  double dt = (time - last_icp_time_).toSec();
  double pr_ch_x, pr_ch_y, pr_ch_a;
  getPrediction(pr_ch_x, pr_ch_y, pr_ch_a, dt);

  // the predicted change of the laser's position, in the fixed frame

  tf::Transform pr_ch;
  createTfFromXYTheta(pr_ch_x, pr_ch_y, pr_ch_a, pr_ch);

  // account for the change since the last kf, in the fixed frame

  pr_ch = pr_ch * (f2b_ * f2b_kf_.inverse());

  // the predicted change of the laser's position, in the laser frame

  tf::Transform pr_ch_l;
  pr_ch_l = laser_to_base_ * f2b_.inverse() * pr_ch * f2b_ * base_to_laser_ ;

  input_.first_guess[0] = pr_ch_l.getOrigin().getX();
  input_.first_guess[1] = pr_ch_l.getOrigin().getY();
  input_.first_guess[2] = tf::getYaw(pr_ch_l.getRotation());

  // *** scan match - using point to line icp from CSM

  sm_icp(&input_, &output_);
  tf::Transform corr_ch;

  if (output_.valid)
```

```
{
  // the correction of the laser's position, in the laser frame
  tf::Transform corr_ch_l;
  createTfFromXYTheta(output_.x[0], output_.x[1], output_.x[2], corr_ch_l);

  // the correction of the base's position, in the base frame
  corr_ch = base_to_laser_ * corr_ch_l * laser_to_base_;

  // update the pose in the world frame
  f2b_ = f2b_kf_ * corr_ch;

  // **** publish

  if (publish_pose_)
  {
    // unstamped Pose2D message
    geometry_msgs::Pose2D::Ptr pose_msg;
    pose_msg = boost::make_shared<geometry_msgs::Pose2D>();
    pose_msg->x = f2b_.getOrigin().getX();
    pose_msg->y = f2b_.getOrigin().getY();
    pose_msg->theta = tf::getYaw(f2b_.getRotation());


    laser_scan_matcher::Pose2DWithCovariance::Ptr pose_cov;
    pose_cov = boost::make_shared<laser_scan_matcher::Pose2DWithCovariance>();
    pose_cov->pose.x = pose_msg->x;
    pose_cov->pose.y = pose_msg->y;
    pose_cov->pose.theta = pose_msg->theta;

    pose_cov->covariance[0] = gsl_matrix_get(output_.cov_x_m, 0, 0);
    pose_cov->covariance[1] = gsl_matrix_get(output_.cov_x_m, 0, 1);
    pose_cov->covariance[2] = gsl_matrix_get(output_.cov_x_m, 0, 2);

    pose_cov->covariance[3] = gsl_matrix_get(output_.cov_x_m, 1, 0);
    pose_cov->covariance[4] = gsl_matrix_get(output_.cov_x_m, 1, 1);
    pose_cov->covariance[5] = gsl_matrix_get(output_.cov_x_m, 1, 2);

    pose_cov->covariance[6] = gsl_matrix_get(output_.cov_x_m, 2, 0);
    pose_cov->covariance[7] = gsl_matrix_get(output_.cov_x_m, 2, 1);
    pose_cov->covariance[8] = gsl_matrix_get(output_.cov_x_m, 2, 2);

    //pose_publisher_.publish(pose_msg);
    pose_covariance_publisher_.publish(pose_cov);

  }
  if (publish_pose_stamped_)
  {
    // stamped Pose message
    geometry_msgs::PoseStamped::Ptr pose_stamped_msg;
    pose_stamped_msg = boost::make_shared<geometry_msgs::PoseStamped>();

    pose_stamped_msg->header.stamp    = time;
    pose_stamped_msg->header.frame_id = fixed_frame_;

    tf::poseTFToMsg(f2b_, pose_stamped_msg->pose);

    pose_stamped_publisher_.publish(pose_stamped_msg);
  }
  if (publish_tf_)
  {
    tf::StampedTransform transform_msg (f2b_, time, fixed_frame_, base_frame_);
```

```
      tf_broadcaster_.sendTransform (transform_msg);
    }
  }
  else
  {
    corr_ch.setIdentity();
    ROS_WARN("Error in scan matching");
  }

  // **** swap old and new

  if (newKeyframeNeeded(corr_ch))
  {
    // generate a keyframe
    ld_free(prev_ldp_scan_);
    prev_ldp_scan_ = curr_ldp_scan;
    f2b_kf_ = f2b_;
  }
  else
  {
    ld_free(curr_ldp_scan);
  }

  last_icp_time_ = time;

  // **** statistics

  double dur = (ros::WallTime::now() - start).toSec() * 1e3;
  ROS_DEBUG("Scan matcher total duration: %.1f ms", dur);
}

bool LaserScanMatcher::newKeyframeNeeded(const tf::Transform& d)
{
  if (fabs(tf::getYaw(d.getRotation())) > kf_dist_angular_) return true;

  double x = d.getOrigin().getX();
  double y = d.getOrigin().getY();
  if (x*x + y*y > kf_dist_linear_sq_) return true;

  return false;
}

void LaserScanMatcher::PointCloudToLDP(const PointCloudT::ConstPtr& cloud,
                                       LDP& ldp)
{
  double max_d2 = cloud_res_ * cloud_res_;

  PointCloudT cloud_f;

  cloud_f.points.push_back(cloud->points[0]);

  for (unsigned int i = 1; i < cloud->points.size(); ++i)
  {
    const PointT& pa = cloud_f.points[cloud_f.points.size() - 1];
    const PointT& pb = cloud->points[i];

    double dx = pa.x - pb.x;
    double dy = pa.y - pb.y;
    double d2 = dx*dx + dy*dy;

    if (d2 > max_d2)
```

```
      {
        cloud_f.points.push_back(pb);
      }
    }

    unsigned int n = cloud_f.points.size();

    ldp = ld_alloc_new(n);

    for (unsigned int i = 0; i < n; i++)
    {
      // calculate position in laser frame
      if (is_nan(cloud_f.points[i].x) || is_nan(cloud_f.points[i].y))
      {
        ROS_WARN("Laser Scan Matcher: Cloud input contains NaN values. \
                  Please use a filtered cloud input.");
      }
      else
      {
        double r = sqrt(cloud_f.points[i].x * cloud_f.points[i].x +
                        cloud_f.points[i].y * cloud_f.points[i].y);

        if (r > cloud_range_min_ && r < cloud_range_max_)
        {
          ldp->valid[i] = 1;
          ldp->readings[i] = r;
        }
        else
        {
          ldp->valid[i] = 0;
          ldp->readings[i] = -1;  // for invalid range
        }
      }

      ldp->theta[i] = atan2(cloud_f.points[i].y, cloud_f.points[i].x);
      ldp->cluster[i]  = -1;
    }

    ldp->min_theta = ldp->theta[0];
    ldp->max_theta = ldp->theta[n-1];

    ldp->odometry[0] = 0.0;
    ldp->odometry[1] = 0.0;
    ldp->odometry[2] = 0.0;

    ldp->true_pose[0] = 0.0;
    ldp->true_pose[1] = 0.0;
    ldp->true_pose[2] = 0.0;
  }

  void     LaserScanMatcher::laserScanToLDP(const     sensor_msgs::LaserScan::ConstPtr&
scan_msg,
                                            LDP& ldp)
  {
    unsigned int n = scan_msg->ranges.size();
    ldp = ld_alloc_new(n);

    for (unsigned int i = 0; i < n; i++)
    {
      // calculate position in laser frame
```

```
    double r = scan_msg->ranges[i];

    if (r > scan_msg->range_min && r < scan_msg->range_max)
    {
      // fill in laser scan data

      ldp->valid[i] = 1;
      ldp->readings[i] = r;
    }
    else
    {
      ldp->valid[i] = 0;
      ldp->readings[i] = -1;  // for invalid range
    }

    ldp->theta[i]    = scan_msg->angle_min + i * scan_msg->angle_increment;

    ldp->cluster[i]  = -1;
  }

  ldp->min_theta = ldp->theta[0];
  ldp->max_theta = ldp->theta[n-1];

  ldp->odometry[0] = 0.0;
  ldp->odometry[1] = 0.0;
  ldp->odometry[2] = 0.0;

  ldp->true_pose[0] = 0.0;
  ldp->true_pose[1] = 0.0;
  ldp->true_pose[2] = 0.0;
}

void LaserScanMatcher::createCache (const sensor_msgs::LaserScan::ConstPtr& scan_msg)
{
  a_cos_.clear();
  a_sin_.clear();

  for (unsigned int i = 0; i < scan_msg->ranges.size(); ++i)
  {
    double angle = scan_msg->angle_min + i * scan_msg->angle_increment;
    a_cos_.push_back(cos(angle));
    a_sin_.push_back(sin(angle));
  }

  input_.min_reading = scan_msg->range_min;
  input_.max_reading = scan_msg->range_max;
}

bool LaserScanMatcher::getBaseToLaserTf (const std::string& frame_id)
{
  ros::Time t = ros::Time::now();

  tf::StampedTransform base_to_laser_tf;
  try
  {
    tf_listener_.waitForTransform(
      base_frame_, frame_id, t, ros::Duration(1.0));
    tf_listener_.lookupTransform (
      base_frame_, frame_id, t, base_to_laser_tf);
  }
  catch (tf::TransformException ex)
```

```
    {
      ROS_WARN("Could  not  get  initial  transform  from  base  to  laser  frame,  %s",
ex.what());
      return false;
    }
    base_to_laser_ = base_to_laser_tf;
    laser_to_base_ = base_to_laser_.inverse();

    return true;
  }

  // returns the predicted change in pose (in fixed frame)
  // since the last time we did icp
  void LaserScanMatcher::getPrediction(double& pr_ch_x, double& pr_ch_y,
                                       double& pr_ch_a, double dt)
  {
    boost::mutex::scoped_lock(mutex_);

    // **** base case - no input available, use zero-motion model
    pr_ch_x = 0.0;
    pr_ch_y = 0.0;
    pr_ch_a = 0.0;

    // **** use velocity (for example from ab-filter)
    if (use_vel_)
    {
      pr_ch_x = dt * latest_vel_msg_.twist.linear.x;
      pr_ch_y = dt * latest_vel_msg_.twist.linear.y;
      pr_ch_a = dt * latest_vel_msg_.twist.angular.z;

      if      (pr_ch_a >= M_PI) pr_ch_a -= 2.0 * M_PI;
      else if (pr_ch_a < -M_PI) pr_ch_a += 2.0 * M_PI;
    }
    // **** use wheel odometry
    if (use_odom_ && received_odom_)
    {
      pr_ch_x = latest_odom_msg_.pose.pose.position.x -
                last_used_odom_msg_.pose.pose.position.x;

      pr_ch_y = latest_odom_msg_.pose.pose.position.y -
                last_used_odom_msg_.pose.pose.position.y;

      pr_ch_a = tf::getYaw(latest_odom_msg_.pose.pose.orientation) -
                tf::getYaw(last_used_odom_msg_.pose.pose.orientation);

      if      (pr_ch_a >= M_PI) pr_ch_a -= 2.0 * M_PI;
      else if (pr_ch_a < -M_PI) pr_ch_a += 2.0 * M_PI;

      last_used_odom_msg_ = latest_odom_msg_;
    }
    // **** use imu
    if (use_imu_ && received_imu_)
    {
      pr_ch_a = tf::getYaw(latest_imu_msg_.orientation) -
                tf::getYaw(last_used_imu_msg_.orientation);

      if      (pr_ch_a >= M_PI) pr_ch_a -= 2.0 * M_PI;
      else if (pr_ch_a < -M_PI) pr_ch_a += 2.0 * M_PI;
      last_used_imu_msg_ = latest_imu_msg_;
    }
  }
```

```
  void LaserScanMatcher::createTfFromXYTheta(
    double x, double y, double theta, tf::Transform& t)
  {
    t.setOrigin(tf::Vector3(x, y, 0.0));
    tf::Quaternion q;
    q.setRPY(0.0, 0.0, theta);
    t.setRotation(q);
  }
  } // namespace scan_tools
```

A.3. Archivo pruebas.launch

```
  /*******************************************************************************
  *
  * Fichero .launch pruebas.lauchn
  *
  *******************************************************************************
  *
  * AUTORES Oliver Martinez Novo
  *
  * FECHA 2015
  *
  * DESCRIPCION Archivo de configuración de arranque de los distintos programas que
componen el sistema
  *
  *******************************************************************************/

<launch>
      <node pkg="depthimage_to_laserscan" type="depthimage_to_laserscan"
name="DepthImageLaserScan" />
      <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_1" args="0 0
0 0 0 0 1 base_link LaserH1 33" />
      <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_2" args="0 0
0 0 0 0 1 base_link LaserH2 33" />
      <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_3" args="0 0
0 0 0 0 1 base_link LaserH3 33" />
      <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_V1" args="0
0 0 1,5708 0 0 1 base_link LaserV1 33" />
      <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_V2" args="0
0 0 1,5708 0 0 1 base_link LaserV2 33" />
      <node pkg="tf" type="static_transform_publisher" name="tf_laser_scan_V3" args="0
0 0 1,5708 0 0 1 base_link LaserV3 33" />

      <node pkg="laser_scan_matcher" type="laser_scan_matcher_node" name="LaserH1">
        <param name="_use_odom" type="bool" value="false" />
        <param name="_use_imu" type="bool" value="false" />
             <remap from="scan" to="scanh1" />
        <remap from="pose2DCov" to="pose2D_H1" />
      </node>

      <node pkg="laser_scan_matcher" type="laser_scan_matcher_node" name="LaserH2">
        <param name="_use_odom" type="bool" value="false" />
```

```
    <param name="_use_imu" type="bool" value="false" />
         <remap from="scan" to="scanh2" />
    <remap from="pose2DCov" to="pose2D_H2" />
  </node>

  <node pkg="laser_scan_matcher" type="laser_scan_matcher_node" name="LaserH3">
    <param name="_use_odom" type="bool" value="false" />
    <param name="_use_imu" type="bool" value="false" />
         <remap from="scan" to="scanh3" />
    <remap from="pose2DCov" to="pose2D_H3" />
  </node>

  <node pkg="laser_scan_matcher" type="laser_scan_matcher_node" name="LaserV1">
    <param name="_use_odom" type="bool" value="false" />
    <param name="_use_imu" type="bool" value="false" />
    <remap from="scan" to="scanv1" />
    <remap from="pose2DCov" to="pose2D_V1" />
  </node>

  <node pkg="laser_scan_matcher" type="laser_scan_matcher_node" name="LaserV2">
    <param name="_use_odom" type="bool" value="false" />
    <param name="_use_imu" type="bool" value="false" />
    <remap from="scan" to="scanv2" />
    <remap from="pose2DCov" to="pose2D_V2" />
  </node>

  <node pkg="laser_scan_matcher" type="laser_scan_matcher_node" name="LaserV3">
    <param name="_use_odom" type="bool" value="false" />
    <param name="_use_imu" type="bool" value="false" />
         <remap from="scan" to="scanv3" />
    <remap from="pose2DCov" to="pose2D_V3" />
  </node>

</launch>
```

Bibliografía

[1] Microsoft para kinect, https://www.microsoft.com/en-us/kinectforwindows/

[2] Placa Odroid U3 http://www.hardkernel.com/main/products/prdt_info.php?g_code=G138745696275

[3] Arducopter, http://www.arducopter.co.uk/

[4] Ardupilot, http://copter.ardupilot.com/?lang=es

[5] Institute for Dynamic Systems and Control, http://www.idsc.ethz.ch/

[6] Andrea Censi, Canonical Scan Matcher, Oct 2013, http://censi.mit.edu/software/csm/

[7] Documentación oficial de ROS, http://wiki.ros.org/

[8] Placa Arduino, https://www.arduino.cc/

[9] Sistema operativo Android, https://www.android.com/

[10] Driver controlador OpenNi, https://github.com/OpenNI/OpenNI

[11] Libreria DepthImage To LaserScan, http://wiki.ros.org/depthimage_to_laserscan

[12] Librería Laser Scan Matcher, http://wiki.ros.org/laser_scan_matcher

[13] Sistema operative Ubuntu, http://www.ubuntu.com/download/desktop

[14] Instalación Ubuntu, http://wiki.ros.org/indigo/Installation/Ubuntu

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