Escuela Politécnica Superior de Jaén

Grado en Ingeniería Electrónica Industrial

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

FUSIÓN SENSORIAL PARA REALIZAR TAREAS ROBÓTICAS EN ENTORNOS PARCIALMENTE ESTRUCTURADOS

Alumno: José Antonio Moral Parras

Tutor: Prof. D. Pablo Cano Marchal
Prof. D. Diego Manuel Martínez Gila
Dpto: Ingeniería Electrónica y Automática

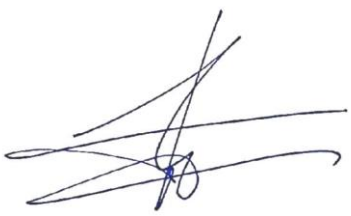
Septiembre, 2017

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Ingeniería Electrónica y Automática

Don Pablo Cano Marchal, tutor del Proyecto Fin de Carrera titulado: **"Fusión Sensorial para realizar tareas robóticas en entornos parcialmente estructurados"**, que presenta José Antonio Moral Parras, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, septiembre de 2017

El alumno: Los tutores:



Moral Parras,

José Antonio

Cano Marchal, Pablo / Martínez Gila, Diego Manuel

RESUMEN

El presente proyecto versa sobre la integración de sensorística en la plataforma robótica ROS, a fin de proporcionar información a un robot acerca de un entorno parcialmente desconocido.

Para ello, se desarrollará una metodología de aprendizaje, selección de material y búsqueda de información sobre el mismo, que permita establecer una red de sensores capaz de obtener los datos requeridos.

Se procurará que dichos sensores estén ya total o parcialmente integrados en ROS, aprovechando las bondades *open-source* del *framework*, de forma que en este documento se trate principalmente acerca de la dotación de funcionalidad a los mismos.

Por último, se indica que los esfuerzos encaminados a obtener información del entorno se centrarán a la detección de personas en el mismo, para su posterior interacción con el robot.

ABSTRACT

This project focuses on integrating different sensors within ROS robotic platform, in order to make information about a partially unknown environment available.

To achieve that, a specific methodology, based on learning, choosing the material and gathering information about it will be developed, so a sensoristic network that can acquire and process the required data could be established.

Sensors should be total or partially integrated within ROS, taking advantage of framework's open-source features; because of that, this document will primarily focus on providing functionality to them.

To finish this abstract, it is necessary to point out that all efforts made to retrieve information about the environment will focus on detecting people within it, for the robot to interact with them.

Índice

Listado de figuras

Glosario de acrónimos, siglas y definiciones

AMCL: *Adaptive Monte Carlo Localization*; algoritmo probabilístico de localización empleado para posicionar a un robot que se mueve en un entorno 2D, con ayuda de un mapa.

API: *Application Programming Interface*; bibliotecas usadas en programación, que permiten realizar un proceso de abstracción entre niveles inferiores y superiores de *software* para facilitar su comprensión.

Bash: *Bourne again shell*; programa informático, cuya función es interpretar órdenes de Unix.

CMOS: C*omplementary Metal-Oxide-Semiconductor*; familia lógica empleada en la fabricación de circuitos integrados (nivel de tensión habitual: 3.3V), así como sensores ópticos (también llamados de píxeles activos).

DHCP: *Dynamic Host Configuration Protocol*; servidor que usa un protocolo de red cliente/servidor y que posee una lista de direcciones IP dinámicas, que asigna a los clientes conforme se van conectando a la red.

Driver: elemento *software* utilizado en informática para interactuar con un elemento periférico, abstrayéndose del *hardware* y proporcionando a veces una interfaz de usuario.

FNN: *Feedforward Neural Network*;

FPGA: *Field Programmable Gate Array*; dispositivo de *hardware* programable, conformado por bloques de puertas lógicas.

Frame: también llamado fotograma, es el nombre que recibe cada una de las imágenes instantáneas que conforman una secuencia de vídeo, que, al ser reproducidas de forma secuencial, generan sensación de movimiento. En el contexto de TF, *frame* significa “representación de un punto en el espacio respecto de otro punto de referencia”.

Framework: consiste en un conjunto de pautas, patrones y normas dictaminadas en programación para llevar a cabo el desarrollo y la implementación de una aplicación concreta.

IDE: *Integrated Development Environment*; es un programa que ofrece un conjunto de herramientas para asistir y ayudar al programador en la tarea de creación de *software*.

IMU: *Inertial Measurement Unit*; unidad de medición inercial, compuesta fundamentalmente por tres sensores: acelerómetro, giroscopio y brújula, cada uno de los cuales suele proporcionar tres grados de libertad.

ISO: *International Standards Organization*.

LIDAR: *Light Detection and Ranging*; sensor que permite calcular la distancia entre sí mismo y un objeto emitiendo un haz láser pulsante.

Meka: abreviatura para referirse al robot Meka M1 de Meka Robotics, un robot humanoide manipulador.

Middleware: *software* que proporciona una proporciona una capa facilitadora e integradora, que facilita la comunicación, conexionado y transferencia de información entre los diferentes elementos que conforman un sistema colaborativo (aplicaciones, paquetes de programas, redes, *hardware* y/o sistemas operativos).

Odometría: término empleado para definir el hecho de que un vehículo sobre ruedas sea capaz de estimar su posición durante la navegación, empleando para ello información sobre las rotaciones realizadas por él mismo.

OSI: *Open System Interconnection* o modelo de interconexión de sistemas abiertos.

P2P: *Peer-To-Peer;* red de dispositivos informáticos que funcionan de forma distribuida, como una serie de nodos iguales entre sí.

PCL: *PointCloud Library;* librería de *software* específicamente programada para el tratamiento y procesamiento de nubes de puntos, que son unas estructuras de datos que representan una visión tridimensional del espacio desde un punto de origen o referencia; es muy empleada en visión por computador, para procesar la información proveniente de dispositivos capaces de proporcionar medidas en profundidad, como láseres o TOFs.

PMB – 2: *Platform Mobile Base*; base robótica móvil con capacidad de navegación autónoma.

Pose: Tipo de mensaje de ROS, que proporciona tanto la posición como orientanción de un punto determinado en el espacio, respecto a un sistema de origen de coordenadas.

RGB-D: *Red Green Blue – Depth*; tipo de imagen estereoscópica a color, que contiene tanto información visual como tridimensional del entorno.

ROI: *Region Of Interest*; en visión por computador, nombre con el que se especifica qué parte en concreto de una imagen resulta de interés a la hora de extraer información útil de la misma.

ROS: *Robot Operating System*; es un *framework* empleado para el desarrollo de *software* para robots.

TOF: *Time of Flight;* acrónimo empleado para referirse a las cámaras de tiempo de vuelo, que son utilizadas para obtener imágenes en tres dimensiones del entorno.

TTL: *Transistor – Transistor Logic*; tecnología electrónica empleada en la construcción de circuitos integrados, que utiliza como base para ello los transistores bipolares.

URDF: *Unified Robot Description Format*; formato basado en XML para representar el modelo de un robot.

XML: *eXtensible Markup Language*; lenguaje de marcado, fundamentalmente empleado para almacenar datos de forma legible.

1. Introducción, motivación y objetivos

Los sensores son elementos capaces de generar una respuesta eléctrica ante el estímulo provocado por el cambio en una magnitud física o química, obteniendo información útil que sirve para conocer el entorno que lo rodea; dicha información puede emplearse, por ejemplo, para que un robot pueda desenvolverse en un **entorno total o parcialmente desconocido**. Para ello, el robot obtiene y procesa información proveniente de dicho entorno, mediante dichos sensores, que se comportan como sistemas de adquisición de datos; un claro ejemplo podría ser un sistema de visión por computador. Ya que hay una constante realimentación proveniente de la información captada por los sensores que integran, los robots funcionan mediante lazos cerrados de control.

El presente trabajo se justifica como un intento de aportar funcionalidades prácticas a los robots en ámbitos de interacción con seres humanos, de forma que se sienten las bases para futuros avances en la materia. Para ello, se explorarán diferentes alternativas para la **detección de personas en ámbitos parcialmente desconocidos**, de cara a facilitar información para la interacción de un robot con las mismas; por todo esto, una de las principales motivaciones que llevan a abordar este proyecto es su atractivo, ya que versa sobre un tema interesante, útil y que implica el uso de conocimientos relacionados con varios ámbitos de la ingeniería, centrándose en los de la programación y la visión por computador. Además, supone un gran reto, ya que tomando como referencia los contenidos cursados a lo largo de la titulación, exige un considerable periodo de formación, familiarización y aprendizaje, antes de empezar a obtener los resultados deseados; esto sirve como un gran aliciente a la realización del proyecto, que no es más que el de afrontarlo, adquiriendo nuevos y útiles conocimientos por el camino.

Otro de los grandes atractivos de este proyecto es el de colaborar sentando las bases de lo que, en un futuro, podría ser un **sistema de interacción humano - robot** más complejo, que permita a las personas beneficiarse de los robots como herramienta de ayuda o apoyo en tareas determinadas, que bien sean difícilmente realizables para la propia persona o que la ayuden a cumplir sus objetivos e interaccionar con el entorno. Para conseguirlo, se trabajará siguiendo un patrón

común en ingeniería, que es el de la segmentación de un problema en partes más pequeñas para su posterior abordamiento y resolución; así, el caso de estudio se dividirá en dos partes diferenciadas, como son la **obtención de información acerca de un entorno parcialmente desconocido** (temática que se abordará en el presente trabajo), enfocándose en la detección de personas en escena; y la navegación de un robot móvil hacia las mismas, para su posterior interacción con ellas; éstas últimas tareas serán tratadas y resueltas en un proyecto independiente, llamado **"Inclusión del robot PMB-2 en la plataforma basada en ROS"** [1] que, no obstante, comparte información y convive con el presente para la consecución de los objetivos propuestos, fusionándose ambos finalmente en una única aplicación capaz de realizar tareas más complejas que por separado.

También cabe destacar que, abordando este trabajo, se espera trabajar con robots, sensores y material en general de altas prestaciones, siendo esta una oportunidad única para alcanzar a comprender cómo funcionan dichos elementos, punteros en investigación y desarrollo; llegándose incluso a colaborar dicho desarrollo, mediante la dotación de funcionalidad al sistema en su conjunto para la consecución de tareas específicas. Por todo ello, se pretende conseguir una formación básica/intermedia en la materia, que pueda servir como base para un futuro aprendizaje e incluso beneficiosa a la hora de intentar acceder al sector profesional de la robótica.

Antes de hablar de los objetivos a cumplir, es importante resaltar el hecho de que, dado que la prioridad es emplear información proporcionada por diferentes sensores para realizar una tarea concreta, es justificable que este proyecto no se centre en el desarrollo desde cero del *software* específico necesario para integrar dichos sensores en una plataforma *software* especializada en robótica, sino que se aprovechará todo el **código *open-source*** (de código abierto) disponible en la red; evitándose, en la medida de lo posible, realizar las tediosas tareas de programar el *hardware* desde cero, y centrando los esfuerzos en, utilizando los datos del mismo, darles uso y aplicabilidad para dotar al sistema de las funcionalidades necesarias para extraer información del entorno parcialmente desconocido objeto del caso de estudio. Debido a todos estos requerimientos (*hardware* utilizable con código abierto, integración de sensores y robots en una única aplicación, realización de tareas

concretas…) se hace necesario buscar una plataforma *software* que sirva de nexo o ***middleware*** de unión entre todos los elementos.

El objetivo general del proyecto se podría sintetizar en la **búsqueda, integración y programación de sensores, que permitan facilitar información a un robot móvil acerca de un entorno parcialmente desconocido para el mismo, y que le habiliten para realizar tareas e interaccionar en dicho entorno**. Concretando un poco más, se pretende emplear la visión por computador, utilizando para ello sensores específicamente diseñados para tal fin (cámaras), así como otros auxiliares, para detectar, localizar e identificar a personas presentes en un espacio determinado con la finalidad de que un robot pueda navegar hacia ellas y realizar diversas tareas de interacción humano – robot. Para ello, se cumplirán los siguientes objetivos parciales:

- Utilizar una plataforma ***middleware*** de comunicación e intercambio de información con otros dispositivos, con la finalidad de aportar datos a un robot acerca de un entorno parcialmente desconocido para su posterior uso; dicha plataforma será **ROS (*Robot Operating System*)**, por los motivos que, más adelante, se indican en el segundo punto del presente documento.
- Familiarizarse con la arquitectura de la plataforma ROS, así como asimilar los conceptos y la jerarquía que dicha plataforma utiliza.
- Realizar una selección de **sensores compatibles con ROS**, aprovechando las facilidades que ofrece dicha plataforma, como la inclusión de *drivers open-source* para la programación e integración de los mismos en sistemas más complejos.
- Diseñar, programar y depurar aplicaciones y algoritmos que permitan, a partir de la información suministrada por los sensores seleccionadas, cumplir con las tareas de detección, localización e identificación de personas en entornos parcialmente estructurados.
- Definir un caso de estudio en el que ejemplificar y probar todo lo anteriormente desarrollado, y llevarlo a cabo y ejecutarlo en conjunción y estrecha colaboración con el proyecto [1].

Para la consecución de los objetivos indicados, se propone la siguiente metodología de trabajo:

- Aprendizaje, familiarización y dominio de la plataforma ROS, que se empleará como *middleware* de comunicaciones entre robots y sensores.
- Adquisición de conocimientos medios/avanzados de programación en C++/Python, que son los lenguajes a utilizar para el desarrollo de las diferentes aplicaciones.
- Selección de los sensores más apropiados para cubrir la problemática planteada, teniendo en cuenta tanto su utilidad y aplicabilidad a la solución como su fácil integración en el sistema ROS; se tendrá preferencia por aquellos para los que existan APIs o librerías *open-source* total o parcialmente desarrolladas.
- Definición de las tareas concretas a realizar por cada sensor, delimitándose para ello la información que se debe obtener con cada uno de ellos.
- Programación del código necesario para obtener, procesar y facilitar al sistema dicha información, a fin de que luego pueda ser empleada por el robot para un mayor conocimiento del entorno que lo rodea.
- Realización de pruebas y ensayos encaminados a depurar y mejorar el sistema, primero de forma aislada y luego en el entorno de trabajo e interacción entre robot y humanos.

Como instrumentos de trabajo, aparte de los diferentes sensores a integrar en la plataforma ROS para cumplir un objetivo específico según la aplicación requerida, se empleará, principalmente, un PC con Ubuntu instalado directamente en el disco duro; los motivos por los que se ha prescindido de realizar una virtualización de dicho sistema operativo bajo Windows se justificarán más adelante, pero, básicamente, esta decisión se ha tomado teniendo en cuenta los recursos requeridos para una adecuada computación de la información a procesar por el sistema. Se opta por utilizar Ubuntu como sistema operativo soporte porque, a pesar de que ROS también está disponible para Windows, resulta ser más inestable, además de que la gran parte de *drivers* para sensores de robots se encuentran programados en librerías Linux, dado el carácter abierto del mismo.

Una vez citados los objetivos a cumplir, así como la metodología a desarrollar para ello, es conveniente delimitar brevemente el caso de estudio a resolver, teniéndose en cuenta que, para que los sensores realicen adecuadamente la tarea de

extraer información de un entorno parcialmente desconocido, dicho entorno deberá cumplir unas condiciones de contorno, a definir al inicio del desarrollo de la aplicación, para procurar que se respeten la mayor parte de variables en juego. Mayores detalles acerca de este tema pueden consultarse en el capítulo cinco.

La plataforma en mente constará de:

- Dos cámaras: una TOF (cámara de tiempo de vuelo), que irá fija en un extremo de la sala, y una sensor RGB-D, montado encima del robot móvil.
- Dos robots [1], siendo uno plataforma móvil con capacidad de navegación autónoma y detección de obstáculos; y un robot humanoide con siete grados de libertad.

Todos los apartados posteriores se encaminan a proporcionar una base sólida sobre la que ejemplificar el caso de estudio mencionado, y versan sobre: conceptos y estructura del *middleware* ROS; definición, integración y puesta en marcha de los elementos sensorísticos empleados; programación de algoritmos y aplicaciones que utilicen dichos sensores y la información que generan para cumplir con los objetivos marcados; caso de estudio y, finalmente, conclusiones.

2. ROS (Robot Operating System)

En este apartado de la memoria se va a explicar la estructura de la plataforma software elegida como medio de comunicación e intercambio de datos entre todos los elementos que conformarán la aplicación, que no es otra que ROS [2]. Los motivos por los que esta plataforma ha sido seleccionada frente a otras, aunque brevemente citados en la introducción, son: carácter *open-source*; compatibilidad y soporte para gran cantidad de sensores y actuadores; facilidad para reutilizar código y escribirlo en varios lenguajes de programación; estructura topológica nodular, que permite segmentar la aplicación en subnúcleos independentes y comunicables entre sí… Además, ROS también resulta ser la mejor elección por estar ya presente como sistema operativo principal en los dos robots que se van a utilizar en el futuro caso de estudio: PMB-2 y Meka M1. Directrices sobre cómo realizar la instalación de ROS pueden encontrarse en el **Anexo I (I)**.

2.1. ¿Qué es ROS?

ROS [3] [4] (Robot Operating System) es un *framework* (conjunto estandarizado de conceptos, convenios, criterios y prácticas establecidos para resolver un tipo de problemática particular) para escribir y desarrollar software para robots. ROS se define como el conjunto de herramientas, librerías y convenios cuyo objetivo es resolver la problemática de crear comportamientos robóticos robustos y complejos, simplificando dicha tarea y extendiéndola a una amplia variedad de robots y plataformas.

2.2. Antecedentes históricos y presente de ROS

ROS fue originalmente desarrollado [3] en 2007 por el Laboratorio de Inteligencia Artificial de Stanford (SAIL en inglés), enfocado al desarrollo del STAIR [5] (Stanford Artificial Intelligence Robot). El objetivo era desarrollar un robot capaz de navegar en entorno de hogar y oficina, manipulando objetos y herramientas e interactuando y ayudando a personas en diferentes tareas. Para ello, se requería un *software* flexible y dinámico, capaz de adaptarse a las diferentes variaciones requeridas por las condiciones de trabajo del robot con el mínimo número de modificaciones posible en el código; este concepto se fue ampliando hasta nacer ROS.

La comunidad de ROS está creciendo muy rápido y existen desarrolladores en todo el mundo. Muchas compañías robóticas están desplazando su *software* para que sea compatible con ROS. Esta tendencia es visible en los robots industriales [6], donde las compañías (Figura 1) cambian el software propietario (que además supone un coste elevado) a ROS.

Figura 1: Diversidad de empresas que utilizan ROS Industrial

El movimiento de ROS aplicado a la industria está ganando terreno en los últimos años, con lo que puede generar muchas oportunidades de trabajo, por lo que, en los próximos años, ROS probablemente será un requerimiento esencial para un ingeniero dedicado a la robótica.

2.3. Razón de uso en el presente trabajo

ROS resulta muy conveniente para la integración de sensores en plataformas robóticas, debido a que proporciona por defecto mensajes estandarizados que permiten adecuar la información captada por los mismos a un lenguaje procesable por computadores y comprensible para los robots; esto se consigue mediante la integración de las APIs de los sensores, comúnmente programadas en C o C++, en ROS, realizándose posteriormente la conversión de los datos “en bruto” a los anteriormente citados tipos de mensajes específicos de ROS; posteriormente, estos mensajes pueden retransmitirse al sistema a través de tópicos a los que el propio robot tendrá acceso, para tomar decisiones; o bien que otros nodos pueden utilizar para procesar o transformar la información recibida a través de dichos tópicos.

Por lo tanto, y ya que la mayor parte de los sensores empleados en robótica (y la totalidad de los empleados en este proyecto) se encuentran ya integrados en ROS,

el usuario de dicha sensorística sólo deberá preocuparse de utilizar la información disponible en los tópicos publicados por estos sensores para filtrarla, tratarla y tomar las decisiones pertinentes en base a ella.

Además, y ya que la tarea principal es facilitar esta información a robots para que interactúen con el entorno parcialmente estructurado en el que se encuentran, ROS ofrece herramientas adecuadas para actuar como *middleware* de comunicación entre los diferentes dispositivos que conforman el sistema, facilitando en gran medida la conexión entre ellos (incluso entre varios PC). Dichas capacidades como *middleware*, además del establecimiento de protocolos específicos para la programación, estructuración y división de tareas, herramientas de visualización y desarrollo, mensajes específicos… conforman un *framework* polivalente para una amplia variedad de sensores, robots y dispositivos.

Las principales ventajas [7] que ofrece ROS para la programación e integración de sensores y robots en aplicaciones conjuntas son:

- Utilidades de alto nivel, capaces de integrar de forma sencilla para el usuario la navegación y manipulación de robots, mediante paquetes ya preinstalados.
- Herramientas de visualización, desarrollo, comprobación de errores y simulación.
- Soporte multiplataforma y multilenguaje: a pesar de que Ubuntu es el sistema operativo preferible para ejecutar ROS, también puede emplearse en otros sistemas. Además, ofrece la posibilidad de desarrollar código propio en Python, C++ u otros lenguajes.
- Capacidad para gestionar tareas multihilo y concurrentes de forma eficiente.
- **Independencia y modularidad**: muchos nodos siguen funcionando si otros fallan, lo que permite no detener completamente la aplicación y localizar más fácilmente dónde se encuentra el error. Además, el hecho de ser modular permite que los subprogramas sean fácilmente modificables y exportables a otras aplicaciones, con el mínimo esfuerzo.
- Amplia comunidad de desarrolladores, que ofrecen tanto soporte y ayuda como nuevas contribuciones de código abierto y gratuito; esto permite la reutilización de código, lo cual hace que no haya que comenzar desde cero cada proyecto y permite simplificar la tarea de acometerlos, puesto que

simplemente hay que centrarse en aportar nuevas funcionalidades o características que, además, conviene compartir para que la comunidad siga creciendo y evolucionando, en un continuo proceso de realimentación por parte de los usuarios.

- **Soporte para multitud de sensores y actuadores**, como LIDAR, servos, cámaras de profundidad o Kinect, entre otros; este es, probablemente, el punto más interesante en cuanto refiere a este trabajo, ya que ROS ofrece la ventaja de que permite la fácil integración y programación de gran variedad de sensores en su plataforma, gestionando la información de los mismos a través de tópicos y permitiendo una fácil configuración de sus parámetros. Además, debido a que ROS es un proyecto *open-source*, gran parte de los sensores más comúnmente utilizados en el ámbito de la automática, robótica y visión por computador se encuentran ya integrados en la plataforma, con lo que su utilización resulta aún más sencilla, puesto que limita la tarea del ingeniero a adaptar la información ofrecida por dichos sensores en ROS para su aplicación final.

Todas estas ventajas hacen que ROS sea la plataforma de programación para robots más utilizada en la actualidad, y la escogida para la realización de este proyecto; no obstante, se citan también algunas desventajas [8] del *framework*:

- Curva de aprendizaje muy pronunciada al inicio para comprender el funcionamiento básico, que retrasa el desarrollo de aplicaciones propias.
- Necesidad de un ordenador que controle la red distribuida; en el caso de sistemas más simples, como los basados en microcontroladores, ROS sería innecesario; no obstante, para el presente trabajo, el uso de computador y de ROS se hace indispensable, ya que es complejo y necesita de segmentación y distribución, así como de fácil conversión entre los diferentes tipos de datos.
- Dificultad a la hora de encontrar *drivers* para los diferentes sensores; esta desventaja no radica tanto en la falta de controladores, sino en que, a veces, existe exceso de los mismos en la red, debido al carácter abierto de ROS, lo que hace que pueda ser confuso el hecho de escoger qué *driver* es el más adecuado para cada sensor, teniendo en cuenta la versión de sistema operativo, el hardware del ordenador y la versión del propio sensor.

2.4. Estructura de ROS

ROS se puede representar como una **red topológica** formada por nodos o programas, que realizan tareas específicas y se comunican entre sí para intercambiar información. Este *framework* consta de cuatro [9] componentes fundamentales: el núcleo, la infraestructura de comunicaciones, las utilidades específicas para robots y las herramientas de desarrollo y visualización.

- Núcleo: llamado “roscore”, [10] consta de una serie de programas que son requisito indispensable para el funcionamiento de un sistema basado en ROS. Proporciona la habilidad de comunicación punto a punto (P2P) entre nodos, así como nombre y servicios de registro para cada uno de ellos (a través del máster); además, el core se encarga de establecer un servidor de parámetros general a todo el sistema.

  Una vez que los nodos se registran en el *roscore*, comunican al mismo la información que suministran, y aquella a la que desean suscribirse, que provendrá típicamente de otros nodos conectados a la red; será el núcleo o **máster** el encargado de proporcionar a los nodos las direcciones de los otros nodos con los que se desea establecer la comunicación (Figura 2); por lo tanto, aunque los nodos estén conectados entre sí (a través de tópicos, como se explicará más adelante), virtualmente todos ellos están relacionados con el *roscore*.

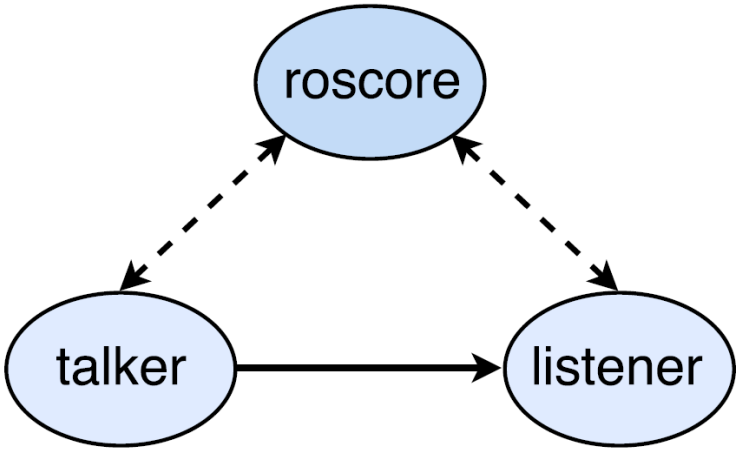


Figura 2: Función del “roscore” o núcleo de ROS

- Infraestructura de comunicaciones: el intercambio de información entre programas se realiza a través de mensajes, por medio del método de publicación/suscripción. Dicho método de comunicación facilita la divisón y estructuración del código, lo que permite que este pueda ser reutilizado en el futuro con facilidad. Los mensajes [11] que los diferentes nodos o programas intercambian entre sí pueden pertenecer a tipos ya definidos por ROS y habituales en las aplicaciones de desarrollo de robots, tales como números, matrices o estructuras más complejas, como información de sensores en entornos tridimensionales; o también pueden pertenecer a tipos definidos por el propio usuario, que puede así establecer su propio protocolo de intercambio de información entre nodos.

  Además, los mensajes pueden transmitirse de forma asíncrona o síncrona, de forma que los nodos simplemente utilicen la información proveniente de otros nodos cuando estos la publiquen; o permitiendo interacciones del tipo cliente/servidor, habilitándose así la posibilidad de establecer prioridades y permisos entre nodos. Finalmente, también existe un servidor de parámetros, que permite modificar manual o automáticamente los ajustes de las diferentes tareas en tiempo de ejecución.

- Utilidades específicas para robots: debido a que ROS es un *framework* enfocado al desarrollo de aplicaciones y software específico para robots, incluye facilidades para la consecución de dichas tareas, tales como:

  - Mensajes estandarizados para robots, que cubren la mayoría de casos habituales en robótica: posiciones, transformadas, vectores… para el caso de los modelos geométricos; nubes de puntos, escaneos provenientes de láseres, imágenes… para adquirir información del entorno a través de sensores; y datos de odometría, mapas y rutas, indispensables para la navegación.
  - Lenguaje específico para la descripción de robots, en formato URDF, que consiste en documentos XML en los que se especifican las propiedades físicas de las partes del robot, así como la forma de interconexión entre cada una de ellas. Esto permite la fácil visualización de los robots en entornos de

simulación trimidensionales como, por ejemplo, **Gazebo** o **Rviz** [12], ya incluidos en el propio ROS.

- Herramientas de ayuda a la navegación, localización y posicionamiento de robots.
- Herramientas de diagnóstico y detección de errores.

- Desarrollo y visualización: entre ellas, son destacables los comandos de consola específicamente diseñados para ROS, que facilitan la creación, compilación y edición de paquetes y nodos, así como la navegación entre ellos y el lanzamiento y puesta en ejecución de los mismos. Además, ROS ofrece varias herramientas de visualización, tales como **Rviz** (para visualizar la información proveniente de sensores y los modelos cinemáticos de los robots, así como la navegación en el entorno de los mismos) y **Rqt** (creación de interfaces gráficas y visualización de datos). Por último, hay que destacar que dichas funcionalidades se pueden ampliar gracias al amplio catálogo de software libre disponible en los repositorios de ROS para su descarga y utilización.

2.5. Nivel del sistema de archivos de ROS

Para comprender correctamente cómo funciona ROS, primero se hará hincapié en cómo se organizan los archivos [13] en el disco, como se muestra en la Figura 3:

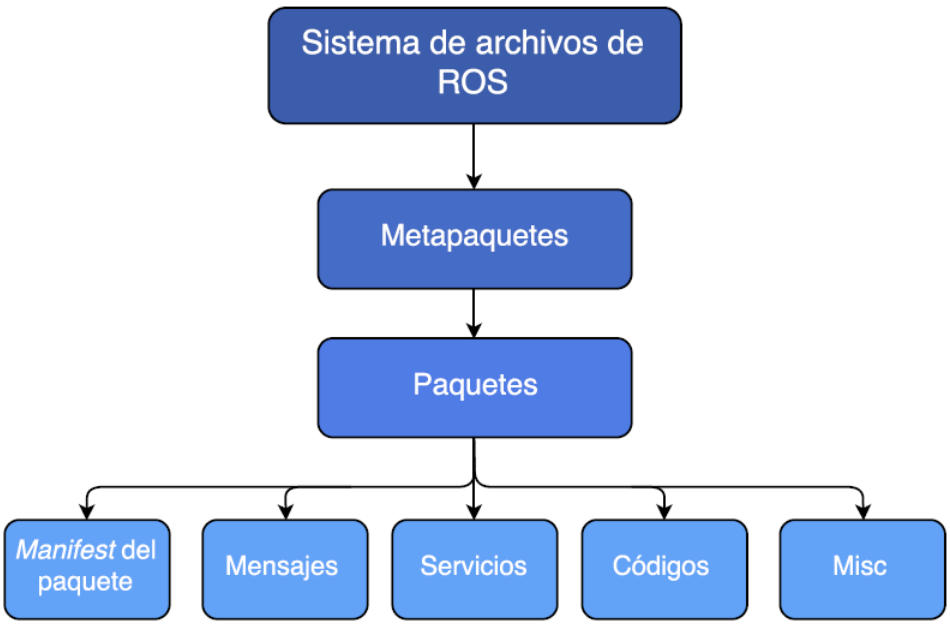


Figura 3: Sistema de archivos de ROS

- Paquetes: son la unidad básica y central de software en ROS. Contienen ejecutables (nodos), librerías, archivos de configuración, archivos de lanzamiento y nuevos tipos de mensajes, servicios y acciones que no sean los ya incluidos por ROS.
- Manifiesto del paquete: es un archivo que contiene información del propio paquete, autor, licencias y las dependencias de otros paquetes. El nombre que suele tener es “package.xml”.
- Meta-paquetes: Es utilizado para un grupo de paquetes con un propósito específico, donde los paquetes internos suelen tener dependencias entre sí.
- Mensajes: Los mensajes son un tipo de información que los procesos de ROS se envían entre sí. La extensión es “.msg” e irán declarados en el manifiesto, además de estar incluidos en la carpeta “msg”.
- Servicios: Similar a los mensajes, pero utilizado cuando los procesos utilizan una comunicación del tipo petición/respuesta en el modelo de comunicación cliente/servidor. La extensión es “.srv” e irán dentro de la carpeta “srv”.
- Archivos de configuración: con extensión tipo “.yaml”, son archivos de un formato específicamente diseñado para almacenar y permitir la fácil carga de parámetros de configuración al inicio de los nodos.

Una explicación más detallada acerca de cómo crear, mediante comandos, esta estructura de archivos, se encuentra disponible en el **Anexo I (II)**.

2.6. Nivel de computación de ROS

La computación [14] en ROS se realiza en procesos llamados **nodos,** empleándose una red punto a punto; además, se basa en otros elementos (Figura 4):

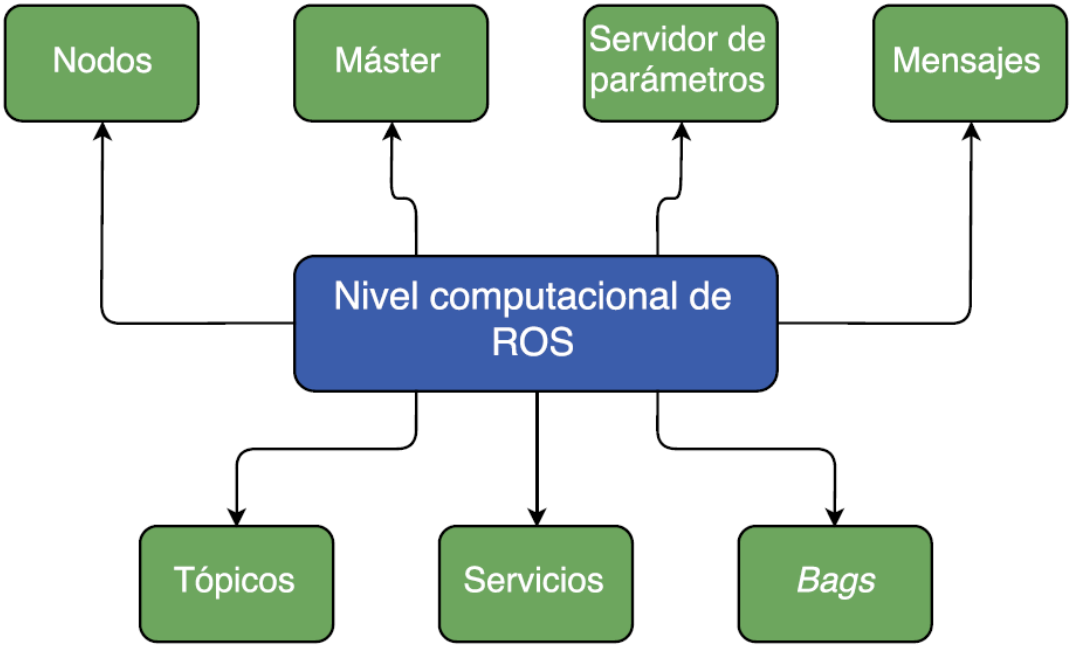


Figura 4: Nivel de computación de ROS

- Nodos: Son los procesos que realizan la computación. Pueden estar escritos en C++ o Python, utilizando librerías de ROS como *roscpp* y *rospy*, que se comunican mediante tópicos o servicios independientemente del lenguaje en el que estén programados. Esto permite atomizar los procesos complejos que ejecuta un robot en pequeñas tareas y poder reutilizar el código, con compatibilidad prácticamente total. Se pueden gestionar y depurar mediante el comando *rosnode*. Para ejecutar un nodo, se escribe la siguiente línea en la consola de comandos:

```
$ rosrun <nombre_del_paquete> <nombre_del_ejecutable>
```

  Si en su lugar se deben ejecutar varios nodos, además de parametrizarlos, se utilizan los archivos ".launch", que contienen descripciones en XML acerca de cómo se deben lanzar los paquetes y con qué parámetros; además, estos archivos lanzan el ROS Máster, en caso de que no esté presente:

```
$ roslaunch <nombre_del_paquete> <nombre_del_archivo.launch>
```

- Máster: El ROS Máster, también mencionado antes como *roscore*, gestiona la comunicación entre nodos, por lo que es obligatoria la existencia de uno para el funcionamiento del sistema. Aunque la ejecución de nodos se haga en ordenadores interconectados diferentes, sólo debe existir un ROS Máster, y en todos los ordenadores debe indicarse cuál es mediante su dirección IP

y su puerto; no obstante, existe la posibilidad, mediante integraciones externas, de tener [15]. Se ejecuta mediante el siguiente comando, adoptando normalmente el puerto *11311*:

```
$  roscore
```

- Servidor de parámetros: Permite guardar información en una localización central que pueden utilizar todos los nodos. Un ejemplo es la descripción URDF de cada uno de los robots. Es parte del ROS Máster.
- Mensajes: Los nodos se comunican entre sí utilizando mensajes de tipos prefijados, como por ejemplo un vector o una cadena de caracteres. Los tipos más básicos están incluidos, aunque se han creado muchos otros para tareas específicas en la robótica, por lo que se recomienda utilizarlos en lugar de crear nuevos tipos.
- Tópicos: Cada mensaje se transporta usando buses de comunicación llamados tópicos. Cuando un nodo publica un mensaje, se dice que un nodo publica un tópico; mientras que cuando lo recibe de otro, se subscribe a dicho tópico. Cada tópico tiene un solo tipo de mensaje, que puede ser de los tipos ya prefijados por ROS o personalizado, previa definición en la carpeta “msg” del paquete. El comando [16] para gestionar los tópicos es el siguiente:

```
$  rostopic <comando> <nombre_del_tópico>
```

- Servicios: mencionados en el apartado 2.5, se diferencian de los tópicos en que la comunicación es bidireccional.
- “Bags” o bolsas: Formato para guardar y reproducir datos de ROS, como aquellos provenientes de sensores. Al reproducir un *bag*, un nodo lo verá exactamente igual que si recibiese información de un sensor auténtico o simulado a través de un tópico:

```
$  rosbag record <tópico_1> <tópico_2>
$  rosbag play <nombre_del_bag>
```

Un ejemplo de cómo interactúan los diferentes elementos de ROS en tiempo de ejecución sería el siguiente:

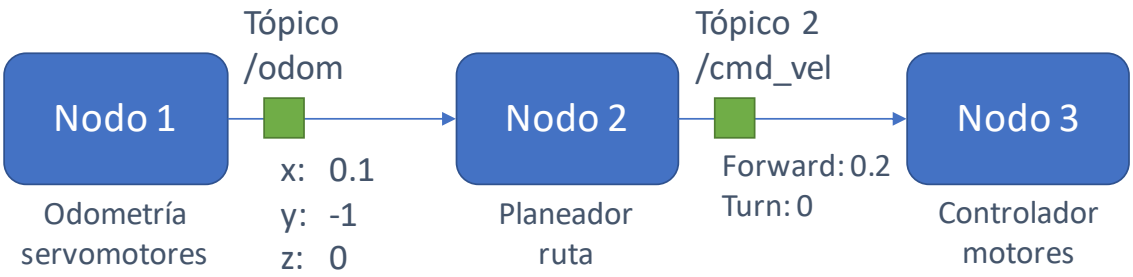


Figura 5: Ejemplo de nodos *publisher/subscriber*

En la Figura 5 se pueden observar tres nodos, que publican y/o se suscriben a diferentes tópicos, que incluyen los mensajes señalados en un momento concreto. El nodo 1 gestiona un sensor (encóder), publicando la odometría estimada respecto de las lecturas de dicho sensor, a través del tópico */odom*. El nodo 2 se suscribe a dicho tópico, procesa la información del mensaje y traza una ruta para la navegación autónoma, al tiempo que publica como salida los mensajes de velocidad que debe procesar y ejecutar el actuador mediante el tópico */cmd_vel*; para tal fin, el tercer nodo, que controla un actuador (servomotor), está suscrito a ese tópico, y se encarga de realizar las cinemáticas inversas y accionar los servomotores para alcanzar dicha velocidad.

3. Sensorización del entorno

La problemática a resolver en este proyecto implica el uso de sensores. Un **sensor** se define como **"todo aquel instrumento electrónico capaz de transformar las variaciones de una magnitud física medible en señales eléctricas, proporcionales a los cambios producidos en dicha magnitud"**. Por tanto, los sensores permiten obtener información acerca del entorno en el que se sitúan, que posteriormente puede ser digitalizada, procesada y utilizada para tomar decisiones en base a la misma. Como en todos los problemas de ingeniería, resulta conveniente segmentar este proceso en varios bloques independientes, que serán los siguientes:

- Detección de personas en escena.
- Ubicación de las mismas en el entorno.
- Identificación entre un grupo delimitado.

Para la correcta comprensión de la labor a realizar por cada sensor, se describe, a continuación, el funcionamiento, puesta en marcha y uso de cada uno de ellos:

3.1. Introducción a las cámaras de profundidad

Las cámaras de profundidad [17] o de tiempo de vuelo (a partir de ahora referidas como TOF, "Time of flight" en inglés) son dispositivos capaces de generar una imagen tridimensional del entorno, a partir de información proveniente de fuentes de luz modulada. El principio físico en el que se basan estos dispositivos es muy sencillo en esencia, aunque bastante complejo en cuanto a implementación; básicamente, una cámara de tiempo de vuelo consta de un *array* de píxeles CMOS sensibles a un espectro de frecuencia de luz determinado, que es el que la propia cámara emite de forma modulada; dicha luz, idealmente, es reflejada por la escena, y capturada por el sensor de la cámara, que mide entonces el desfase entre la señal luminosa modulada emitida y la recibida; con dicho desfase se puede saber el tiempo que la luz ha empleado desde su emisión por la cámara hasta la recepción en el sensor, y dicho tiempo, conociéndose la velocidad de la luz, permite conocer la distancia a la que se encuentra la porción de escena capturada en cada píxel. Dado que el tiempo medido es el que la luz tarde en llegar al objeto y ser reflejada, la fórmula (Ec. 1) que proporciona la distancia es:

$$D = \frac{c}{2f} \qquad (Ec.1)$$

En esta expresión, “c” es la velocidad de la luz (299792458 m/s), y “f”, la frecuencia de modulación de la luz emitida por la TOF; ya que ambas son constantes, “D” será la distancia máxima que se podrá medir con dicha cámara, hecho el cual, traducido a visión por computador, representará, típicamente, el valor más bajo en una imagen en formato de escala de grises (negro). Valores típicos de la frecuencia de modulación suelen ser 15 o 30 MHz, correspondiéndose estos con rangos de distancia de 0-10m, en el primer caso, y 0-5m en el segundo.

Figura 6: Escena (izquierda) y su correspondiente visualización en imagen de profundidad (derecha)

Las distancias medidas en los píxeles serán fracciones de esta distancia máxima; si la imagen es de 8 bits, como suele ser habitual, la imagen de profundidad deberá ser exclusivamente representada por valores enteros entre 0 y 255, lo cual proporciona, para el caso de D=10m,una resolución de 10/256=4cm (2cm en el caso de D=5m). Por tanto, se observa que, a mayor distancia medible por la TOF, la resolución disminuye, y se deberá alcanzar un compromiso en función de la aplicación.

Aunque teóricamente una sola medida por *frame* o toma sería suficiente para medir la distancia de cada píxel, en la práctica esto resultaría ser arriesgado, así como poco preciso. Por ello, lo más habitual es encontrar que las cámaras de tiempo de vuelo trabajan realizando varias mediciones de longitud para la misma toma, promediando después para calcular el valor final. Uno de esos métodos es el pulsante [18], que aparece ilustrado en la Figura 7. En el mismo, la fuente de luz ilumina la

escena por un breve periodo de tiempo Δt, y la energía reflejada es muestreada para cada pixel dos veces (C1 y C2), con el mismo intervalo de tiempo. Las cargas eléctricas acumuladas (Q1 y Q2) durante ese intervalo son medidas y empleadas para calcular la distancia (Ec. 2):

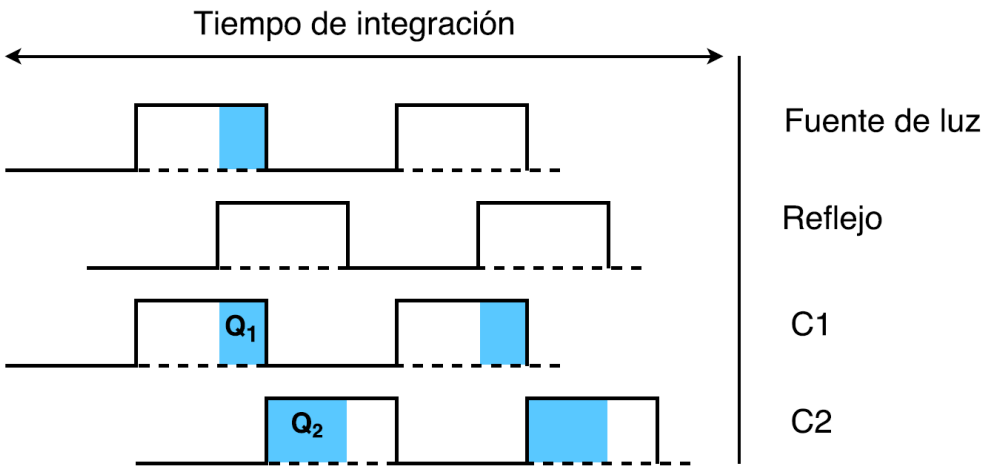


Figura 7: Cálculo gráfico del tiempo de integración [18] según la duración del proceso de medida de una distancia, a partir del viaje que realiza la luz

$$d = \frac{1}{2}\, c\, \Delta t \left(\frac{Q_2}{Q_1 + Q_2}\right) \qquad (Ec.2)$$

El otro método, llamado método de adquisición continuo, toma múltiples muestras por medida, cada una con un desfase de 90º, para un total de cuatro mediciones, a partir de las cuales y, utilizando el desfase, calcula la distancia (debido a que este método es más complejo, se insta al lector a comprobar su funcionamiento en [18]).

El tiempo que se emplea en este proceso se conoce como tiempo de integración (Figura 7), y es lógico afirmar que, mientras mayor sea el mismo, más confiable será la imagen tridimensional capturada del entorno, como puede apreciarse en la Figura 8, donde se ve que a mayor tiempo de integración el mapa de confianza toma valores más altos. Los valores más altos en el mapa de confianza serán aquellos más cercanos al uno probabilístico, que en escala de grises se traduce en el valor más alto del fondo de escala de la imagen; en este caso, dicho valor resulta ser 255, ya que las imágenes son de 8 bits. Sin embargo, un aumento excesivo del tiempo de integración provocará una bajada en la tasa de frames por segundo, lo cual resulta perjudicial para escenarios en movimiento. De nuevo, la aplicación para la cual se esté utilizando la TOF determinará el valor óptimo de este parámetro.

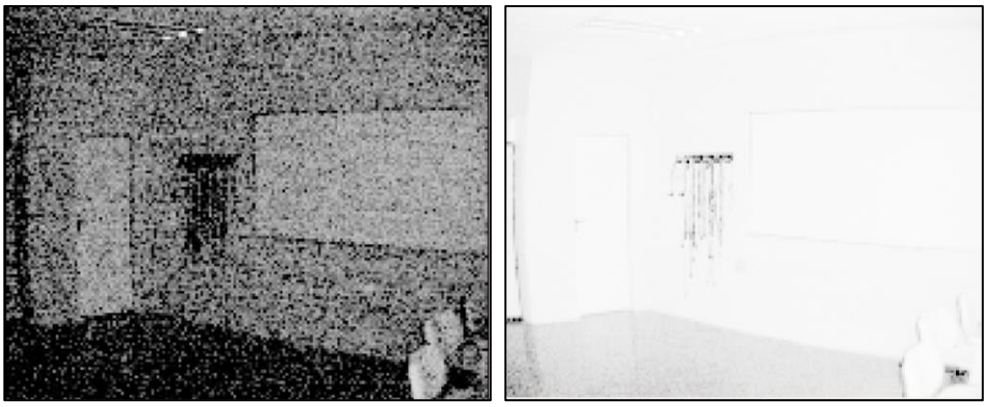


Figura 8: Mapa de confianza para el escenario del caso de estudio, con un tiempo de integración bajo (izquierda) y alto (derecha)

El resto de variables que influyen en el ajuste del mismo aparecen reflejadas en el esquema de la Figura 9, perteneciente a [19] (Anexo externo 1). Entre ellas, cabe destacar la presencia o no de objetos en movimiento, así como el índice de reflectividad, ya que la primera hace que si se quiere capturar dichos objetos sea necesario aumentar este tiempo, mientras que la segunda hace que si un objeto refleja excesiva luz haya que reducir el tiempo de integración para que la imagen no se vea saturada.

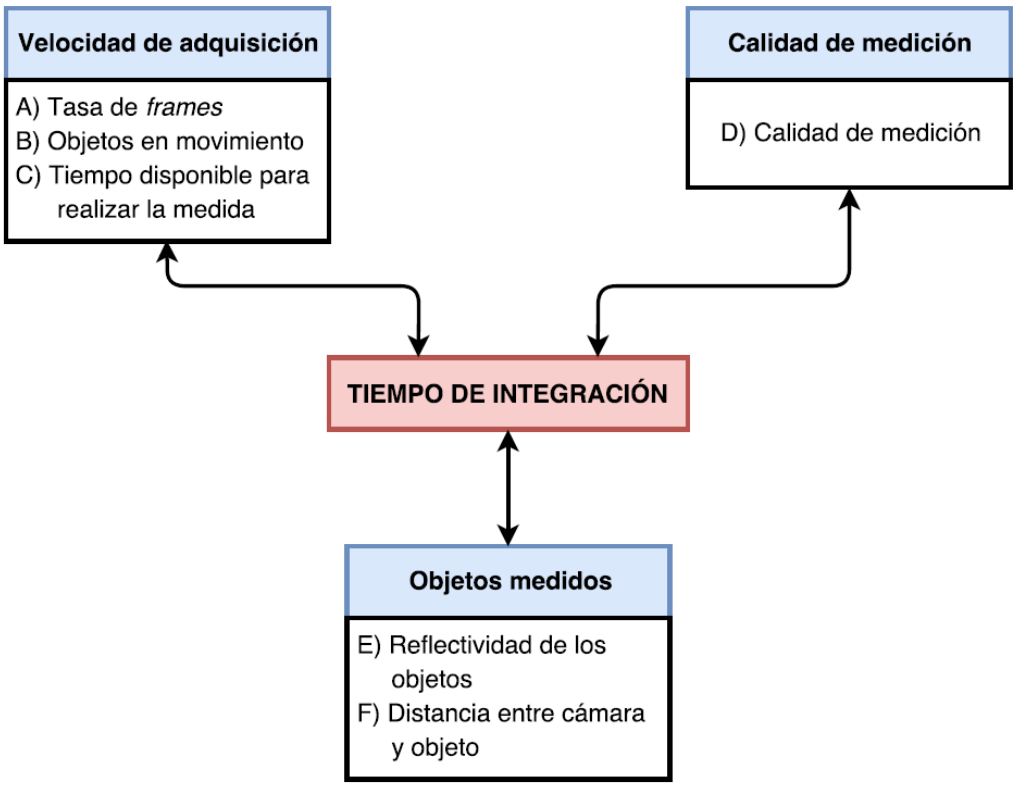


Figura 9: Variables que influyen en el tiempo de integración de una cámara TOF

Resulta también de gran interés conocer los parámetros intrínsecos y extrínsecos de cada sensor, puesto que gracias a ellos, y aplicándose las transformaciones pertinentes, es posible no sólo obtener información acerca de la distancia a la que cada píxel se encuentra del sensor, sino de las coordenadas cartesianas de los mismos respecto a un sistema de referencia solidario al sensor; esto permite reconstruir el entorno tridimensional en un computador, obteniéndose nubes de puntos (*pointcloud* en inglés, Figura 10) que resultan de gran utilidad y aplicabilidad en visión por computador, gracias especialmente a librerías específicamente diseñadas para trabajar con ellas, como la PCL [20] (“PointCloud Library”, en inglés).

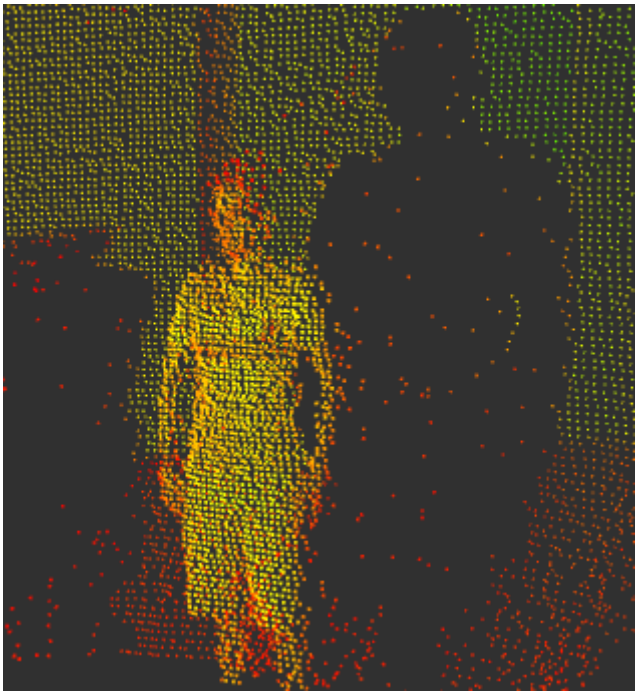


Figura 10: Persona, tal y como es representada en una *PointCloud*

Finalmente, es necesario destacar algunas desventajas y factores limitantes de estos dispositivos:

- Mala adecuación a entornos con alto índice de reflexión, que dificultan enormemente la generación de datos mapas de profundidad adecuados, debido precisamente al hecho de que impiden que la luz vuelva reflejada de forma correcta al sensor.
- No suelen ofrecer, generalmente, una resolución muy elevada, debido al exigente nivel de computación que requiere procesar la imagen, puesto que

cada píxel debe ser tratado de forma exclusiva e individual; no obstante, suele ser más que suficiente para la mayor parte de aplicaciones.

- Aunque teóricamente se podrían medir distancias desde 0 hasta el valor máximo, en la práctica existe un umbral mínimo de distancia, a partir del cual los valores proporcionados por el sensor no son verosímiles; esto se debe a las propias limitaciones físicas del sensor, y resulta de vital importancia tenerlo en cuenta para evitar mediciones erróneas. Por otra parte, distancias mayores a la distancia máxima, en ocasiones, pueden ser vistas como puntos mucho más cercanos (por ejemplo, un píxel situado a 14 metros de una cámara operando en el rango de los 10 metros, con un paso de cuantificación σ = 2mm entre medidas correlativas, podría verse a 4 metros, como se ve en la Figura 11 para el modelo concreto de cámara TOF empleado).

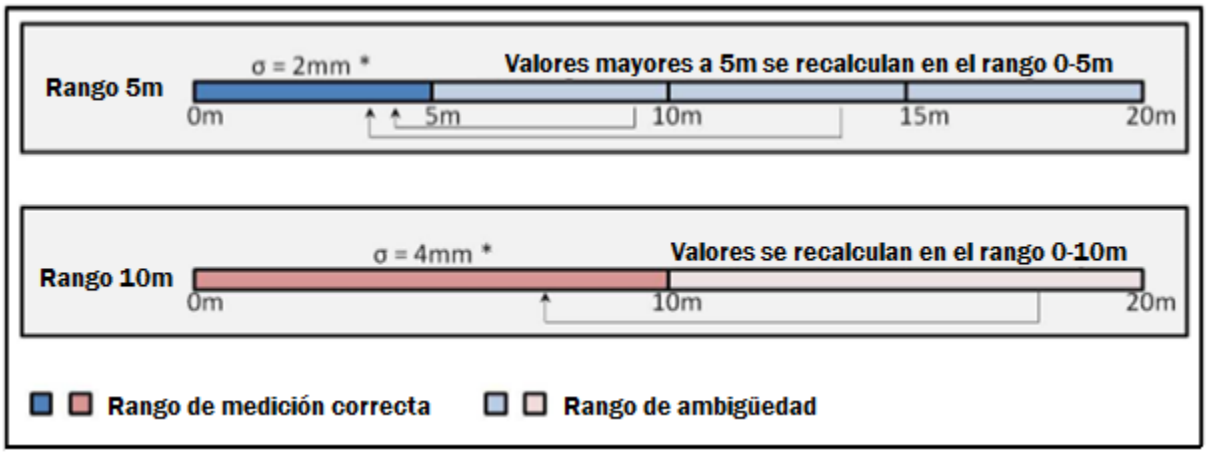


Figura 11: Rango de ambigüedad y de medición correcta para la Mesa SR4000, según el modo de operación (5 – 10m)

Es por todo ello que, generalmente, las cámaras TOF suelen emitir, además de imágenes de profundidad e intensidad (típica imagen de una cámara), mapas de confianza, que muestran hasta qué punto la distancia medida en cada píxel resulta ser un valor certero. Una representación de los tres tipos de imagen puede visualizarse en la Figura 12.

Figura 12: De izquierda a derecha, imágenes de profundidad, intensidad y mapa de confianza generadas por una TOF

3.2. Cámara TOF Mesa SR4000

Dentro de la variedad de dispositivos existentes en el mercado, se va a emplear la cámara SwissRanger Mesa SR4000 como encargada de monitorizar el espacio de trabajo en búsqueda de personas en el mismo. En los siguientes apartados, se describirá brevemente el *hardware* de la misma, así como las condiciones de trabajo óptimas y los datos de salida que ofrece, prosiguiéndose con una explicación sobre cómo se integra en la plataforma ROS para su pleno funcionamiento y usabilidad.

3.2.1. Hardware

La **SR4000** o SwissRanger 4000 de **Mesa Imaging** (Figura 13) es una cámara de tiempo de vuelo de emisión de luz activa, capaz de proporcionar **información tridimensional** acerca de entorno en el que opera.

Figura 13: Cámara de tiempo de vuelo SR4000, de Mesa Imaging

Algunos de sus parámetros, así como una representación gráfica de la misma, especificados completamente en su hoja de características (Anexo externo 2), son:

- Rango de detección/ Frecuencia de modulación: 0.3m a 5m / 30MHz
- Tensión nominal de alimentación: $12V_{DC}$
- Longitud de onda empleada: 850nm
- *Frame rate*: hasta 54FPS
- Precisión: +/-1cm
- Resolución: 176 x 144 píxeles

Externamente, la estructura física de la cámara se compone de un cuerpo metálico, que alberga, en su parte delantera, un sensor de array CMOS, encargado de recoger las ondas de luz reflejadas por el entorno, y emitidas por un conjunto de LEDs que lo rodean. En la parte trasera, se encuentran los conectores, tanto de alimentación como de transferencia de datos, que pueden ser USB o Ethernet, prefiriéndose como protocolo de transmisión este último, debido a su mayor velocidad.

Internamente, la Mesa SR4000 dispone tanto de circuitos de disparo, adquisición y procesamiento de datos, siendo destacable en este último bloque la inclusión de varias FPGA, dado el alto nivel de cálculo y procesamiento requerido.

3.2.2. Condiciones físicas de trabajo

El fabricante [19] recomienda montar la cámara en entornos libres de vibración y con una ventilación adecuada, que garanticen, en la medida de lo posible, medidas estables y precisas. Dentro de estas precauciones, es importante saber que, a mayores tiempos de integración, la Mesa SR4000 se calienta más.

Es imprescindible asegurar una ruta única de flujo de luz modulada (Figura 14), en línea recta desde la cámara al entorno, y de vuelta al sensor; solo de esta manera, se puede asegurar que las mediciones realizadas se corresponden con la realidad. Para ello, se debe evitar enfocar la cámara hacia escenarios cóncavos (como esquinas), así como las superficies reflectantes.

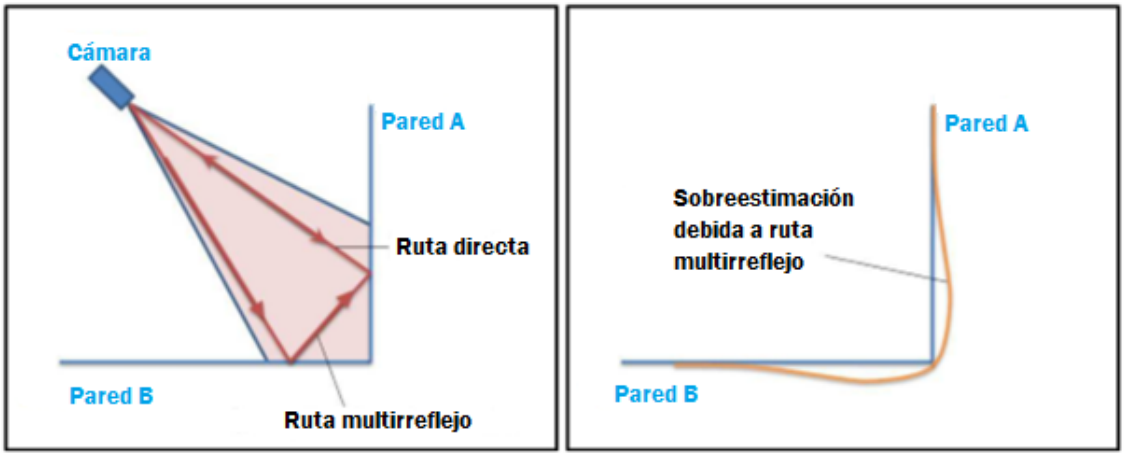


Figura 14: Sobreestimación de distancia debido a reflexiones indeseadas

Por último, como se citó en los inconvenientes comunes a todas las TOF, la Mesa SR4000 presenta un rango de ambigüedad (Figura 11), lo cual significa que puede considerar valores de distancia mayores a su máximo detectable como más cercanos; en la mayor parte de las aplicaciones, el propio entorno de trabajo (por ejemplo, un muro situado al fondo de la escena) es el encargado de solventar este problema.

3.2.3. Datos obtenidos

Para cada píxel, la cámara proporciona información acerca de la distancia de dicho píxel respecto a la misma e intensidad lumínica; además, se proporciona un mapa de confianza, que proporciona una estimación de la veracidad de los datos.

Además de estas tres imágenes, en formato de matrices de 176x144 píxeles, se pueden obtener las coordenadas XYZ de cualquier píxel respecto a un sistema de coordenadas solidario a la Mesa SR4000, y situado en el centro del sensor, tal y como se muestra en la Figura 15:

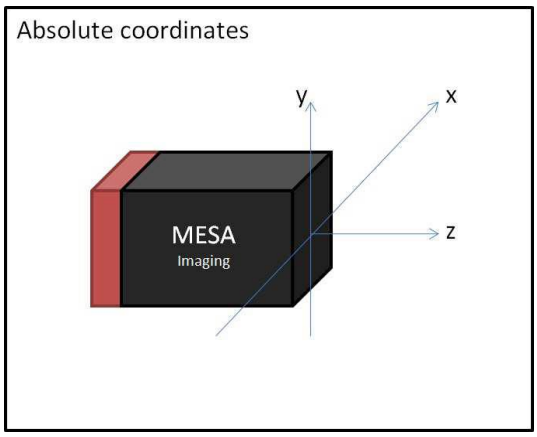


Figura 15: Sistema de coordenadas de la Mesa SR4000

También hay que destacar que la Mesa SR4000 puede trabajar según dos modos de adquisición diferentes: continuo y mediante disparo. En el primer caso, la cámara comenzará a tomar imágenes desde el momento de su conexión; mientras que, en el segundo, deberá esperar a recibir una señal del usuario para tomar un único fotograma. Dado que se pretende emplear la TOF para monitorizar un escenario parcialmente desconocido, lo ideal es que la cámara trabaje en modo de adquisición continuo, que es como viene configurada por defecto.

3.2.4. Integración y puesta en marcha en la plataforma ROS

Antes de comenzar a explicar el proceso de instalación e integración de la cámara Mesa SR4000 en ROS, conviene realizar una serie de observaciones:

- Se empleará un ordenador portátil con Ubuntu 14.04 LTS instalado en una partición del disco duro interno. El motivo de usar Ubuntu, como anteriormente se mencionó, es que ROS funciona mejor sobre sistemas operativos basados en Linux, además de que la gran parte de *drivers* disponibles en repositorios online para sensores y actuadores compatibles con ROS se encuentran escritos para Linux, por la propia filosofía de software libre de este sistema operativo. La versión de Ubuntu seleccionada (14.04) corresponde a una versión *LTS* o *Long Term Supported,* lo cual significa que goza de mayor estabilidad y es menos vulnerable a fallos que otras versiones más recientes.
- Aunque Ubuntu, y por lo tanto ROS, pueden ejecutarse en una máquina virtual en Windows, gracias al software de virtualización ***VirtualBox***, se va a **prescindir de su uso** en el presente trabajo, debido a que los sensores empleados (principalmente cámaras) proporcionan información en forma de imágenes que requieren de cierta capacidad de procesamiento, incapaz de ser proporcionada de forma óptima por un sistema virtualizado.
- La versión de ROS elegida para desarrollar la aplicación es **ROS Indigo**, debido a que, aunque no es la versión más reciente, es muy estable, y totalmente compatible con versiones anteriores de ROS; así, se aboga por la total estabilidad y compatibilidad entre ROS, los diferentes sensores utilizados en el presente proyecto y los robots empleados en [1].

Para efectuar una primera toma de contacto, se puede instalar el software demostrativo LibMesaSRTester en Windows, disponible a través del CD de instalación de los *drivers* de la TOF que viene incluido en su embalaje. Con dicho *software*, pueden visualizarse los datos de salida generados por la cámara, en forma de imágenes y de nubes de puntos.

Más interesante para este trabajo resulta la utilización del paquete correspondiente para Ubuntu, que permite hacer saber que la instalación de los *drivers* en este sistema ha resultado exitosa y que la TOF está lista para ser integrada en ROS. Para probarlo, una vez instalado el archivo Debian con los *drivers*, disponible en la página oficial [21], se abre una terminal y se ejecuta el siguiente comando:

```
$ libMesaSRTester
```

Este programa mostrará en terminal una serie de opciones, entre las que se encuentran probar el conexionado de la cámara al PC o “visualizar” los datos de salida en bruto, lo cual, si la cámara está bien conectada, aparecerá en terminal como una gran cantidad de datos en hexadecimal; de esta forma, queda comprobado que la cámara Mesa SR se encuentra lista para su integración en ROS.

Una vez apuntado esto, se procede a explicar el proceso de instalación e integración de la TOF en ROS Indigo. Los *drivers* necesarios para el correcto funcionamiento de la misma deberán descargarse de la página del fabricante [21], mientras que el firmware que permite abrir, acceder a la información publicada por la cámara y modificar sus parámetros, se obtendrá gracias a la integración realizada en el proyecto [22] ***“Open_PTrack”***.

Para instalar los drivers de la Mesa SR4000, una vez en la página del fabricante, se debe descargar la versión correspondiente. Después, la instalación resulta tan sencilla como hacer doble click.

Para efectuar la instalación de la plataforma “Open_Ptrack” en ROS, teniéndose en cuenta que sólo se pretenden instalar los drivers de la Mesa SR4000, se deberán realizar varios pasos, resumidos en el **Anexo II**. A pesar de que dicho anexo lo menciona, es importante indicar que si se desea trabajar con el caso de estudio

explicado en el apartado 5 de la presente memoria se debe **modificar** el paquete "swissranger_camera" por la versión presente en el CD adjunto.

Ya que la cámara que se va a emplear utiliza el Ethernet como protocolo de conexión y comunicación, es conveniente disponer de una red con conexión a Internet mediante protocolo DHCP, que ayude a fijar la dirección IP de la cámara de forma automática; en caso de no ser así, deberá ser introducida manualmente (se puede consultar el proceso de asignación de IPs en [19]). Así, según el caso:

- Si se dispone de red **DHCP**, la Mesa SR4000 deberá, **sin estar enchufada a alimentación**, conectarse mediante Ethernet' a uno de los puertos libres del *router* y, posteriormente, conectarse a la red eléctrica, mediante el cargador suministrado; de esta forma, la cámara, por sí sola, obtendrá una dirección IP, con la que se podrá arrancar e inicializar desde ROS.
- Si, por el contrario, no es posible utilizar el método de asignación automática por DHCP, bien porque no se disponga de conexión activa a Internet o porque la red no utilice este protocolo, se deberá asignar la IP manualmente. Para ello, los pasos a seguir son los siguientes:

    - Instalar, desde una terminal, el programa "Telnet":

    ```
    $ sudo apt-get install telnet
    ```

    - Alimentar la cámara **sin** conectar el cable Ethernet. Tras veinte segundos, conectar dicho cable al puerto del ordenador. En este momento, la Mesa SR4000 tiene, por defecto, asignada la IP estática 192.168.1.42, que deberá ser cambiada.
    - Ejecutar el siguiente comando para entrar al menú de configuración de la cámara:

    ```
    $ telnet 192.168.1.42
    ```

    - Establecer la nueva IP de la cámara:

    ```
    $ fw_setenv staticip 192.168.X.XX
    ```

    - Cerrar la terminal y desconectar y conectar la alimentación de la cámara mientras el cable Ethernet sigue conectado. Ahora, la cámara

tendrá asignada la IP que se ha especificado, y podrá iniciarse en ROS.

Para poder lanzar la cámara, primero debe especificarse cuál es la IP en el archivo de lanzamiento (*.launch*). Para ello, en una nueva terminal, navegar hasta la carpeta de lanzamiento:

```
$ cd ~/<nombre_workspace>/src/open_ptrack/src/swissranger_camera/launch
$ gedit sr_eth.launch
```

Este último comando abrirá un editor de texto con el archivo de lanzamiento en formato XML; en el apartado “IP”, sustituir la que aparece por defecto por la nueva IP; en el caso de la asignación manual, dicha dirección será la especificada anteriormente en “telnet”; si se ha realizado la asignación de IP mediante DHCP, para averiguarla, ejecutar:

```
$ sudo arp-scan -l –interface=eth0
```

De todos los dispositivos conectados, el que aparezca con un identificador llamado “Mesa Imaging” es el correspondiente a la cámara. Copiar la IP y sustituirla en el archivo de lanzamiento.

Finalmente, tan solo queda lanzar la cámara:

```
$ roslaunch swissranger_camera sr_eth.launch
```

Aunque no se recomienda, debido a la mayor lentitud de la conexión, también puede utilizarse una de las versiones USB de la Mesa SR4000; en este caso, el archivo de lanzamiento que debería ejecutarse sería el “sr_usb.launch”, localizado en el mismo paquete que el anterior.

Una vez lanzada correctamente la cámara, Rviz permite visualizar los datos de salida que proporciona la misma en ROS. Para ello, en el caso de las imágenes, resulta tan sencillo como seleccionar los tópicos “SwissRanger/distance/image_raw” para el caso de la imagen de profundidad; “SwissRanger/intensity/image_raw” para la imagen de intensidad, y “SwissRanger/confidence/image_raw” para el mapa de confianza. Una visualización de estas tres imágenes simultánea puede verse en la Figura 12.

En caso de quererse visualizar la **nube de puntos** (PointCloud en ROS), es importante saber que un PointCloud debe siempre tener un ***frame* de referencia**, que en este caso debe llamarse **SwissRanger**. Para solventar la inexistencia de dicho *frame*, se puede utilizar la herramienta “static_transform_publisher”, del paquete TF [27].

```
$ rosrun tf static_transform_publisher 0 0 0 0 0 0 1 world SwissRanger 10
```

El resultado se puede consultar en la Figura 16, donde aparece una nube de puntos de un laboratorio.

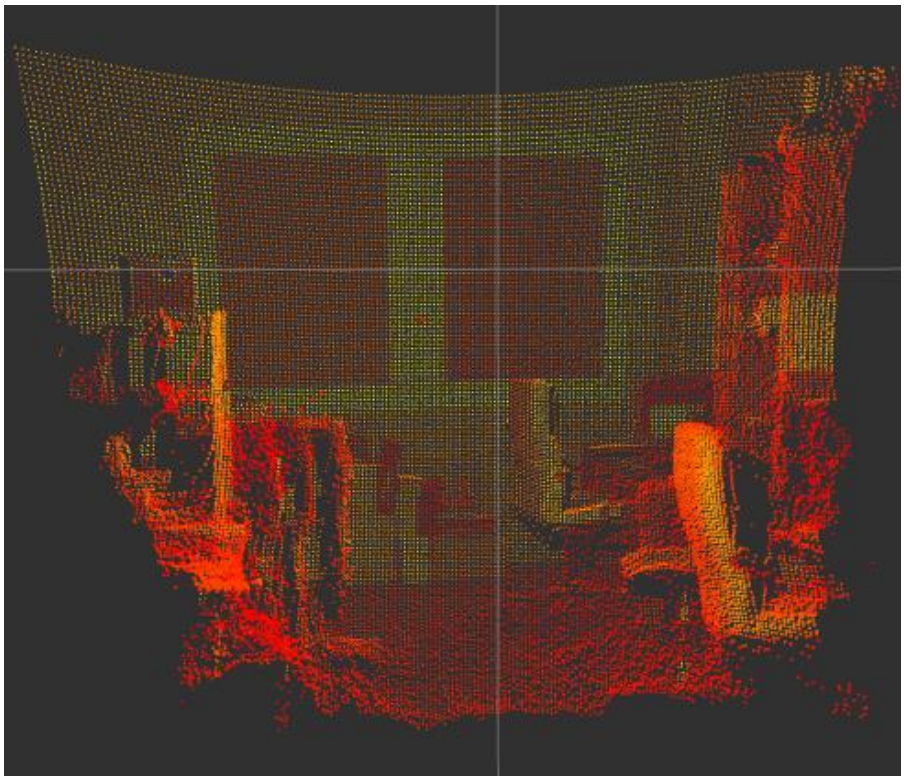


Figura 16: Ejemplo de una *PointCloud*

3.2.5. Información adicional: empleo de varias cámaras en escena

A pesar de que en el presente trabajo no se requiere del uso de varias cámaras de tiempo de vuelo para conseguir suficiente información para monitorizar el entorno parcialmente desconocido objeto de estudio, es conveniente realizar una serie de aclaraciones respecto a este tema, de forma que, si en el futuro fuese necesario, se establezcan aquí unas pautas mínimas para la adecuada consecución de dicho objetivo.

Según los artículos [23] y [24], dos cámaras de tiempo de vuelo podrían emplearse simultáneamente en una misma escena, sin provocar interferencias en su funcionamiento, ajustando la frecuencia de onda a la que trabaja cada una; esto es posible en las cámaras Mesa SwissRanger, por lo que se procederá a efectuar una prueba para comprobar lo anteriormente expuesto. Para el desarrollo de las mismas, se empleará el software SwissRanger 3D Viewer en Windows, que proporciona una sencilla interfaz de visualización de los datos generados por las cámaras de tiempo de vuelo Mesa. Se van a emplear dos modelos de cámara diferentes, el SR4000 (que se está utilizando para el desarrollo del presente trabajo) y el SR4500 (modelo sucesor del 4000, que incorpora algunas características adicionales). Una de las cámaras se conectará mediante USB, encargándose el sistema operativo de encontrar los drivers adecuados para su puesta a punto y posterior uso; mientras que para la otra, operable mediante conexión Ethernet, deberán seguirse los pasos de conexionado anteriormente descritos; en este caso, cada cámara se probará en un ordenador diferente, para evitar incompatibilidades, así como para poder visualizar los datos de ambas de forma simultánea.

Una vez conectadas ambas cámaras, en la aplicación SwissRanger 3D Viewer y tras seleccionarse cada una de ellas, se pulsa en “Start”; desde ese momento, se comenzarán a adquirir imágenes de profundidad. Para poder comprobar si ambas cámaras pueden operarse de forma simultánea en la misma escena, hay una serie de parámetros que deberán variarse en varias tomas:

- Frecuencia de trabajo, es decir, al frecuencia de modulación de onda luminosa emitida por cada cámara con el fin de que rebote en los objetos y pueda así calcularse la distancia a la que se encuentran. Teóricamente, esta frecuencia debe ser **diferente** [24] en cada cámara, a fin de evitar la aparición de interferencias.
- Tiempo de integración: a pesar de que, mientras mayor es el tiempo de integración, mayor es la precisión y menor es el ruido, también es cierto que se pierde velocidad de respuesta (no pueden detectarse fácilmente objetos en movimiento); este tiempo deberá ser aumentado, por tanto, si se pretenden medir distancias lejanas, en escenas preferiblemente estáticas.

A continuación, se muestran algunas imágenes captadas por ambas cámaras, tanto funcionando de forma independiente como simultánea. En la Figura 17, se observa una representación gráfica de los datos de salida obtenidos por la cámara, tanto en forma de nube de puntos como de imágenes 2D. La escena en cuestión es la de una mesa con varios objetos y una silla en el centro, y se observan en el margen izquierdo tres imágenes: profundidad (arriba), intensidad luminosa (centro) y mapa de confianza (abajo); es esta última la que resulta ser más interesante, puesto que muestra que, con un tiempo de integración bajo, las zonas más lejanas de la imagen aparecen representadas en rangos poco confiables (zonas oscuras del mapa), mientras que las más cercanas están fidedignamente representadas (en tonos verdosos). El rango de funcionamiento es de 0-5m, empleándose para ello frecuencias luminosas de modulación para el sensor de la TOF en el margen de los 29-31MHz.

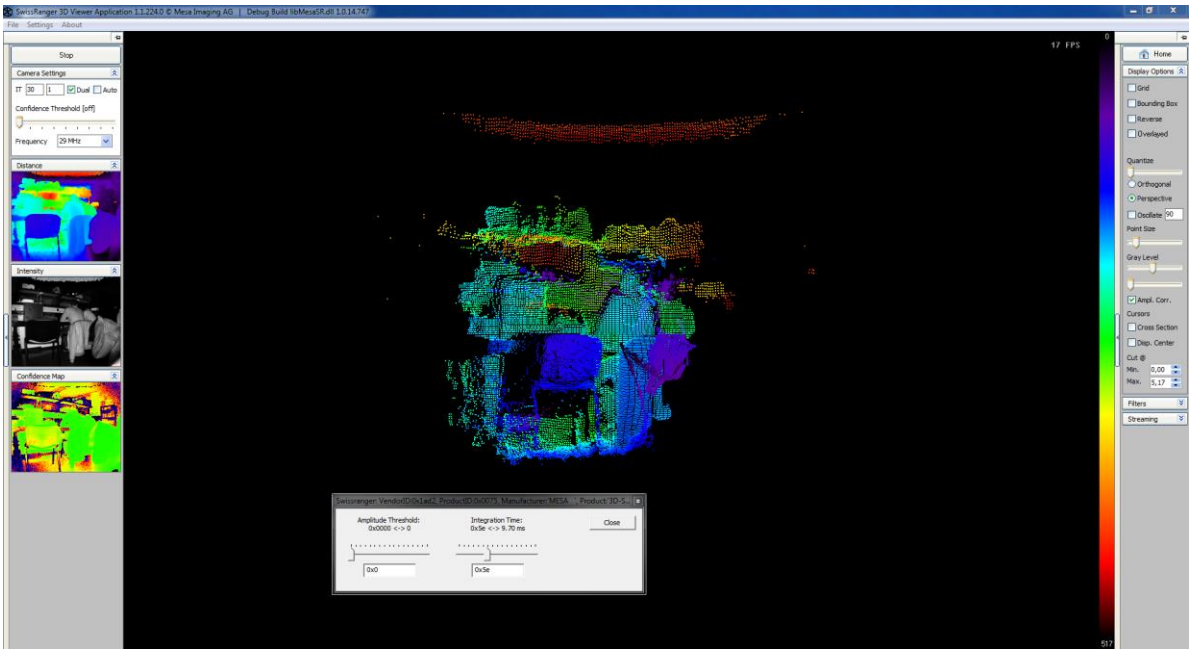


Figura 17: Cámara TOF Mesa SR4000 funcionando de forma aislada, con un tiempo de integración bajo

Si, en la misma escena, se aumenta el tiempo de integración (Figura 18), el resultado es que la nube de puntos resulta ser similar, pero el mapa de confianza proporciona unos rangos de mucha mayor fiabilidad, eso sí, a costa de saturar excesivamente la imagen de intensidad y de provocar que los objetos en movimiento aparezcan de forma difusa. Esto demuestra que debe llegarse a un compromiso a la hora de ajustar este parámetro; típicamente, se dejará de cargo de la propia cámara la tarea de hacerlo, puesto que dispone de un modo de regulación automática.

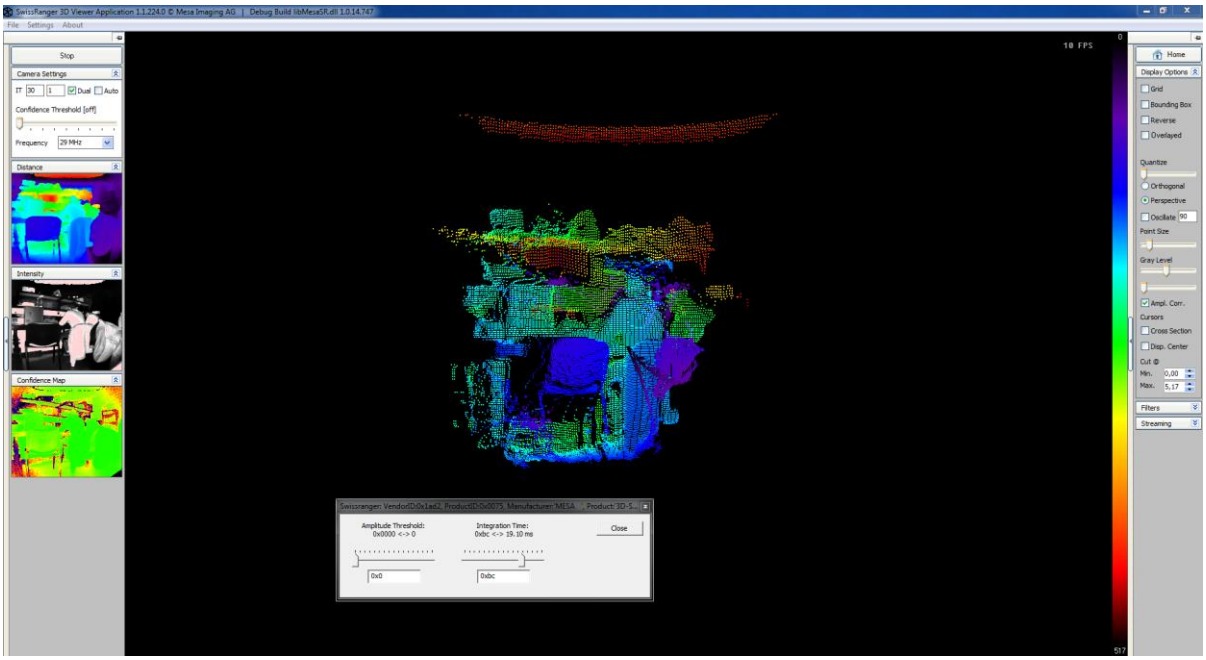


Figura 18: Cámara TOF Mesa SR4000 funcionando de forma aislada, con un tiempo de integración alto

Antes de proceder a efectuar una prueba de dos cámaras TOF operando de forma simultánea, se comprobará experimentalmente la relación entre frecuencia de modulación y distancia máxima que la cámara puede detectar en una escena determinada. Para ello, se cambia la frecuencia de modulación al rango de los 14.5-15.5 MHz, observándose que ahora el rango de detección pasa a ser de 0 a 10 metros. Por tanto, menores frecuencias de modulación proporcionan mayores rangos de medida.

Finalmente, se realizan capturas de la misma escena por dos cámaras, una Mesa SR4000 y otra Mesa SR4500. En la Figura 19, se observan dos imágenes tomadas por el primero de los sensores, operando en el rango de los 14.5 MHz; a la izquierda, aparece funcionando de forma independiente, mientras que a la derecha se puede visualizar una captura realizada mientras la Mesa SR4500 estaba también en funcionamiento.

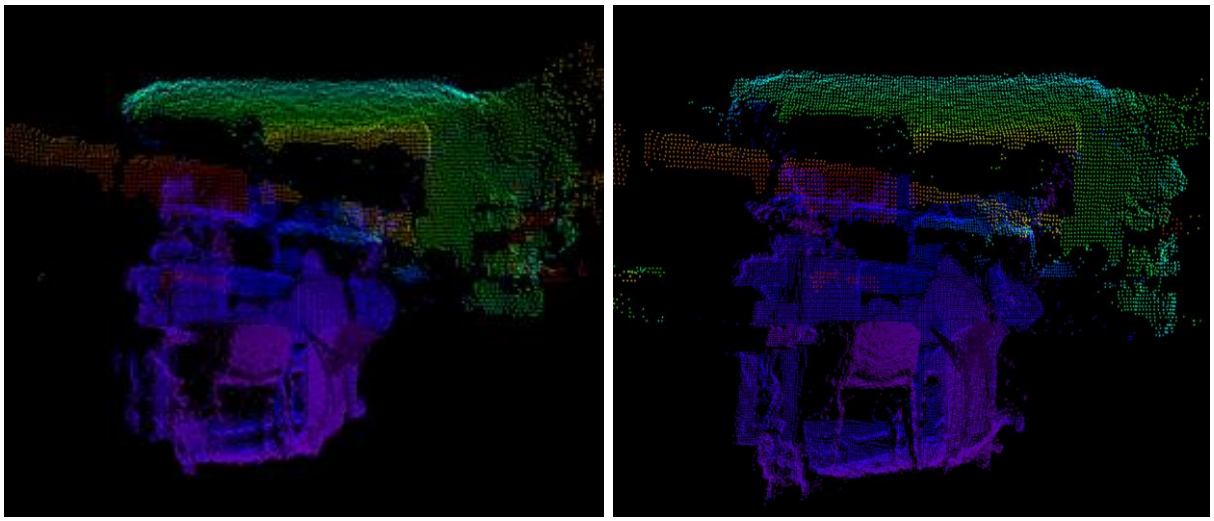


Figura 19: Mesa SR4000 Funcionando de forma independiente (izquierda) y simultánea (derecha)

La Mesa SR4500, por su parte, y según lo establecido en el artículo académico [23], se fija a un valor diferente, 15.5 MHz en este caso. La disposición de imágenes en la Figura 20 es idéntica a la Figura 19, mostrando las dos situaciones de funcionamiento:

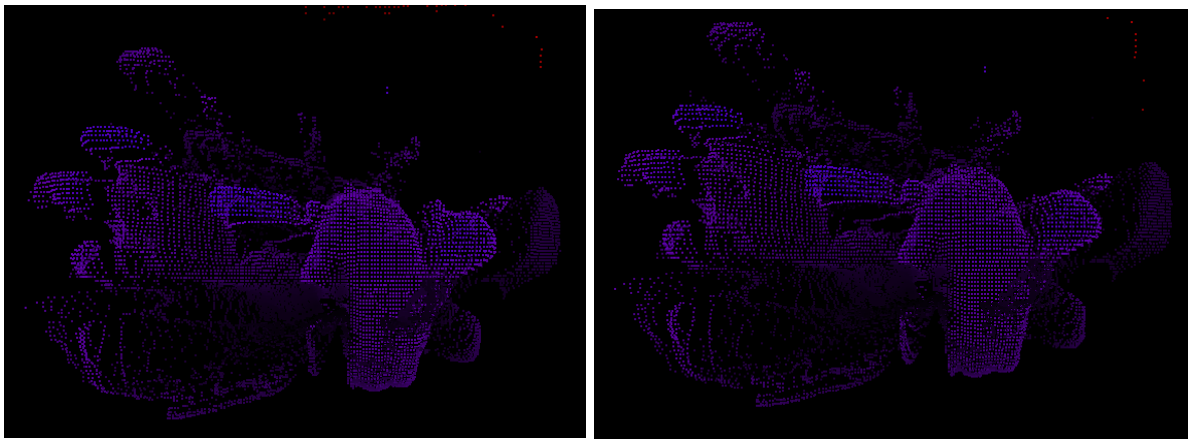


Figura 20: Mesa SR4500 Funcionando de forma independiente (izquierda) y simultánea (derecha)

Como puede observarse, las nubes de puntos generadas por cada cámara resultan ser prácticamente idénticas tanto cuando funcionan de forma aislada en la escena como cuando lo hacen en presencia de otra TOF, con lo **cual se puede concluir que la presencia de ambas funcionando simultáneamente en la misma escena no afecta notoriamente al rendimiento de cada una de las cámaras por separado**. No obstante, se deberá tener cuidado con superficies especialmente reflectantes, ya que siempre afecta al rendimiento de estos sensores.

3.3. Cámara Kinect V1

Acto seguido, se pasa a describir y contextualizar el empleo de la cámara Kinect V1 de Microsoft en el presente proyecto, como elemento encargado de la identificación de personas.

3.3.1. Descripción y hardware

Originalmente, Kinect fue (y sigue siendo) un elemento desarrollado por Microsoft como complemento a la videoconsola Xbox, que permite a los usuarios interactuar con la consola y jugar sin necesidad de mantener un contacto físico, a diferencia de otros controladores tradicionales; esto se consigue gracias a varios sensores de visión por computador que reconocen la escena, objetos, partes del cuerpo… así como otros elementos adicionales, como comandos de voz.

Más específicamente hablando, este sensor contiene los siguientes elementos [25], tal y como se muestran en la Figura 21:

- Una cámara RGB (en color), que obtiene imágenes en resolución de 1280x960 píxeles.
- Un sensor infrarrojo (830nm), compuesto de emisor y receptor, que tiene un funcionamiento similar al explicado para las cámaras de tiempo de vuelo: el emisor produce ondas de luz infrarrojas, que son reflejadas de vuelta por los objetos y capturadas por el sensor; entonces, con ayuda de circuitería adicional, las ondas capturadas se comparan con las emitidas y se obtiene una medida de distancia en profundidad, que hace posible generar una **imagen de profundidad o tridimensional** del entorno.
- Micrófono multiarray, para capturar sonido desde diferentes ángulos.
- Acelerómetro de tres ejes, que ayuda a obtener la orientación de la Kinect.

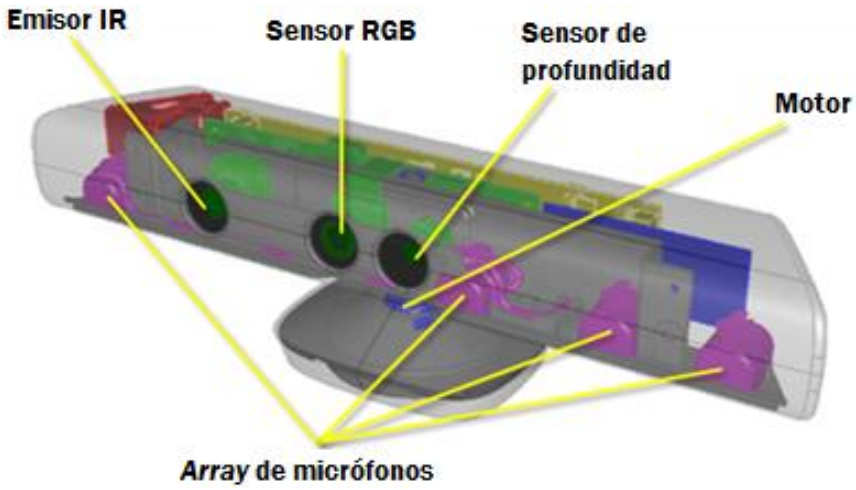


Figura 21: Partes del sensor Kinect

Además, la Kinect V1 dispone de actuadores motorizados situados en su parte inferior, que permiten que la cámara disponga de una mayor libertad a la hora de moverse, para monitorizar y seguir un elemento concreto en escena.

3.3.2. Razón de uso en el presente trabajo

A pesar de que los sensores Kinect fueron originalmente concebidos para su uso en plataformas de entretenimiento, se puede afirmar que, en los últimos años, han experimentado un gran auge como consecuencia de su empleo en aplicaciones de investigación y desarrollo relacionadas con el mundo de la automática, robótica y visión por computador; concretamente, debido a su capacidad para generar imágenes "RGB-D" (RGB-*Depth*), gracias al uso combinado de sus dos sensores de visión. Así, la plataforma Kinect proporciona una herramienta para conocer el entorno de fácil uso, medianamente confiable y, sobre todo**, asequible económicamente**, en comparación con otros sensores de captación de información tridimensional del espacio. Por todo ello, y especialmente en proyectos de desarrollo y prototipado, la Kinect se utiliza como la mejor opción de bajo coste para desarrollar sistemas de navegación independiente, ya que la imagen RGB-D generada por el sensor permite la creación de nubes de puntos, útiles para su tratamiento con la "PCL" (PointCloud Library) que, en última instancia, simulan a los láseres empleados en sistemas convencionales de navegación autónoma.

Además, y debido al carácter lúdico para el que fue concebido, Kinect de Microsoft dispone de una serie de librerías, desarrolladas por la propia compañía, que permiten realizar complejas y variadas tareas, enfocándose en la interacción con

personas (originalmente, el propósito de esta cámara era el de “trackear” a dichas personas en escena para incluirlas en el entorno virtual de juego); estas librerías son las que resultarán de mayor interés para este proyecto, puesto que permitirán conocer características morfológicas de las personas encontradas en escena, que servirán de ayuda para su posterior identificación.

3.3.3. Integración y puesta en marcha en la plataforma ROS

Para la inclusión del sensor Kinect en la plafatorma ROS, se necesitará instalar:

- Drivers que permiten el acceso a los controladores de la cámara, así como la información proporcionada por sus sensores.
- Framework **OpenNI** [26] (Open Natural Interaction), que proporciona herramientas *open-source* para una interacción lo más natural posible entre dispositivos y personas; además, se requerirá la instalación del software **NITE,** que permite acceder a funcionalidades como el seguimiento del “esqueleto” (estructura morfológica) de personas en tiempo real. Es importante destacar que la versión empleada requiere de una **calibración** previa de la persona a *trackear*, que se llevará a cabo situándose en la conocida como “Psi Pose”, en referencia a la letra griega **Ψ;** una descripción gráfica de este proceso puede visualizarse en la Figura 22:

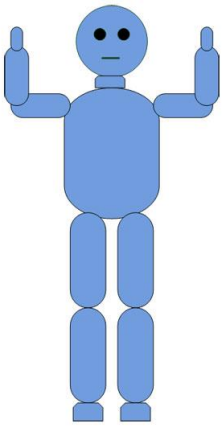


Figura 22: “Psi Pose”, necesaria para calibrar a la persona y poder realizar un seguimiento de su esqueleto en tiempo real

- Paquetes de ROS que se encarguen específicamente de convertir la información obtenida en los dos puntos anteriores y transformarla a mensajes

entendibles por robots y compartibles en ROS, tales como transformadas espaciales, imágenes a color, nubes de puntos…

Los comandos necesarios para instalar todo lo anteriormente expuesto y poder emplear la cámara Kinect en ROS se encuentran especificados en el **Anexo III**. Una vez ejecutados, se dispondrá de varios paquetes que permitirán la inicialización y uso del sensor, entre los cuales cabe destacar:

- Openni_launch: se encarga de gestionar el arranque, inicialización y puesta en marcha del sensor Kinect, previamente alimentado y conectado al USB, para permitir su transferencia de datos con el ordenador. Además, este paquete proporciona herramientas para la conversión de los datos en bruto obtenidos por el sensor a mensajes compatibles con ROS. La forma de invocarlo, habitualmente, suele ser a través del siguiente comando:

```
$ roslaunch openni_launch openni.launch
```

- Openni_tracker: este paquete utiliza las herramientas proporcionadas por el *framework* OpenNI para realizar un seguimiento de la estructura morfológica o esqueleto de las personas en escena, proporcionando como datos de salida transformadas espaciales o TF [27] (consultar más información sobre TF en **Anexo I (III)**), que indican la posición relativa de las principales articulaciones de la persona respecto de un punto de referencia fijo, que es la propia cámara. Esta esqueletización es realizada por la cámara utilizando herramientas de visión por computador, con las cuales localiza el cuerpo de la persona y, una vez lo calibra (Figura 22), ubica las diferentes extremidades, representándolas mediante segmentos; así, las intersecciones de los segmentos representarán las diferentes articulaciones, como puede verse en la Figura 23, que reconstruyen completa y virtualmente la estructura de un cuerpo humano en cabeza, tronco y extremidades, respetando las proporciones y medidas de la persona real. Esta herramienta se empleará, en un futuro, como base para desarrollar una aplicación que permita identificar a la persona que esté presente en una escena. La forma de ejecutarla es:

```
$ rosrun openni_tracker openni_tracker
```

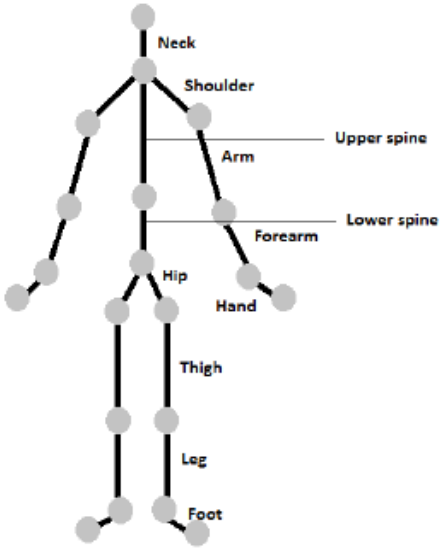


Figura 23: Esqueletización de una persona, según Kinect

3.4. Sensor de medición inercial *IMU 9DOF*

Para completar la descripción del conjunto de sensores que van a formar parte de la plataforma encargada de obtener información sobre el entorno objeto de estudio, que debe recordarse es **parcialmente desconocido**, se procede ahora a tratar en este apartado la unidad de medición inercial Razor IMU (a partir de ahora *IMU)* de SparkFun Electronics. El papel principal de este sensor en la aplicación final será el de dotar a la cámara que permanece fija en la escena (la cámara de tiempo de vuelo, encargada de detectar la presencia de humanos en la misma) de un sistema de referencia; de esta forma, la cámara podrá ser localizada en un mapa y utilizada como punto de referencia para situar respecto a ella los elementos detectados, es decir, las personas presentes en el entorno de trabajo. Concretamente, el IMU cumplirá con dos tareas:

- Servir de elemento auxiliar en la localización de la cámara de tiempo de vuelo en el mapa del escenario objeto de trabajo. Dicho mapa será generado y proporcionado como parámetro de entrada por el robot móvil PMB-2, que dispone de un potente láser capaz de generarlo por sí mismo; para esta tarea, se empleará el algoritmo AMCL (*Adaptive Monte Carlo Localization*), que se describe con mayor detalle en el apartado 4.3.2.
- Generar un sistema de referencia triaxial estático, con origen en la lente de la propia cámara y con orientación invariante, que pueda ser utilizado como

origen de las coordenadas de las personas detectadas, teniéndose en cuenta que las coordenadas generadas por la cámara TOF Mesa SR4000 responden al triedro presente en la Figura 15; dicho sistema de referencia, adecuadamente referenciado en el mapa, será capaz de expresar las coordenadas de la personas de forma automática respecto al sistema de referencia global absoluto, gracias a la herramienta **TF** [27].

3.4.1. Hardware

La placa Razor IMU 9DOF (*9 Degrees Of Freedom* o “9 grados de libertad” en español), modelo SEN-10736 de SparkFun Electronics consta de tres sensores de tres ejes cada uno (acelerómetro, giroscopio y brújula), que proporcionan tres grados de libertad cada uno. Los datos de salida de estos sensores son procesados por un microprocesador ATMega, y enviados a través de puerto serie. Una visión general del aspecto del sensor, en la que pueden observarse tanto el microprocesador como los sensores, además de circuitería adicional, sería la presente en la Figura 24:

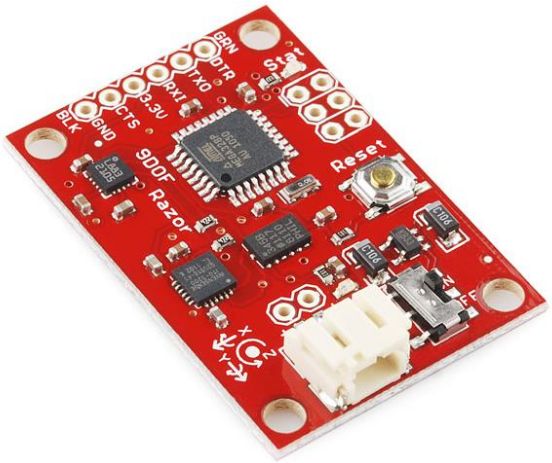


Figura 24: Sensor Razor IMU 9DOF

Para una información más detallada acerca de los diferentes circuitos integrados que intengran los sensores presentes en el dispositivo, así como el esquemático de la placa que integra la Figura 24, consultar los Anexos externos 3 y 4 respectivamente, adjuntos a la presente memoria de proyecto.

El IMU funciona a niveles de tensión CMOS (3.3V), por lo que para que sea posible la comunicación entre el mismo y un computador, que será el elemento receptor más habitual de la información generada por el dispositivo sensor, deberá disponerse de un conector conversor de niveles de tensión CMOS a TTL (5V);

habitualmente, un cable FTDI (nombre comercial para dispositivos conversores de niveles de tensión) será más que suficiente para acometer esta tarea. Resulta, por tanto, fundamental saber cómo debe realizarse la **conexión entre el IMU y el PC**, puesto que un error podría ser fatal y destruir los sensores; esto viene reflejado en la Figura 25, perteneciente al manual de uso del dispositivo FTDI empleado en este proyecto, que es el cable TTL-232R:

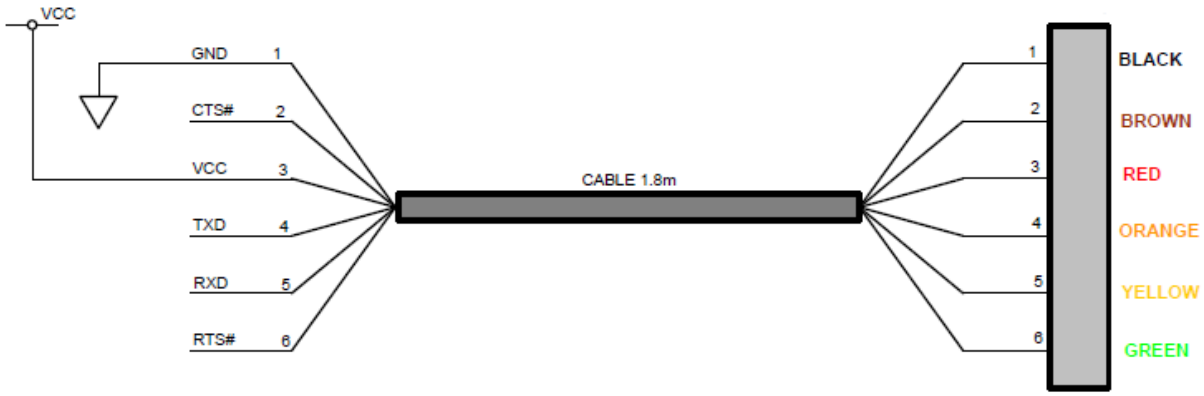


Figura 25: Diagrama de conexión entre el cable FTDI y el IMU

Si se observa la Figura 25, se puede apreciar cómo cada conector presente en el IMU tiene un correspondiente color, con lo cual resulta sencillo realizar la conexión del IMU al computador de forma correcta.

3.4.2. Instalación del software específico: Arduino / ROS

La instalación del *software* necesario para la integración de este sensor en la plataforma ROS resulta relativamente sencilla, ya que, gracias al carácter *open-source* de la plataforma, se dispone de *drivers* que facilitan la extracción de datos de los sensores a través del puerto serie y la conversión de estos a mensajes específicos de ROS. En concreto, el paquete en cuestión que hará posible la inclusión del IMU en la plataforma sensorial será el **razor_imu_9dof** [28].

Ya que este paquete se encuentra incluido en los repositorios de ROS, disponiendo de conexión a Internet, su instalación resulta muy sencilla; como en el presente trabajo se está empleando la versión *Indigo* de ROS, el comando a ejecutar desde terminal para instalarlo es:

```
$ sudo apt-get install ros-indigo-razor-imu-9dof
```

Para el correcto funcionamiento del IMU, lo primero que debe realizarse es cargar en el microcontrolador integrado en el mismo el código necesario para extraer y procesar la información proveniente de los sensores y enviarla a través del puerto serie; en este sentido, el fabricante facilita mucho esta tarea al incorporar un *chip* de la familia ATMega en su dispositivo, puesto que estos microcontroladores son perfectamente compatibles con el entorno de desarrollo de usuario (IDE) de Arduino, conocido lenguaje de programación que lleva por bandera la simplicidad de programación y la legibilidad de su código, así como facilitar la utilización de librerías y código libre compartido por usuarios de todo el mundo. Por lo tanto, y sabiéndose que se va a emplear Arduino [29] como interfaz de desarrollo, se instala el mismo en el computador, en su versión para Linux (NOTA: tras descargar Arduino de la web oficial, en su versión para Linux, descomprimir y ejecutar el script ./install.sh, con permisos de superusuario).

Una vez instalado Arduino, se debe navegar hasta el directorio fuente del paquete **razor_imu_9dof:**

```
$ roscd razor_imu_9dof
```

Ahora, se copiará el directorio “Razor_AHRS” a la carpeta “Arduino” del directorio personal de Ubuntu:

```
$ cp -r src/Razor_AHRS ~/Arduino
```

En este momento, conectar el IMU al ordenador, según lo indicado en la Figura 25. Abrir la IDE de Arduino, y cargar el archivo **“Razor_AHRS.ino”**, presente en la carpeta anteriormente copiada. En este *sketch*, buscar el apartado “HARDWARE OPTIONS” y descomentar la línea correspondiente a la versión de hardware exacta del dispositivo empleado, que en este caso es la **10724**. Finalmente, e Herramientas – Placa, elegir “Arduino Pro or Pro Mini (3.3V, 8Mhz) w/ATMega328”, seleccionar el puerto serie adecuado y subir el código al microprocesador del IMU. Este proceso sólo deberá ser realizado una vez, y adicionalmente, otra vez tras el proceso de calibración, que se explica en el siguiente apartado de esta memoria.

(**Nota:** si durante la subida del código al IMU desde la IDE de Arduino se experimenta un problema de permisos, consultar el enlace[1] inferior.)

Antes de probar el IMU, se deberá crear el archivo de configuración (*.yaml*) que contenga los parámetros de calibración del mismo. Para ello, copiar el archivo plantilla "razor.yaml" y renombrarlo a "my_razor.yaml", en la carpeta *config* del directorio fuente del paquete razor_imu_9dof.

Si se han completado todos los pasos anteriores con éxito, la placa está lista para ser utilizada. Mientras permanece conectada al puerto USB del ordenador, ejecutar en una nueva terminal el siguiente comando:

```
$ roslaunch razor_imu_9dof razor_pub_and_display.launch
```

3.4.3. Calibración y colocación del dispositivo IMU 9DOF sobre la cámara TOF

Para que los sensores integrados en la placa Razor IMU 9DOF proporcionen información fiable, será necesario realizar una calibración previa, buscando los valores máximo y mínimo que cada sensor proporciona. El procedimiento de calibración se describe en el apartado 7 de [28], siendo necesario realizar las siguientes observaciones sobre el mismo:

- Los movimientos realizados durante el proceso de calibración deben ser muy suaves, para que no se falseen las mediciones.
- Los tres sensores que conforman la placa deben calibrarse siempre antes de que sea utilizada. No es necesario calibrar la placa más de una vez por entorno de trabajo, pero cada vez que se cambie dicho entorno debe repetirse el proceso de calibración, ya que los sensores, especialmente la brújula, pueden verse afectados (por ejemplo, por la presencia de elementos magnéticos).
- Para calibrar la brújula, se recomienda utilizar Processing, en su versión para Windows.

Una vez realizada la calibración, los valores obtenidos, que deben haber sido convenientemente anotados, deberán introducirse tanto en el *sketch*

[1] http://arduino-er.blogspot.com.es/2014/08/arduino-ide-error-avrdude-seropen-cant.html

"Razor_AHRS.ino", que se volverá a compilar y subir al microcontrolador del IMU, como en el archivo "my_razor.yaml". Hecho esto, el sensor queda calibrado y listo para su uso.

Debido a que, en última instancia, el cometido del IMU no es otro que el de proporcionar un sistema de referencia para la cámara de tiempo de vuelo, además de ayudar a localizarla en el mapa del entorno parcialmente desconocido objeto de trabajo, será necesario definir adecuadamente la ubicación del mismo respecto a la propia cámara. La determinación de esta ubicación se realizará mediante prueba y error, debiéndose cumplir una serie de condiciones:

- Preferiblemente, el eje X del sistema de coordenadas generado por el sensor deberá apuntar hacia el frontal de la TOF (que, según la Figura 15, es la parte positiva del eje Z del sistema de coordenadas solidaria a la propia cámara).
- El IMU permanecerá fijado a la cámara, para evitar en la medida de lo posible movimientos que lo descalibren una vez colocado y calibrado sobre la misma.
- Deberá procurarse la fácil conexión y desconexión del cable FTDI al sensor IMU mientras está adherido a la TOF, de forma que se evite provocar daños al cable o al propio IMU.

Teniéndose en cuenta todas estas variables, se encuentra que la forma óptima de colocar el sensor IMU en la cámara, de forma que sirva como elemento auxiliar en la ubicación de la misma respecto a la escena, es la siguiente (Figura 26):

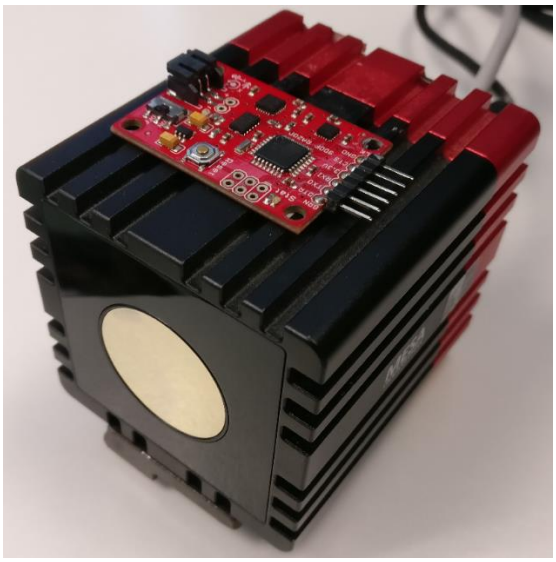


Figura 26: Sensor IMU situado correctamente sobre la cámara TOF

Para comprobar la calibración realizada en el punto anterior, así como la concordancia entre la información proporcionada por la unidad de medición inercial del robot móvil PMB-2 y el Razor IMU de SparkFun Electronics, se sitúa dicho sensor sobre el propio robot, y se compara la información proporcionada por ambos IMU. Los resultados de este experimento de verificación pueden verse en la Figura 27, donde se observa que ambos dispositivos generan una salida muy similar:

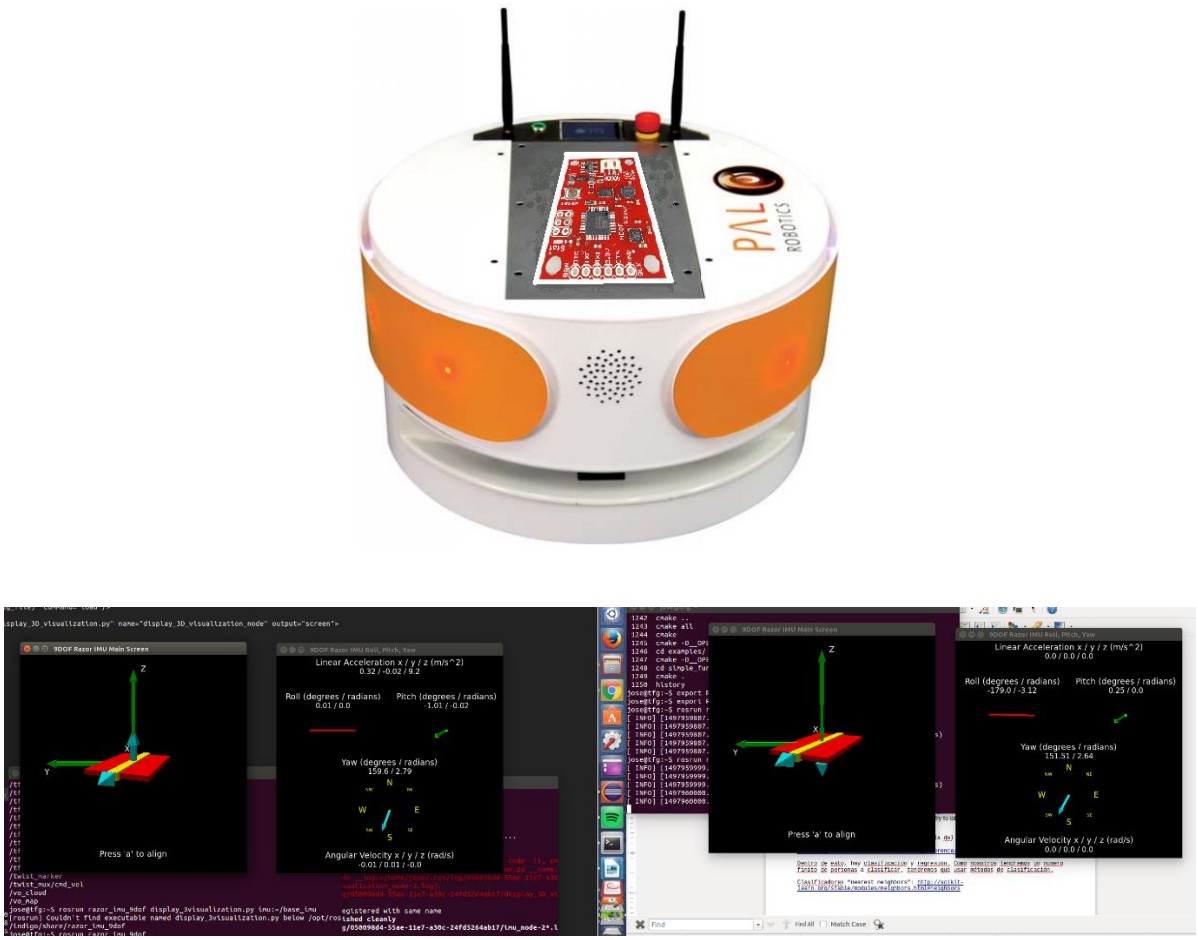


Figura 27: Orientación que debe tener el IMU, una vez calibrado, para que genere un sistema de coordenadas solidario al del robot móvil PMB-2

Para finalizar con el apartado de descripción, instalación y puesta a punto de la unidad de medición inercial que servirá como elemento auxiliar para la cámara de tiempo de vuelo, conviene citarse que se ha observado, durante las pruebas realizadas con la misma, que el sensor puede presentar ligeras oscilaciones (± 1º) a la hora de realizar las mediciones; no obstante, y teniéndose en cuenta la aplicación para la que pretende ser utilizado, dicho error de precisión no adquiere mayor relevancia.

Explicado el comportamiento de cada sensor, así como el método de instalación y utilización de los mismos, se está en disposición de pasar a definir y plantear de forma concreta la problemática a resolver en el presente proyecto, así como el *software* desarrollado para ello, que posteriormente se dotará de utilidad cuando se contextualice el **caso de estudio** sobre el cual se ejecutará la aplicación.

4. Software desarrollado

En el presente apartado de este proyecto, que podría considerarse el núcleo del mismo (junto con el **caso de estudio**), se explicará en detalle el *software* a desarrollar, estructurándose el contenido en los siguientes bloques:

- Definición de labores a realizar por cada sensor.
- Análisis de posibles soluciones para cada tarea, teniéndose en cuenta los **factores limitantes** y la sencillez, modularidad y posibilidades de optimización del código a programar.
- Explicación del *software* desarrollado, primero desde un punto de vista global y luego atendiendo a los diferentes bloques que lo conforman, con el objetivo de resolver las tareas expuestas en el apartado anterior.

Todos los algoritmos se programarán en los lenguajes C++ o Python, utilizando para ello la herramienta **Eclipse IDE Neon Cpp** y su extensión **PyDev** para Python. Adicionalmente, y dado el carácter personal de este apartado (el *software* desarrollado se realiza casi partiendo desde cero), se explicarán en el Anexo V los pasos a seguir para minimizar los posibles errores que puedan ocasionarse durante la instalación del *software* desarrollado.

4.1. Tareas a ejecutar

La aplicación final deberá ser capaz, en rasgos generales, de:

- Identificar correctamente la presencia de una persona en el entorno parcialmente desconocido que sea objeto del caso de estudio, minimizándose los falsos positivos y realizándose un seguimiento adecuado del movimiento de la misma en el escenario. Para ello, se utilizará como elemento sensorial la cámara de tiempo de vuelo, colocado de forma fija en un extremo de la escena.
- Ubicar de forma suficientemente precisa a la persona en la escena, tomándose como referencia un mapa proporcionado por el robot móvil PMB-2 [1]. Para ello, se utilizará el sensor IMU 9DOF como elemento generador del sistema de referencia, así como también se empleará dicho sensor, junto con la cámara TOF, para autoposicionar a dicha cámara en el mapa.

- Una vez que el robot haya navegado hasta donde se encuentre la persona en cuestión, será necesario saber de quién se trata, para pasar a realizar una serie de tareas concretas. Para ello, y empleándose como elemento sensorial la cámara Kinect, se realizará una identificación a partir de características fisionómicas, añadiéndose así un **carácter diferenciador** y valor añadido a la aplicación final, puesto que se aleja de otros métodos más convencionales y comúnmente usados, como los de reconocimiento facial. Utilizar las medidas del cuerpo para identificar a una persona, en vez utilizar rasgos característicos presentes en su rostro (u otro método de identificación) presenta la ventaja de que no existe posibilidad de falseo de datos, por ejemplo, utilizando vestimenta que oculte el rostro; no obstante, debido a que existen personas con características fisiológicas similares, sólo podrá considerarse este método como fiable si se emplea en relación a un grupo reducido de gente.

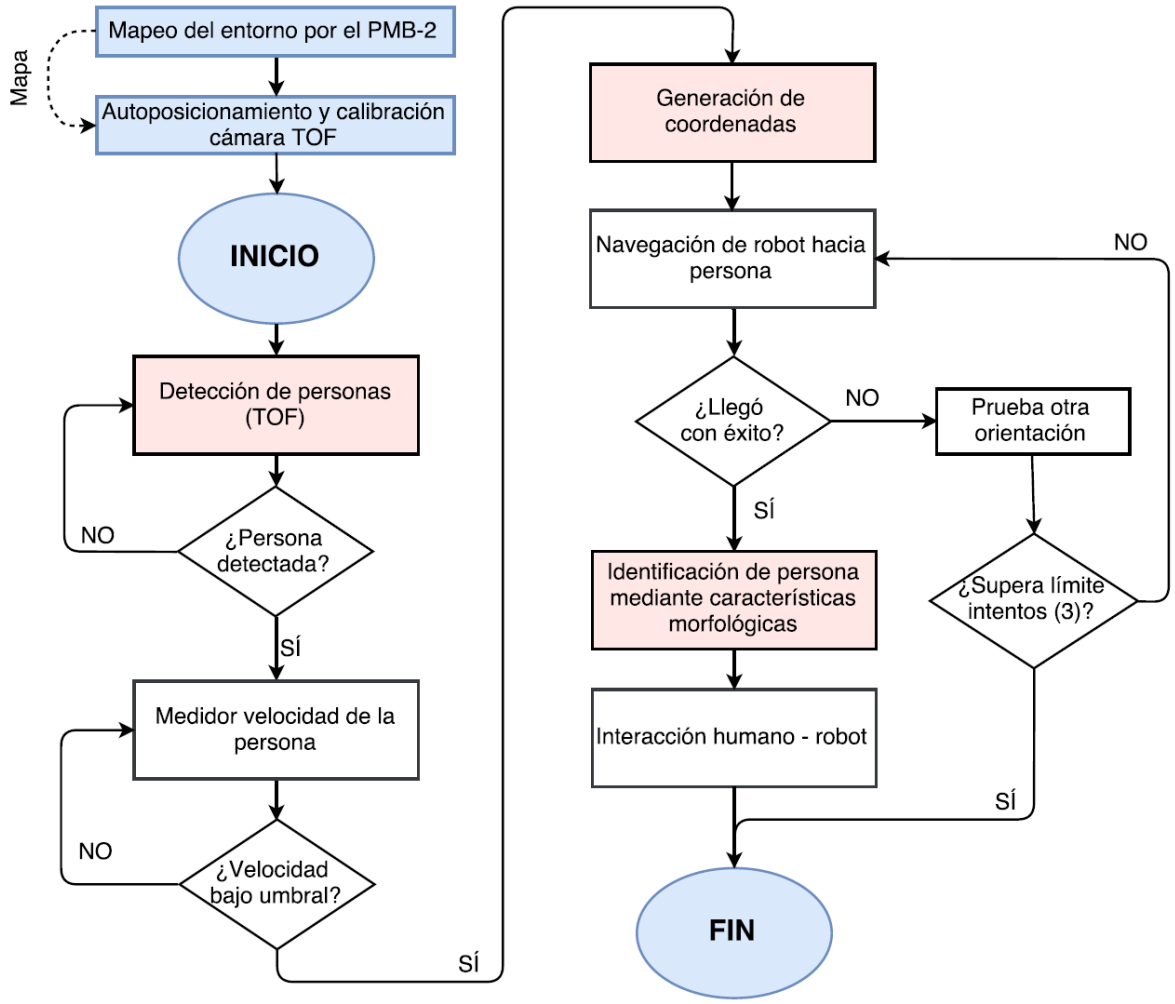


Figura 28: Diagrama general de la aplicación a desarrollar

Antes de pasar a desarrollar cada uno de los puntos descritos, y aunque ya se ha citado con anterioridad, resulta apropiado recordar que el presente documento tiene como objetivo explicar la parte de fusión sensorial para obtener información de un entorno parcialmente desconocido, con la finalidad de servir esta información de base para realizar tareas de interacción entre humanos y robots en dicho entorno; a pesar que la problemática de los robots se trata en otro proyecto colaborativo [1] con este, existen muchos nexos de unión entre ambos, con lo que no debe ser de extrañar que aparezcan constantes alusiones al mismo. En la Figura 28, en rojo, quedan indicadas aquellas partes (rectángulos) cuyo desarrollo es competencia del presente proyecto; el resto, por tanto, serán analizadas y resueltas en [1]. Las dos tareas iniciales, en azul, serán tratadas de forma colaborativa.

4.2. Posibles soluciones

En este apartado se describirá, en líneas generales, la forma en que cada tarea será abordada, teniéndose en cuenta que algunas de ellas presentan más conflicto que otras a la hora de resolverse; se hará especial hincapié en los objetivos a cumplir por la cámara de tiempo de vuelo, puesto que es el sensor más susceptible de error de todos los que se utilizan y, en función de cómo sea el entorno de trabajo, la aplicabilidad del *software* desarrollado y/o empleado con esta cámara podrá verse parcial o totalmente cuestionada.

4.2.1. Detección de personas en escena con la Mesa SR4000

Para la consecución de esta tarea, tras un proceso de búsqueda e investigación, se llega a la conclusión de que pueden seguirse dos caminos: utilizar nubes de puntos como información a partir de la cual detectar la presencia de personas; o emplear imágenes bidimensionales que, convenientemente tratadas, proporcionen información suficiente como para discernir si una región de píxeles determinada corresponde o no a la silueta de una persona.

Aunque ambas soluciones parecen, a priori, igualmente válidas, existen varios factores limitantes o condicionantes que harán decantarse por una u otra, y que se mencionarán y explicarán a continuación según el orden cronológico en que fueron apareciendo durante la resolución de esta problemática:

- A priori, una búsqueda a fondo desvela que existe una librería, llamada *PointCloud Library* [20] *(PCL)* que permite trabajar directamente con nubes de puntos. Las ventajas de emplear esta librería, así como las nubes de puntos, para discernir sobre la identidad de elementos presentes en una escena es obvia, puesto que, salvo en aplicaciones concretas, la manera habitual de decidir qué tipo de objeto es el que se presenta delante de una cámara es analizando su forma, y una imagen tridimensional de la escena proporciona mucha más información al respecto que una bidimensional. Además, se encuentra que, para una amplia cantidad de sensores, entre los que se encuentran las cámaras Mesa SR, existe una plataforma *software*, perfectamente integrada en ROS y que, haciendo uso de la PCL, permite realizar un seguimiento o *trackeo* de las personas que aparecen en un escenario concreto, llamada ***Open_PTrack*** [22]. Todos estos motivos son considerados como razones de peso para decantarse, de manera inicial, por el empleo de *Open_PTrack* como herramienta destinada a la detección de personas en el entorno parcialmente desconocido objeto de estudio, cumpliéndose así con la filosofía de reutilización de código *open-source* seguida a lo largo del desarrollo de este proyecto. Más documentación acerca de cómo se llevó a cabo el proceso de instalación y pruebas del software *Open_PTrack* hasta su puesta a punto para ser utilizado se encuentra disponible en el **Anexo IV**.
- Una vez se intentó adaptar el *software de Open_PTrack* para ser empleado en la aplicación final, se detectaron una serie de problemas, entre los cuales destaca uno en concreto: el escenario del que formará parte el caso de estudio presenta el inconveniente de tener un suelo con una superficie altamente pulimentada, que resulta ser un gran inconveniente para la cámara de tiempo de vuelo, puesto que los rayos emitidos por la misma, que idealmente deberían rebotar y volver al sensor, no lo hacen directamente, sino que toman rutas alternativas antes de volver a ser captados por el sensor, por lo que la imagen tridimensional generada por la TOF presenta una superficie irregular en la parte baja, correspondiente con el suelo, que para nada se corresponde con la realidad, que es la de una superficie totalmente plana. Un ejemplo de este inconveniente puede visualizarse en la Figura 29, donde se aprecia claramente que el techo, no reflectante, aparece

representado como una superficie plana; mientras que el suelo, altamente reflectante, es una amalgama de puntos bastante irregular que no dan sensación de coplanaridad, además de que los pies de la persona aparecen ligeramente por encima del suelo en sí, lo que lleva a pensar que los rayos reflejados en el suelo están tomando rutas alternativas antes de volver a la cámara y representando los puntos más lejos de lo que en realidad están.

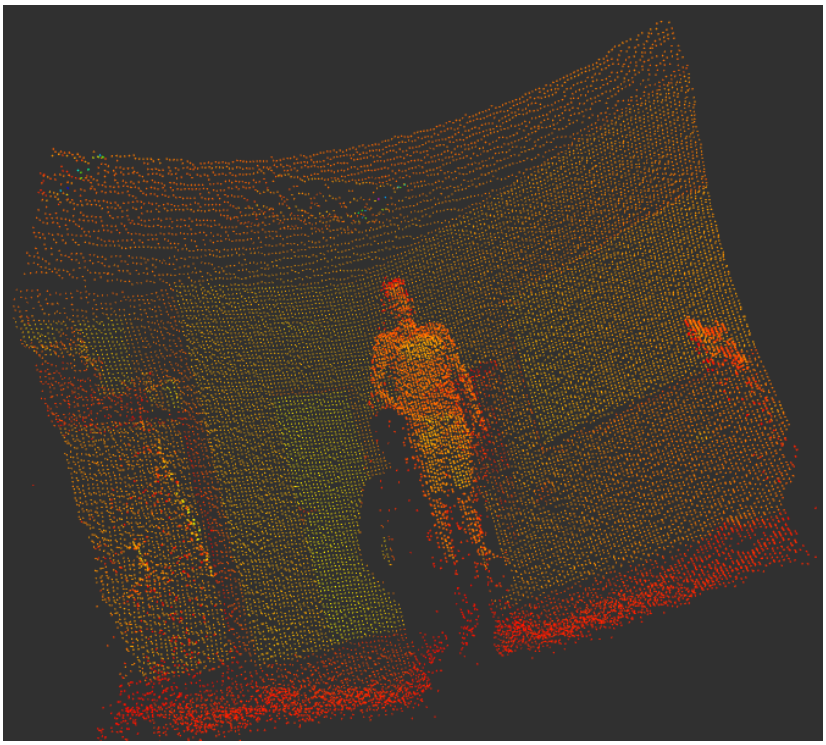


Figura 29: Nube de puntos en la que se aprecia la irregularidad del plano generado por la TOF a partir de un suelo reflectante

El no disponer de una superficie plana que pueda ser fácilmente identificada por los algoritmos y funciones de la PCL como la superficie del suelo implica que dichas funciones no disponen de un elemento de referencia a partir del cual situar y extraer el resto de contornos tridimensionales, y si los contornos no quedan claramente delimitados no pueden extraerse características de los mismos que ayuden a identificarlos. En resumen, sin un plano de suelo no es posible utilizar la PCL como herramienta de detección de personas, y Open_PTrack no resulta ser una herramienta conveniente para acometer este propósito. Formas de solventar este inconveniente serían procurar que el entorno parcialmente desconocido objeto de estudio disponga de un suelo mate, o colocar elementos no reflectantes en el mismo, como alfombras (Anexo IV).

Llegado este punto, resulta lógico pensar que la única alternativa posible es la de emplear una imagen bidimensional obtenida por la cámara como información a partir de la cual extraer contornos y siluetas de personas, para su posterior detección y seguimiento; para ello, es necesario definir previamente qué librerías y herramientas van a utilizarse para cumplir con tal fin. Tras realizarse una búsqueda, se llega a la conclusión de que lo más adecuado será utilizar la librería de visión por computador **OpenCV** [30], una serie de librerías *open-source* escritas en C++ (también disponibles en otros lenguajes de programación, como Python y Java) cuyas siglas significan *Open Computer Vision.* OpenCV [39] es un sistema multiplataforma, muy utilizado en tareas de desarrollo en investigación, y que se encarga, fundamentalmente, de proporcionar un sustrato y conjunto de herramientas para ayudar a construir aplicaciones relacionadas con el mundo de la visión por computador de forma fácil y sencilla. Para ello, esta librería contiene cerca de 500 funciones que cubren diversos campos en visión, como inspección de calidad, seguridad, calibración de cámaras, visión esteoroscópica… Además, incorpora funciones para el aprendizaje supervisado o *machine learning,* que suele ser etapas posteriores a las de tratamiento de imagen y que tienen como finalidad reconocer e identificar los elementos de interés presentes en la misma.

Para instalar OpenCV en Ubuntu, no es necesario, en un principio, realizar nada adicional si ya se ha instalado ROS, puesto que viene incluido de serie en los paquetes de dicha plataforma robótica. No obstante, es cierto que ROS utiliza formatos propios para sus mensajes, por lo que es necesario utlizar una herramienta específica que ayude a convertir estos tipos de datos de imagen a otros identificables y modificables por OpenCV; dicha herramienta será **“cv_bridge”** [31], un paquete de ROS que sirve como puente entre el mismo y OpenCV, permitiendo convertir los tipos de datos de imagen entre una plataforma y otra de forma sencilla.

4.2.2. Localización de personas en escena. Generación de coordenadas.

En este caso no hubo tanta ambigüedad a la hora de decidirse qué herramientas se emplearían, puesto que ROS proporciona, casi en su totalidad, todo lo necesario para resolver esta tarea, quedando pendiente adaptar los paquetes descargados a la función concreta a realizar en la aplicación final.

Primero, se deberá obtener un punto de referencia de la silueta de la persona detectada en por la cámara TOF, que típicamente será el centroide de la misma. Acto seguido, se calcularán las coordenadas de dicho punto respecto a la cámara, transformándose dichas coordenadas en mensajes entendibles para ROS, que se publicarán para la posterior utilización de los mismos por el robot móvil PMB-2. Para que el robot sea capaz de navegar hasta la persona, debe conocerse la ubicación de la cámara de tiempo de vuelo en el mapa, tarea que conformará un paso importante en la etapa previa de calibración y puesta a punto de los sensores en el el caso de estudio, y que será realizada utilizándose herramientas de autoposicionamiento, que toman como parámetros de entrada la información proveniente del propio mapa y de la cámara TOF, así como del acelerómetro IMU. Toda esta información, finalmente, servirá al robot para navegar hasta la persona, como ya se ha dicho anteriormente; así como proporcionará una forma de determinar el instante en que el robot **debe** iniciar el movimiento, mediante el estudio de la variabilidad de la posición de la persona en el escenario parcialmente desconocido respecto al tiempo (es decir, su velocidad de desplazamiento); estas últimas tareas serán resueltas en [1].

4.2.3 Identificación de personas mediante características fisionómicas

Para finalizar con este apartado de la memoria, se expondrá y justificarán las herramientas elegidas para resolver esta problemática.

En primer lugar, y teniendo en cuenta la información presente en [32], [34] y [35], que indican que la utilización de este método resulta viable, para obtener de una forma suficientemente, precisa y veraz las características fisionómicas que permitirán llevar a cabo la identificación de la persona hasta la cual se ha navegado con el robot móvil, será imprescindible que la cámara que va a actuar como elemento sensorial, que no es otra que la Kinect V1, esté adecuadamente colocada encima del robot Meka M1, por lo que se conoce de forma precisa su ubicación, gracias al árbol de transformadas proveniente del proyecto de navegación, en el cual la cámara Kinect aparece incluida en el modelo URDF (lenguaje propio de ROS en formato XML, específicamente diseñado para definir el modelo geométrico y cinemático de un robot) del Meka.

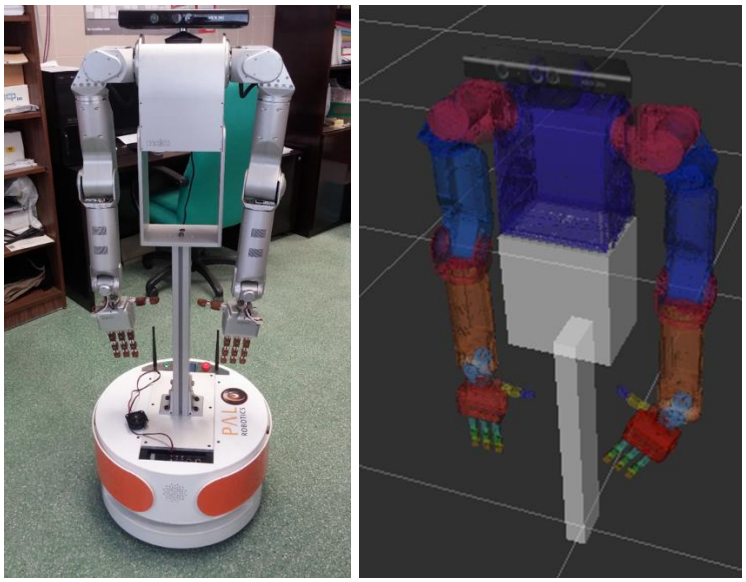


Figura 30: Situación de la cámara Kinect en el robot móvil (izquierda). Modelo URDF del robot, incluyendo la cámara Kinect, para su adecuada referenciación espacial con TF (derecha)

Obviando este detalle, sólo existe una herramienta capaz de proporcionar esta información, y no es otra que las librerías **OpenNI** y **NiTE**, que por defecto inlcuyen un *tracker* de la estructura morfológica de un ser humano, y que se encuentra programado en ROS, dentro del paquete *openni_tracker*. Una vez aclarado esto, resulta conveniente discutir acerca de qué extremidades o partes del cuerpo serán tomadas como puntos de referencia para efectuar las medidas que posteriormente servirán para discernir ante quién se encuentra el robot. Para ello, se ha tenido en cuenta, fundamentalmente, el hecho de que dichas partes del cuerpo **siempre** queden visibles a la cámara Kinect, así como que su posición espacio-temporal sea lo más invariante posible. Teniéndose en cuenta la primera condición, se llega a la conclusión de que no es convieniente emplear datos provenientes de la mitad inferior del cuerpo (piernas), puesto que no siempre quedan visibles a la Kinect y pueden generar falseamiento de datos; esto deja como únicas candidatas las articulaciones del torso y extremidades superiores, las cuales también se eliminan de la lista, puesto que son muy subceptibles a entrar en movimiento y provocar que ciertas medidas puedan ser muy variables. Con toda esta información, se seleccionan cuatro articulaciones del cuerpo cuyas coordenadas se obtendrán como parámetros de entrada (cabeza, torso y hombro izquierdo y derecho); posteriormente, se calcularán las distancias relativas entre cabeza-torso y entre los hombros, que deberían ser lo suficientemente constantes en el tiempo como para permitir discernir, entre un grupo relativamente reducido de personas, de quién se trata.

El siguiente dilema que se plantea es elegir qué herramienta de decisión se utilizará para identificar a la persona a partir de los datos obtenidos. Tras una breve investigación, se llega a la conclusión de que lo más adecuado será entrenar una red neuronal con múltiples datos provenientes de las personas que formen parte del conjunto de individuos que van a interactuar con el entorno parcialmente desconocido, para que posteriormente dicha red neuronal se encargue de identificar ante quién se encuentra el robot.

Una **red neuronal artificial** es un sistema computacional que intenta imitar a los sistemas neuronales biológicos, basándose en los conceptos de entrenamiento y aprendizaje cognitivo; para ello, los elementos atómicos que conforman una red neuronal, típicamente organizados en capas, se interconectan entre sí, intercambiando información ponderada (generalmente entre 0 y 1) que va autorregulando su comportamiento y respuesta ante estímulos de entrada.

Las redes neuronales se emplean para resolver problemas de aprendizaje, que pueden clasificarse [37] fundamentalmente en dos tipos:

- De aprendizaje supervisado, en las que los datos de entrada representan atributos de los diferentes elementos conocidos y almacenados en una base de datos. Dichos problemas se subdividen en problemas de clasificación (las muestras pertenecen a dos o más clases discretas) y regresión (la salida es un valor continuo relacionado con los atributos de entrada).
- De aprendizaje no supervisado, en las que los datos de entrada no tienen una correspondencia a un conjunto de individuos conocidos; el objetivo fundamental de las redes neuronales pertenecientes a esta categoría es el de encontrar grupos de similaridad en los que clasificar al conjunto de individuos existente.

Teniendo en cuenta lo explicado en dicho anexo, el tipo de red neuronal que se necesita implementar pertenece a la categoría de aprendizaje supervisado, puesto que se debe entrenar con datos (medidas fisiológicas) de un conjunto previamente delimitado (conjunto de personas), para así ofrecer ante un nuevo estímulo (una nueva muestra) una salida ponderada que se aproxime a uno de los individuos del conjunto de datos con los que fue entrenada; esto es lo que se denomina *feedforward neural*

network (a partir de ahora FNN). Para implementar un nodo en ROS que se encargue de estas tareas, se lleva a cabo una búsqueda sobre qué librerías existen, encontrándose las siguientes:

- OpenNN, librería para el desarrollo e implementación de redes neuronales en C++.
- PyBrain, herramienta en Python con utilidades similares a OpenNN.
- Scikit-Learn, librería de Python especializada en el *machine-learning*.

Analizando las diferentes opciones, se descarta OpenNN por estar escrita en C++, lo cual dificultaría la comprensión del código, que es más legible en Python; además, C++ sólo es indispensable para tareas que requieran de mucha velocidad de procesamiento, lo cual no es el caso. Por lo tanto, se decide utilizar **Python**, lenguaje de programación perfectamente compatible con ROS y que es elegido por su mayor sencillez y versatilidad a la hora de programar y porque permitirá una mayor facilidad a la hora de crear una interfaz gráfica que ayude a que el proceso de entrenamiento sea lo más sencillo posible. Ya que Python será el lenguaje a emplear, se comparan las dos alternativas restantes, eligiéndose **PyBrain** [36] como la más adecuada, puesto que Scikit-Learn abarca un abanico mucho más amplio que el de las redes neuronales y su documentación y proceso de aprendizaje resultan ser mucho más complejos.

Con la mención de la herramienta PyBrain como librería encargada de construir redes neuronales, sólamente queda decidir qué librería se utilizará para crear una interfaz gráfica, lo más simple posible, que ayude a entrenar la red neuronal, escogiéndose **Glade** y **GTK** como herramientas para acometer esta tarea. Glade es una herramienta de asistencia a la creación de interfaces gráficas, que permite diseñar las interfaces visualmente mediante el método de *drag and drop*, lo cual simplifica enormemente el proceso de creación de la ventana o ventanas de la aplicación; además, Glade maneja los eventos (pulsación de botones, introducción de texto en campos…) mediante *handlers* o señales, que son las que se utilizan en Python para, mediante funciones, indicar qué realizará en concreto cada elemento. Dichas señales no son otras que las propias de GTK, que es un API para el desarrollo de interfaces gráficas y que contiene el código y funciones necesarias para que los diferentes elementos (widgets, botones, listas…) de dichas interfaces realicen la función para la

que fueron diseñados. Por tanto, podría decirse que GTK es el sustento de la interfaz, mientras que Glade es un mero mediador, un asistente que facilita el proceso de ubicación visual y configuración de los elementos de la interfaz a desarrollar.

Una vez tomadas todas las decisiones en cuanto al proceso de implementación de las aplicaciones a programar, se cierra el proceso de búsqueda de alternativas y debate, pasándose en los puntos siguientes a explicarse, en base a las decisiones tomadas en este punto, cómo han ido desarrollándose los programas encaminados a resolver cada una de las tareas propuestas a realizar por cada sensor para la obtención de la información necesaria y suficiente para la caracterización del entorno parcialmente desconocido, con el fin de facilitar información a un robot, lo cual, y es conveniente recordarlo, constituye el objetivo fundamental de este proyecto.

4.3. Implementación

La aplicación a desarrollar, cuyo objetivo general es el de facilitar información a un robot acerca de un entorno parcialmente desconocido, centrándose en la detección e identificación de personas en el mismo, procede ahora a desgranarse, según cada uno de los grandes bloques que aparecen en rojo en flujograma de la Figura 28. Si a lo largo de los subapartados de este punto se hace referencia a **nombres de paquetes o nodos**, todos ellos se encuentran dentro del *workspace* "**fusion_sensorial_ws**", incluido en el CD y cuyo sistema de archivos se encuentra representado gráficamente en el **Anexo externo 5**.

4.3.1. Detección de personas en escena con la Mesa SR4000

Tal y como ya se explicó en el apartado 4.2 del presente documento, esta tarea se resolverá mediante la ayuda de OpenCV, y empleándose el lenguaje de programación C++. Un resumen de la evolución que seguiría el código sería el que aparece en la Figura 31:

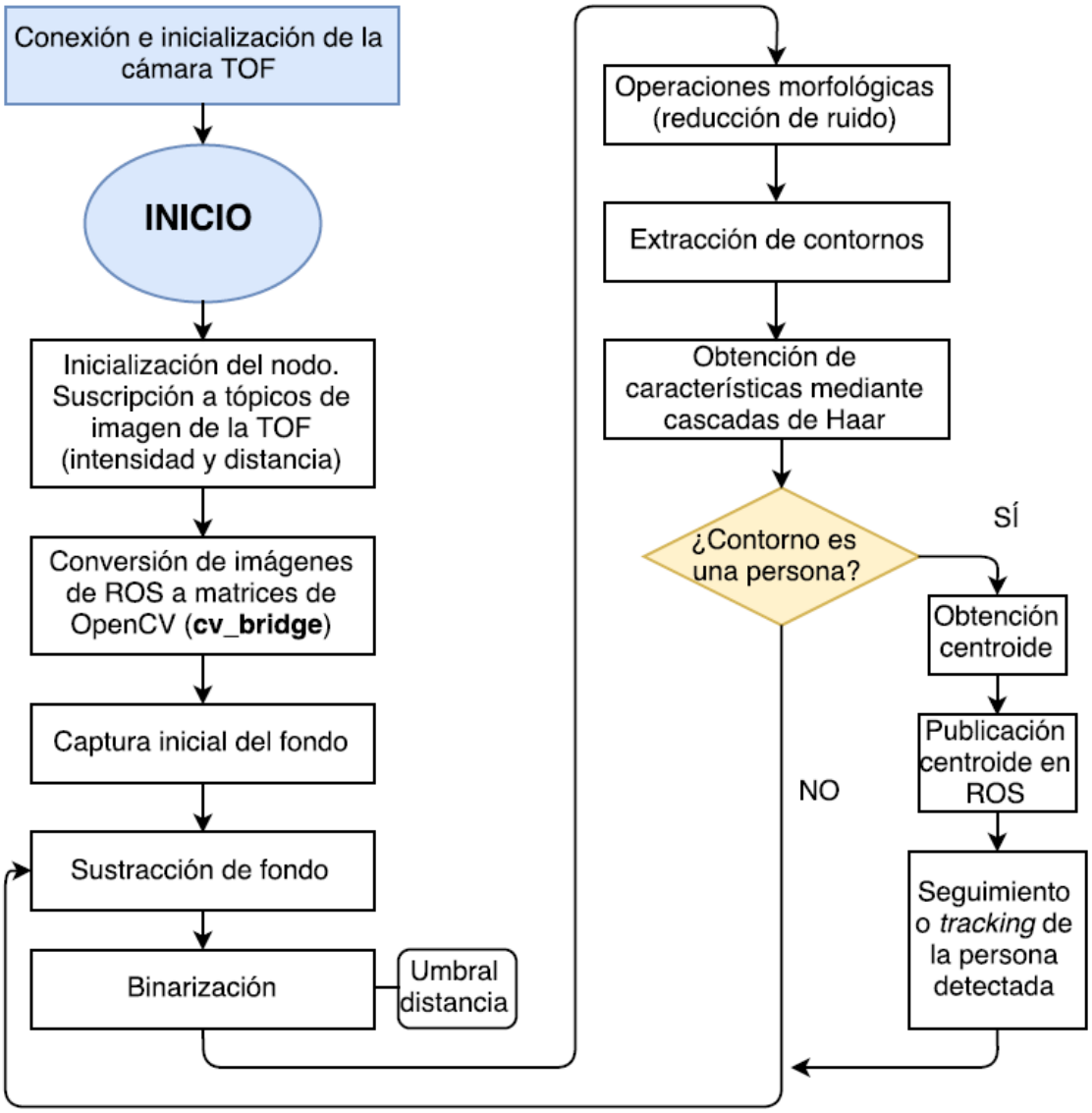


Figura 31: Nodo de tratamiento de imágenes de la TOF y detección de personas en las mismas

El primer paso para poder ejecutar este nodo, llamado “image_processing” en el sistema de archivos, dentro del paquete “swissranger_image_tracker”, será conectar la cámara Mesa SR4000 al PC, mediante conexión Ethernet, y situarla de forma fija en un trípode, que deberá tener una altura aproximada de 1.70m para que el entorno pueda ser monitorizado de forma adecuada; dicha altura se justifica teniendo en cuenta que es similar a la altura media de una persona, con lo cual la cámara obtendrá así una perspectiva adecuada de la escena. Una vez hecho esto, la cámara se inicializará según lo dispuesto en el apartado 3.2.4 de la presente memoria.

Posteriormente, el nodo en cuestión es inicializado, y se realizan las suscripciones a los tópicos de imágenes de intensidad y distancia de la cámara TOF; dichos tópicos, publicados por el *software* predeterminado de la cámara (instalado en

el punto 3.2.4), son del tipo “sensor_msgs::Image”, y deberán ser convertidos a imágenes del tipo “cv::Mat”, que son las empleadas por OpenCV; esta tarea de conversión se lleva a cabo con la herramienta **cv_bridge** [31].

Seguidamente, se realiza una captura inicial de la escena. Esta captura, hecha en base a la imagen de distancia proporcionada por la cámara (en escala de 0 a 255, ya que las imágenes son de 8 bits) servirá como **fondo** o *background* durante todo el proceso de adquisición y tratamiento de imágenes, por lo que es fundamental que se realice en **ausencia de personas** y que, una vez hecha, se **evite mover la cámara**, si lo segundo ocurriera, un *slider* localizado en la interfaz final de la aplicación permitirá volver a realizar una captura del fondo, solventando esta inconveniencia (Figura 32).

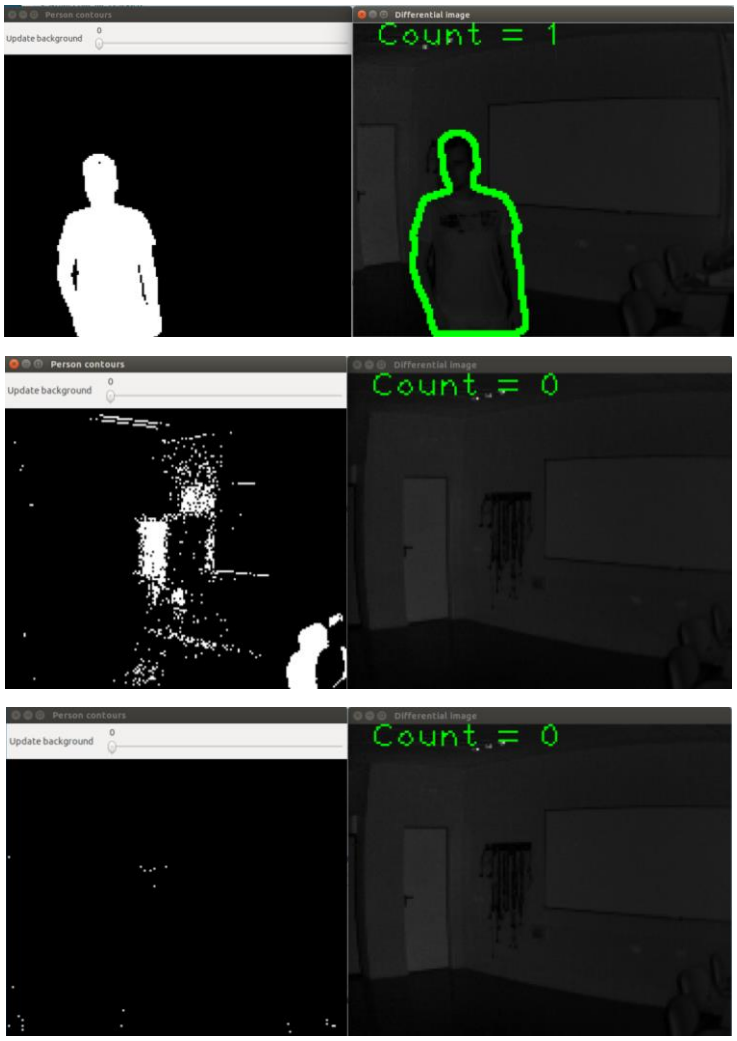


Figura 32: Interfaz gráfica final, en la que aparecen dos imágenes: diferencial binarizada (izquierda) e imagen de intensidad con contornos detectados como personas (derecha). Las imágenes, de arriba abajo, representan: situación de detección normal; escenario con cámara movida; y escenario tras haberse realizado una nueva captura de fondo, con el *slider* situado arriba a la izquierda.

A partir de este momento, las sucesivas imágenes de distancia recibidas a través de la suscripción realizada al tópico correspondiente serán comparadas y sustraídas a la imagen del fondo, quedando, por tanto, una imagen **diferencial** de la escena disponible para su tratamiento; esto facilita en gran medida la detección de posibles elementos que entren a la escena, estáticos o en movimiento, puesto que las partes que no resultan de interés se eliminan, evitándose además la aparición de falsos positivos en futuras etapas de detección de contornos y uso de operadores morfológicos.

Después de realizarse la sustracción de fondo, la imagen debe binarizarse (Figura 33) para que, de esta forma, las ROI o regiones de interés de la escena aparezcan con valor “uno” o “verdadero” (color blanco); y el resto, en negro. Se tomará como umbral de binarización un **umbral de distancia**, definido como una variable en el archivo de cabecera (“.h”) de este nodo y configurable a través de su archivo *launch* o de lanzamiento. Dicho umbral no es autoconfigurable, sino manual, y viene por defecto establecido a un valor adecuado (200) para evitar la aparición de ruido en escena y permitir una adecuada detección de los objetos que entren a la misma. Además, debe tenerse en cuenta que dicho umbral no es absoluto, sino diferencial, lo que querrá decir que, una vez sustraídas ambas imágenes, aquellos valores de píxeles entre la imagen de fondo y la actual que se encuentren a una distancia mayor entre sí de la especificada en el umbral tomarán valor “uno”, mientras que el resto adquirirán el valor “cero”. Deberá llegarse a un compromiso a la hora de configurar este parámetro, puesto que, aunque valores más bajos aumentan la sensibilidad a la presencia de nuevos objetos en escena, también hacen que la inmunidad al ruido proveniente de la adquisición de la propia imagen por parte de la cámara sea mucho menor.

Figura 33: Fotograma de la imagen diferencial binarizada, obtenida a partir de la sustracción de la imagen de profundidad de la TOF con el fondo, donde se observa una persona en escena

Acto seguido, y si la etiqueta “do_morph” de efectuar operaciones morfológicas está activa en el archivo de lanzamiento, se efectuará una erosión y dilación de la escena, que elimine píxeles falsos provenientes del ruido generado en la propia imagen. Una comparativa de cómo este proceso puede mejorar la imagen se muestra en la Figura 34.

Figura 34: Imagen diferencial antes y después del proceso morfológico de eliminación de ruido

Después, la imagen binarizada está lista para obtener los contornos presentes en ella, que se filtrarán por tamaño y ratio de proporción para eliminar aquellos que claramente no puedan corresponder al de una persona (función “proportionsPerson”). Este filtrado se realiza, fundamentalmente, para evitar que, objetos que podrían ser detectados como una persona mediante los operadores morfológicos elegidos (cascadas de Haar [32]) pero que en realidad no son personas (como, por ejemplo, el robot Meka) den un falso positivo; en una persona promedio, el ratio altura/anchura es mayor que en el robot, lo cual ofrece el elemento discriminante necesario para diferenciarlos. No obstante, cuando tanto la persona como el robot se encuentren cerca del objetivo de la TOF, no serán vistos de cuerpo entero, por lo que el ratio

deberá ser variable (Ec. 3), en función de la distancia a la que se encuentre el contorno a identificar; este problema se resuelve teniendo en cuenta la distancia del propio contorno respecto a la cámara, mediante funciones que permiten obtener este dato y que se utilizarán también para obtener las coordenadas espaciales de la persona en el apartado 4.3.3.

La fórmula empleada para obtener dicho ratio es:

$$R = d \cdot r \qquad (Ec. 3)$$

En la Ec. 3, "r" es un ratio de aspecto fijo, establecido en 200 que se pasará a tanto por uno y se ponderará con "d", que es la distancia al centroide de la persona; así, a mayores distancias, el ratio final "R" será mayor y, en consecuencia, la figura deberá ser más estilizada para que la detección sea válida, eliminándose así en gran parte los falsos positivos dados por el robot, como puede verse en la Figura 35.

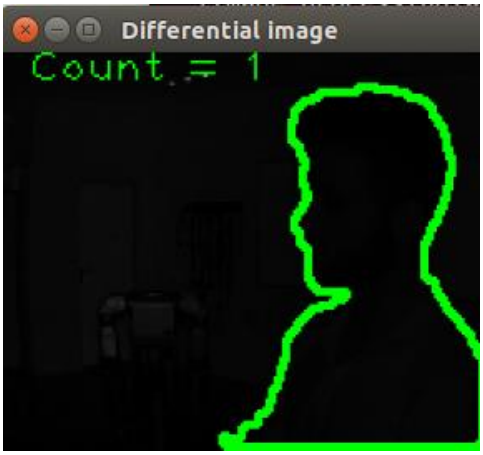


Figura 35: Detección de persona frente a ausencia de detección del robot Meka

El resto del código se centra en la parte de identificar los contornos existentes como personas, prestándose especial atención al hecho de que, en cada *frame*, la misma persona puede haber cambiado de posición, por lo que se deberá mantener un registro de las personas que existen, así como cuál era su última posición. Así, se podrá comprobar, en cada *frame,* que dicha persona sigue en escena, y así poder realizar un seguimiento de su posición; casos especiales de esto son aquellos en los que la persona entra en escena por primera vez o sale de ella. Para explicar mejor este proceso, se adjunta un diagrama de flujo del mismo, que ya se incluyó en la Figura 31 como la disyuntiva "¿Contorno es una persona?" (en amarillo) y que ahora se desgrana en la Figura 36:

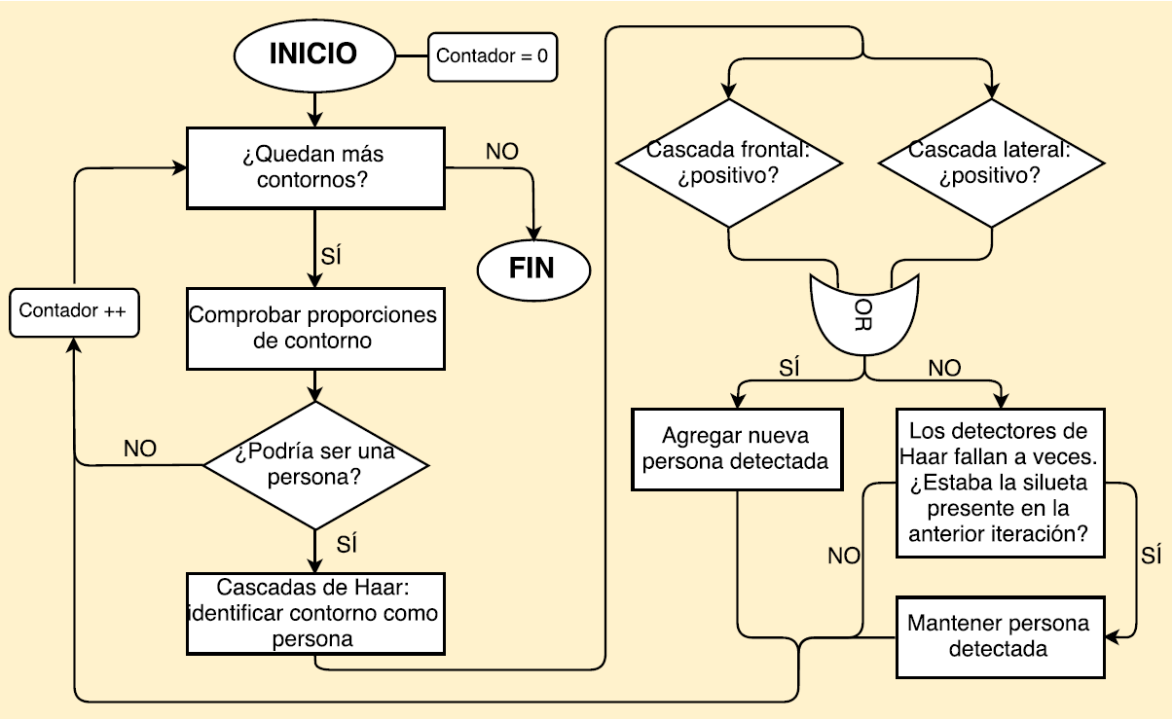


Figura 36: Algoritmo de decisión para detectar a una persona en escena

La idea es que, en cada *frame*, se mantenga un registro de las personas detectadas, y en el *frame* posterior se realice una comparación, para ver si las personas detectadas en el mismo ya estaban presentes o no; en caso de que, momentáneamente, en dicho *frame* o fotograma no se detecte a la persona mediante una cascada de Haar, siempre se podrá corroborar, comparando el contorno actual con el registrado en el *frame* anterior, si son el mismo; para ello, se comprobará si el centroide de uno está incluido dentro del otro. Si alguna de las condiciones se cumple, tal y como se ve en la Figura 36, la detección se da por válida.

Para finalizar la explicación de este algoritmo, indicar que, una vez finalizada la etapa de detección, se publicarán en el sistema de ROS los siguientes tópicos:

- "people_track/found": número en formato entero que indica el número de personas presentes en escena.
- "people_track/centroids": mensaje en el que aparecen las coordenadas, tanto en X (1 a 176 píxeles) como en Y (1 a 144 píxeles), de los centroides de los contornos que han sido identificados como personas. Si existe más de una persona a la vez en escena, se enviarán dichas coordenadas en formato de *array*, según mensajes propios definidos en la carpeta *msg* del paquete "swissranger_image_tracker" (ver Anexo externo 5).

4.3.2. Autoposicionamiento de cámara TOF en el entorno de trabajo

El requerimiento básico para que el presente trabajo funcione en conjunción con [1] es el de referenciar correctamente todos los elementos en el espacio, tanto en posición como en orientación (*pose* o 6D en ROS). Para ello, la herramienta más indicada sobre la que se apoya ROS es TF (Anexo I (III)). Se realiza, por tanto, una colaboración con el proyecto anteriormente mencionado, para proporcionar una precisa localización de la cámara TOF en el entorno que permita ubicar corerctamente a las personas en escena, empleándose los conocimientos pertinentes relacionados con la transformación y tratamiento de los datos porporcionados por la cámara de profundidad y con la ayuda de los conocimientos desarrollados en [1] en temas como la navegación autónoma, especialmente en lo relacionado con el ámbito de la localización y autoposicionamiento de un elemento móvil en un mapa determinado.

Antes de proceder a explicar el proceso de autoposicionamiento de la cámara TOF en el mapa, conviene aclarar cómo será la situación de partida para la misma. En concreto, la cámara se situará empotrada en un soporte de pared o en un trípode de altura similar a una persona (1.70 m). En esta posición, una medición tridimensional exacta de la misma, además de una correcta obtención de la orientación de los tres ángulos de Euler (*roll, pitch* y *yaw,* como se pueden observar en la Figura 37) de la cámara es prácticamente imposible de conseguir de forma manual, y sería imprecisa y cambiante, ya que la cámara debería situarse en el mapa tomando algún punto de referencia característico, lo cual no serviría para cualquier entorno, como el caso de una habitación vacía. Por tanto, el apartado de la orientación debe resolverse mediante el empleo del sensor IMU colocado sobre la cámara (apartado 3.4.3).

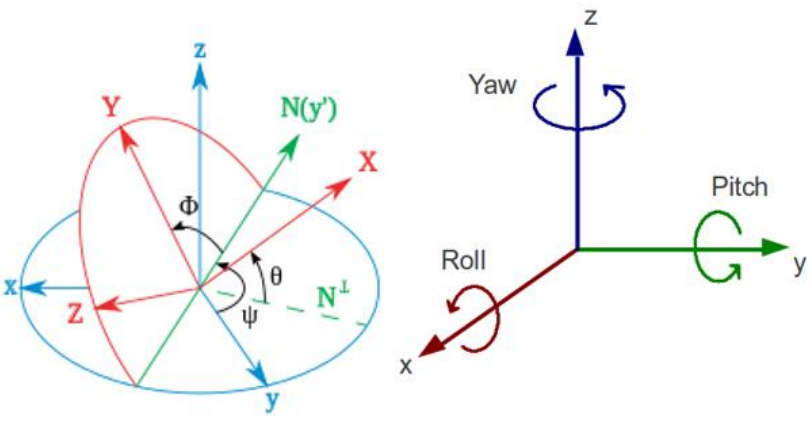


Figura 37: Ángulos Euler renombrados como en la aeronáutica y robótica (*roll, pitch, yaw*)

Para poder utliizar los datos que proporciona el IMU como información útil, se deberá generar una transformada espacial entre el frame de referencia de la nube de puntos de la cámara y un nuevo frame, que se llamará "**gyroscope**" y que estará situado en el mismo punto que la cámara, pero con orientación variable, en función de cómo el sensor se mueva. Para ello, se extraen los datos del sensor del tópico publicado por los nodos incluidos en el paquete "razor_imu_9dof", instalado en el apartado 3.4.2, y que son del tipo "Sensor_msgs::IMU", y se convierten en la transformada citada anteriormente, con la ayuda de la librería TF. Este proceso se puede encontrar resuelto en C++ en el nodo "gyroscope_tf".

Ya que ha quedado patente que un posicionamiento manual de la cámara implicaría serias desventajas, se procede a idear un sistema que permita la autolocalización de la misma en un mapa conocido. Dicho mapa será generado en [1], en concreto por la base móvil PMB-2, gracias a un sensor láser que incorpora, y del cual será de vital importancia conocer la altura a la que encuentra, puesto que el mapa se ha generado según la información captada por el láser a dicha altura.

La base autónoma PMB-2 se autolocaliza [1] en el mapa que crea el algoritmo **AMCL** (*Adaptive Monte Carlo Localization)*. Se pretende emplear dicho algoritmo para autoposicionar la cámara de tiempo de vuelo en el mapa, para lo cual deben tenerse en cuenta algunas consideraciones:

- Dicho algoritmo está ideado para el uso de LIDAR, no para cámaras de profundidad.
- Se hace necesario el uso de odometría para poder ejecutar de forma satisfactoria el algoritmo AMCL; sin embargo, la cámara carece de ella, ya que no es un robot y no se desplaza como tal, por lo que la autocorrección que un robot realiza gracias a la odometría según se traslada o gira no podría ocurrir.

Para solventar la primera de las consideraciones planteadas, se hace indispensable realizar una conversión de la información que proporciona la cámara de profundidad, en forma de nube de puntos (***PointCloud*** en los tipos de mensajes de ROS), al tipo de datos empleado por el robot para generar los mapas y efectuar la navegación autónoma, que es del tipo ***LaserScan***; dicha tarea puede ser fácilmente

resuelta gracias al paquete **"pointcloud_to_laserscan"**, descargable desde los repositorios oficiales de ROS y que se puede instalar de la siguiente manera:

```
$  sudo apt-get install ros-indigo-pointcloud-to-laserscan
```

Dicho paquete ofrece su utilidad gracias al uso de los archivos de lanzamiento (*launch*), en el cual deben especificarse parámetros como el tópico desde el que se suministra el *PointCloud* de entrada, la región angular que se observará para realizar la conversión y el paso para el ángulo que se tomará como intervalo de muestreo de la nube de puntos (se intentará ajustar este parámetro para que el paso del ángulo coincida con el que se recorre entre un píxel y otro de la imagen, teniéndose en cuenta que la cámara TOF cumbre aproximadamente un rango (α) de 45 grados y su imagen tiene una anchura de 176 píxeles); la fórmula que relaciona ambos parámetros con el paso (Δ) del ángulo aparece descrita en la Ec. 4, siendo α el rango de visión de la cámara, que deberá especificarse en radianes:

$$\Delta = \frac{\alpha}{176} \qquad (\text{Ec. } 4)$$

La **altura** a la que se encuentra situada la cámara también es un parámetro de vital importancia a la hora de efectuar la conversión de *PointCloud* a *LaserScan,* y deberá especificarse para que tome valores similares a los de la altura respecto al suelo que tiene el láser del robot móvil PMB-2, teniéndose en cuenta que la cámara se encuentra situada a una cierta altura con respecto al suelo (aproximadamente 1.7m); esta medida se deberá conseguir de forma manual (Figura 38).

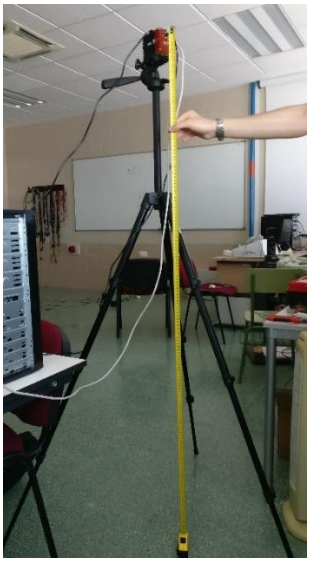


Figura 38: Medición manual de la altura a la que se encuentra situada la cámara TOF

Uno de los problemas que se encontraron al intentar utilizarse este paquete fue que la cámara TOF proporciona la nube de puntos referenciada a su propio sistema de coordenadas intrínseco, que como se puede ver en la Figura 15 tiene como eje principal de proyección de datos (el perpendicular al sensor) el eje Z; sin embargo, el eje horizontal principal en un sistema de coordenadas convencional es el X, por lo que esta nube de puntos debe ser rotada convenientemente antes de poder ser empleada como información útil para el conversor de datos p*ointcloud_to_laserscan*. Para ello, se programa un simple nodo, que se suscribe a dicha nube de puntos, efectúa las rotaciones pertinentes en los ángulos *roll* y *pitch* y devuelve un *PointCloud* exactamente igual pero convenientemente orientado, como se puede observar en la Figura 39. Aprovechando el *frame* creado con la librería TF [27] a partir de la información que proporciona el IMU, se referencia esta nueva nube de puntos respecto a dicho *frame*, para que la *PointCloud* se mueva siempre de forma solidaria a él.

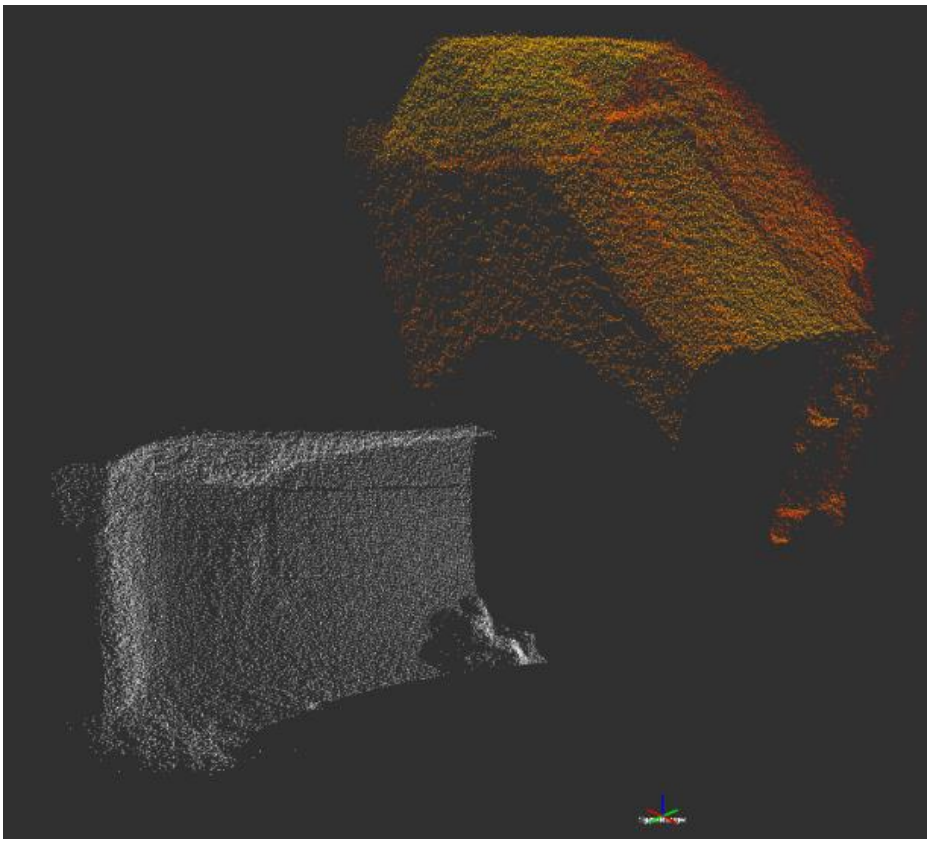


Figura 39: *PointCloud* original de la cámara TOF (en color), orientado respecto al eje Z de la misma; y *PointCloud* convenientemente girado (blanco)

Para solventar la carencia de odometría, requerida para que se pueda ejecutar el algoritmo AMCL, se empleará una “falsa” odometría, aprovechando que el trípode

sobre el que va montado la TOF permite efectuar giros en cualquiera de los tres ángulos de Euler; el paquete *laser_scan_matcher* se encargará de generarla, tomando como información de entrada el *LaserScan* generado anteriormente y proporcionando una odometría aproximada a la salida. Más información acerca de este aspecto puede encontrarse en [1].

Una vez resueltos todos pasos previos, tan sólo queda pendiente cargar el mapa en Rviz [1], lanzar los nodos de conversión de datos de la TOF así como el encargado de ejecutar el algoritmo AMCL, y visualizar los *frames* del "falso robot" creado como ayuda para autoposicionar la cámara. Cuando todo está listo, se deberá indicar una posición o "pose" inicial de la cámara en el mapa, que debe ser tal que coincida en la medida de lo posible con su posición real (para ello, comprobar la correspondencia entre los límites del mapa y el *LaserScan* creado por la cámara TOF) y **rotar** la cámara lentamente en el trípode sobre el eje Z (ángulo ***yaw***) para que el algoritmo de autolocalización vaya ajustando la posición de la TOF (Figura 40):

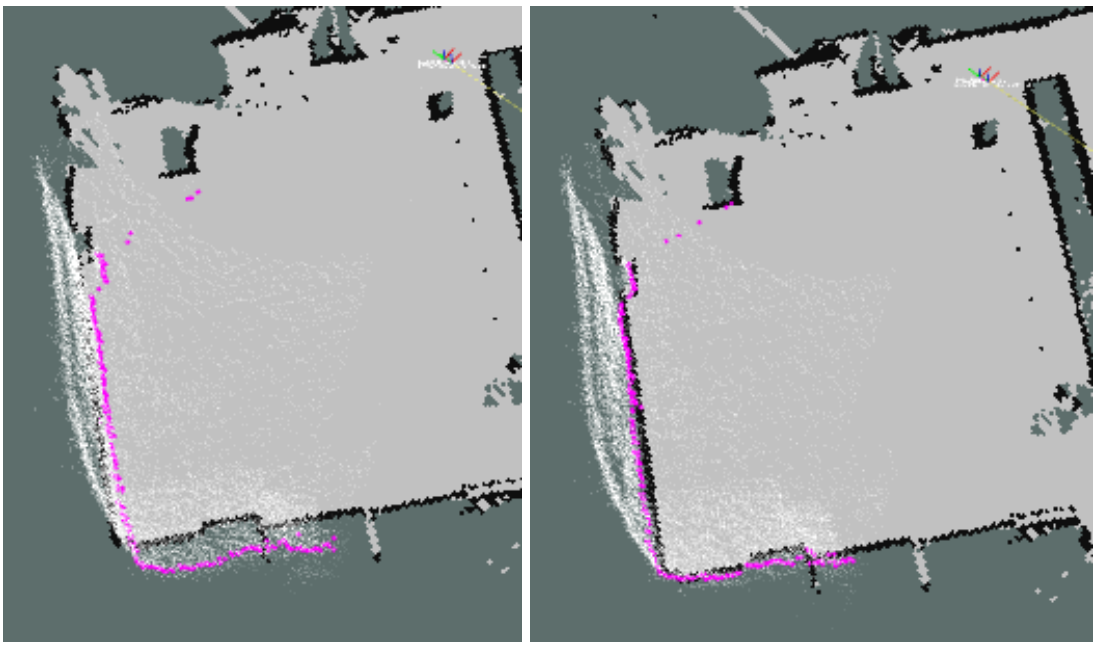


Figura 40: Proceso de autolocalización de la cámara TOF: posición inicial (izquierda) y posición autocorregida final (derecha)

La posición obtenida se guardará en un fichero [1] para ser cargada cada vez que se desee utilizar la cámara como elemento de referencia para localizar a personas dentro del entorno parcialmente desconocido en el cual se esté trabajando. Finalmente, y aunque resulta obvio, es conveniente destacar que este procedimiento sólo deberá realizarse cada vez que la cámara o el trípode sean desplazados, por lo

que se procurará, en la medida de lo posible, evitar que eso ocurra, salvo cuando sea estrictamente necesario.

4.3.3. Ubicación de personas en escena. Generación de coordenadas

Una vez que las personas en escena han sido detectadas y se han obtenido puntos relevantes (centroides de sus contornos) que ayuden a completar la tarea de seguimiento de las mismas a lo largo de su movimiento en el entorno, hay que convertir las coordenadas bidimensionales, publicadas en el tópico "people_track/centroids" (tal y como se especificaba en el apartado 4.3.1) a tridimensionales, referenciadas al sistema de coordenadas de la cámara. Para ello, se utilizará la API de Mesa SR, escrita en C++ por el propio fabricante y que se puede consultar en detalle en el manual de la cámara (Anexo externo 1). Ya que la cámara, en el momento de ejecución de la aplicación final, se encontrará en modo de adquisición, debido a la ejecución archivo de lanzamiento "sr_eth.launch" del paquete "swissranger_camera", no queda más remedio que modificar los nodos presentes en dicho paquete para que realicen la función conversión de coordenadas bidimensionales a tridimensionales; es por ello que, **si se desea trabajar en el caso de estudio** que se expondrá más adelante, se deben utilizar los archivos fuente del paquete "swissranger_camera" proporcionados en el CD a la hora de compilar, y no los descargables a través de Internet según lo explicado en el apartado 3.2.4.

Para ello, se busca la sección de código en la que aparecen listadas las funciones definidas en la API de la cámara, entre las cuales hay funciones que permiten obtener las coordenadas en X, Y y Z de cualquier píxel de la imagen. En dicha sección, se añade una función, cuyo comportamiento puede resumirse en el flujograma siguiente (ver Figura 41):

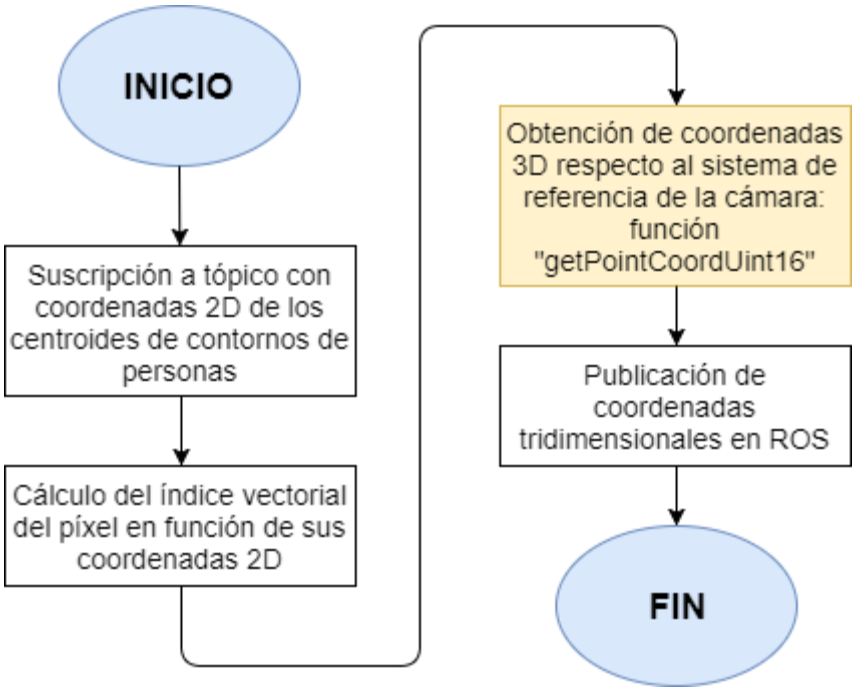


Figura 41: Generación de coordenadas tridimensionales para un píxel de la TOF

La función resaltada en amarillo corresponde a una adaptación de una serie de funciones incluidas por defecto en la API de la TOF, que permiten obtener las coordenadas de cualquier punto tomando como parámetro de entrada un índice; este índice es el que corresponde a cada píxel de la imagen, y que va desde 0 hasta 176x144 - 1 = 25343; la fórmula empleada para obtenerlo (Ec. 5) a partir de las coordenadas bidimensionales procedientes del tópico suscrito, siendo “y” las filas y “x” las columnas (Figura 42), es:

$$index = (centroid.y - 1) \cdot 176 + centroid.x - 1 \qquad (Ec.5)$$

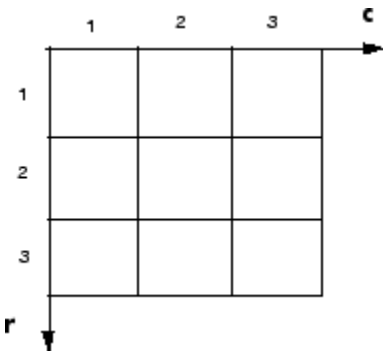


Figura 42: Sistema de coordenadas para los píxeles de una imagen: r (*rows*) y c (*columns*)

Una vez obtenidas las coordenadas, estas son publicadas en ROS y empleadas por un nodo auxiliar, que realiza la simple tarea de publicar una transformada TF entre el *frame* del sensor IMU creado anteriormente (“gyroscope”) y la posición de la

persona. El resultado es la obtención de una posición variable (la de la persona) respecto de un punto de referencia "fijo" (el IMU), siempre y cuando la cámara TOF no se mueva de su ubicación; el resultado puede visualizarse en la Figura 43:

Figura 43: Posicionamiento de dos personas respecto al giroscopio, gracias a la herramienta TF

Para finalizar con este apartado, es conveniente indicar que las coordenadas obtenidas respecto al sistema de referencia de la TOF (Figura 15) se publican en ROS con formato de mensaje propio, definido también en la carpeta *msg* del paquete; así, cada paquete de la aplicación final que deba hacer uso de dicho tópico, entre los cuales se encontrarán paquetes específicos de la navegación del robot [1], deberá incluir en su carpeta *msg* la información relacionada con la definición de este tipo de mensaje propio, para que puedan ser reconocidos como tal e interpretados.

4.3.4. Identificación de personas a partir de características fisionómicas

A la hora de explicar los algoritmos programados en este apartado, que podría considerarse la última de las tareas a resolver en el presente trabajo, se tendrá en cuenta que existen dos fases fundamentales:

- **Fase previa (adquisición y procesamiento de datos)**

En esta etapa, que puede considerarse preparatoria y que tendrá a cabo en el proceso de calibración y toma de datos anterior a la ejecución de la aplicación final, deberán realizarse varias labores:

- Selección y definición del **grupo de personas** que van a participar en el caso de estudio final.

- **Extracción de medidas características** de cada uno de los individuos. Para ello, y teniéndose en cuenta lo dicho en el apartado 4.2 de la presente memoria, se programa el siguiente *script* en Python (Figura 44):

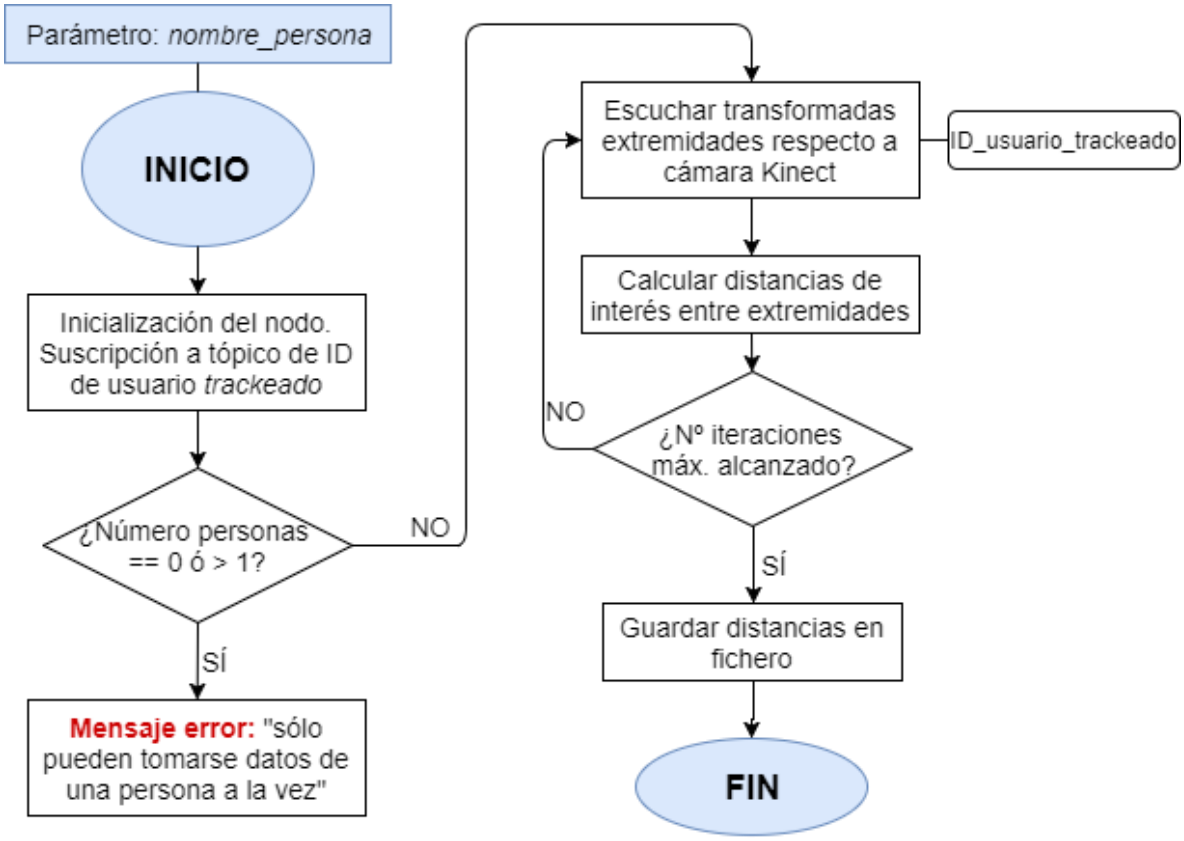


Figura 44: Extracción de medidas características de una persona

El procedimiento explicado en la figura superior, tal y como ya se ha mencionado anteriormente, sirve para extraer las medidas consideradas de interés (cabeza – torso y distancia entre hombros) a la hora de identificar a una persona a partir de sus características fisionómicas. Para ello, será indispensable conocer el nombre de la persona cuyos datos se desean tomar, pasable como parámetro en el momento de lanzar el *script.* Acto seguido, se realiza una suscripción al tópico "/openni_tracker/kinect_tracked_users"; este tópico contiene información sobre los identificadores (números) de las personas que son incluidas en el proceso de *tracking* de este nodo, que debe recordarse realizaba un seguimiento del esqueleto o estructura corporal de las mismas; el motivo de necesitarse este tópico, que **no está publicado en el paquete por defecto** disponible en los repositorios de ROS (el código está ligeramente modificado en el nodo *openni_tracker*, por lo que deberá

siempre compilarse este paquete desde los archivos incluidos en el CD), es que los *frames* o transformadas (Anexo I (III)) de las articulaciones utilizan dicho número, por lo que, para obtener las coordenadas de cada una de ellas, es indispensable conocerlo. Posteriormente, se comprueba que exista sólo una persona en escena, para que los datos no puedan ser falseados, y se comienza a obtener medidas corporales hasta que se alcanza el máximo de iteraciones fijadas (1000 por defecto); en ese momento, los datos se escriben a un fichero, que tendrá como nombre "<nombre_persona>_joints.txt" y que se incluirá en la carpeta *data/person* del paquete.

En caso de que los datos de una persona hayan sido adquiridos con anterioridad, se dará opción a que el fichero que los contenga sea cargado desde el sistema de archivos, para evitar así que dicha persona deba realizar de nuevo el proceso de calibración y adquisición de datos. En principio, no existe un límite de usuarios fijado; aunque, como ya se ha explicado anteriormente, el grupo de personas deberá ser relativamente reducido, ya que en caso contrario puede haber individuos con morfología similar, que dificulten los procesos posteriores de entrenamiento de la red neuronal e identificación de la persona, haciendo uso de dicha red.

- **Cálculo de red neuronal** que permita discernir entre el conjunto de personas definido, de acuerdo a sus características fisionómicas. Para crearla, como ya se citó en el apartado 4.2, se empleará la herramienta **PyBrain**[36], que no es más que un conjunto de librerías que facilitan la implementación de una red neuronal en Python, mediante el uso de funciones y librerías de creación, configuración y verificación. Los tutoriales para instalar esta herramienta, así como breves guías introductorias de uso, se pueden encontrar en su propia página web [36], así como información sobre las redes neuronales, que ya se describieron y explicaron en el apartado 4.2.3 de la presente memoria. en resumen, podría decirse que una red neuronal es un modelo computacional, basado en estructuras atómicas llamadas perceptrones que, mediante sumas ponderadas e interconectándose enter sí, permiten realizar tareas de aprendizaje, clasificación y regresión; en este caso, se debe emplear una red neuronal de clasificación, para lo que se recomienda crear una **FNN** o *feed-*

forward neural network (red neuronal prealimentada), que presentaría una estructura similar a la indicada en la Figura 45.

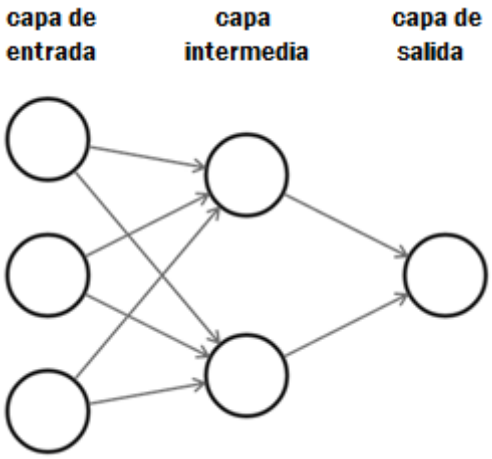


Figura 45: Estructura de una red neuronal prealimentada

Para cumplir con el objetivo marcado, se debe crear una red que tenga tantas capas de entrada como características fisionómicas se vayan a emplear (dos), y que tenga una única capa de salida en la que se indique qué persona es la identificada a partir de las mismas; el número de capas intermedias, en principio arbitrario, se fija en dos. Sabiendo esto, el script **"generate_neural_network.py**" se limita a obtener los datos de entrada de las personas participantes (guardados en la etapa anterior en archivos de texto), añadirlos a una base de datos y, a partir de dicha base de datos, construir y entrenar la red neuronal, de forma que el error que se obtenga sea menor a un máximo especificado del 5%; de no ser así, el entrenamiento prosigue, hasta que una red neuronal adecuada es entrenada y guardada en el fichero **"network.xml"**, en la carpeta *scripts* del paquete "kinect_skeleton_ident". Además, se guardará una tabla *hash* o diccionario con los nombres e identificadores de las personas dentro de la red neuronal (PyBrain asigna a cada conjunto de datos de la base de datos un número único con el que lo identifica, que posteriormente aparecerá a la salida de la red neuronal y será utilizado para saber qué persona se está identificando). Todo lo explicado puede resumirse en el siguiente flujograma (Figura 46):

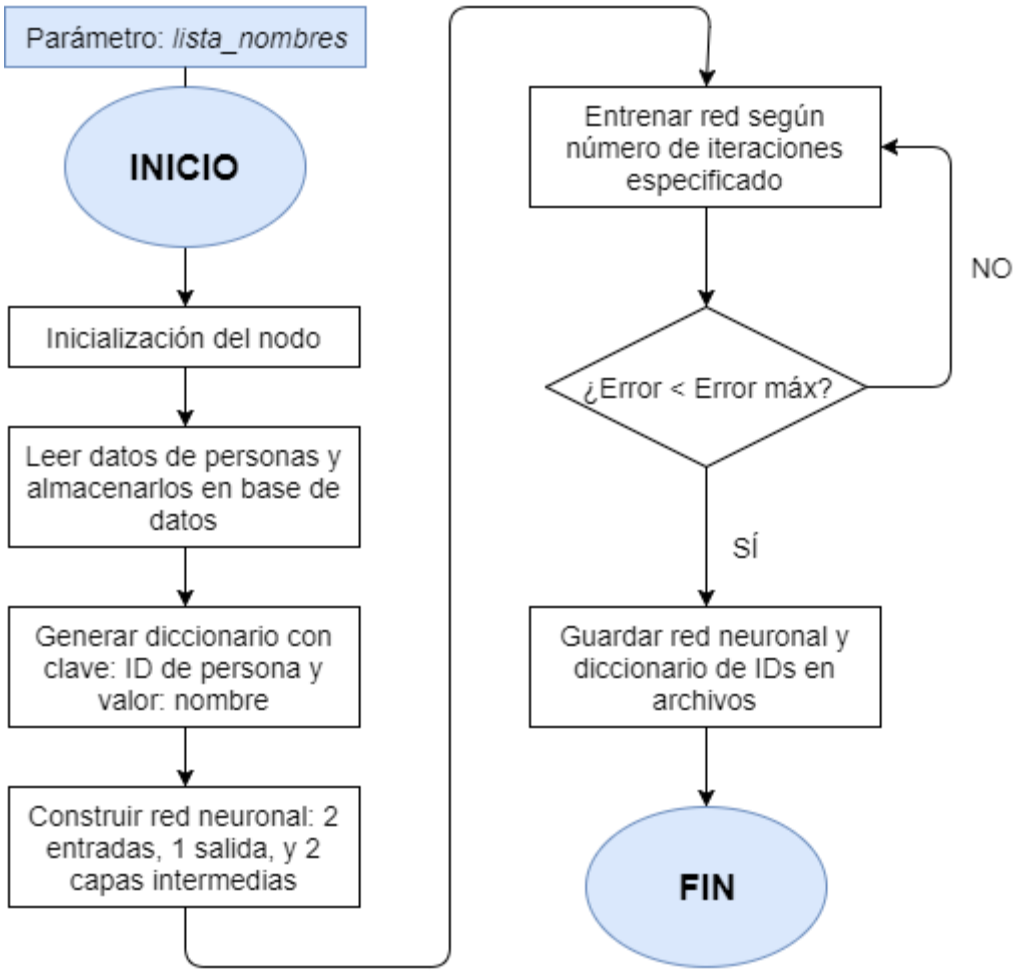


Figura 46: Entrenamiento de la red neuronal encargada de identificar personas según sus características fisionómicas

- **Fase de ejecución (identificación de individuo)**

Una vez adquiridos los datos y entrenada la red neuronal con ellos, se está en condiciones de poder discriminar e identificar a los individuos a partir de sus características fisionómicas en tiempo real; para ello, el individuo deberá efectuar el proceso de calibración delante de la Kinect (ver Figura 22), tras lo cual se procederá a calcular la distancia entre las extremidades consideradas de interés (cabeza – torso y hombros, como ya se ha descrito anteriormente) y a introducirlas como parámetros de entrada a la red neuronal anteriormente generada, que devolverá unos coeficientes ponderados entre 0 y 1 que indican, con mayor o menor precisión, de qué persona se trata; comparándose dichos coeficientes con el contenido de la tabla *hash* o diccionario anteriormente creada, se puede saber de qué persona se trata (Figura 47):

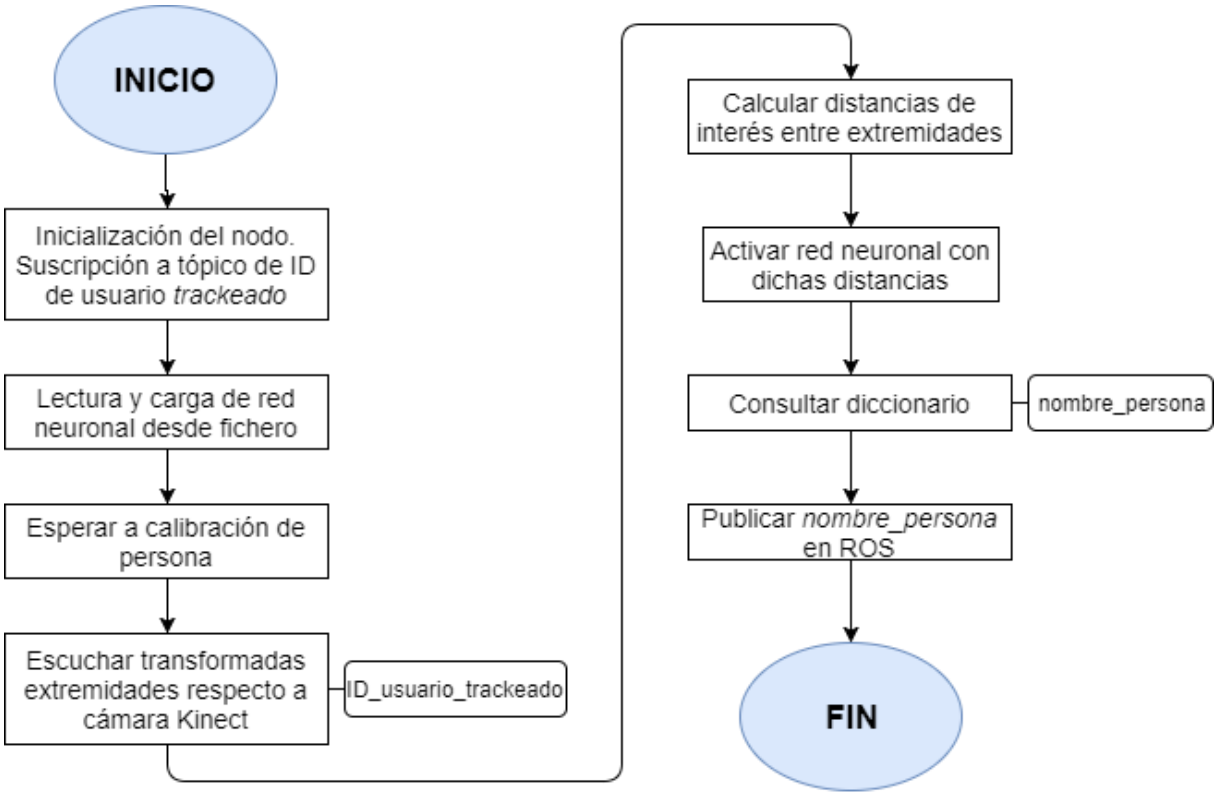


Figura 47: Script encargado de identificar a una persona empleando la red neuronal

- **Interfaz gráfica**

Debido a que todo el proceso de adquisición de datos, generación de la red neuronal y testeo de la misma es complejo, se ha considerado conveniente realizar una interfaz gráfica que facilite acometer tal propósito. Para ello, según lo establecido en el apartado 4.2, se empleará Python y GTK, con la ayuda del diseñador de interfaces gráficas Glade.

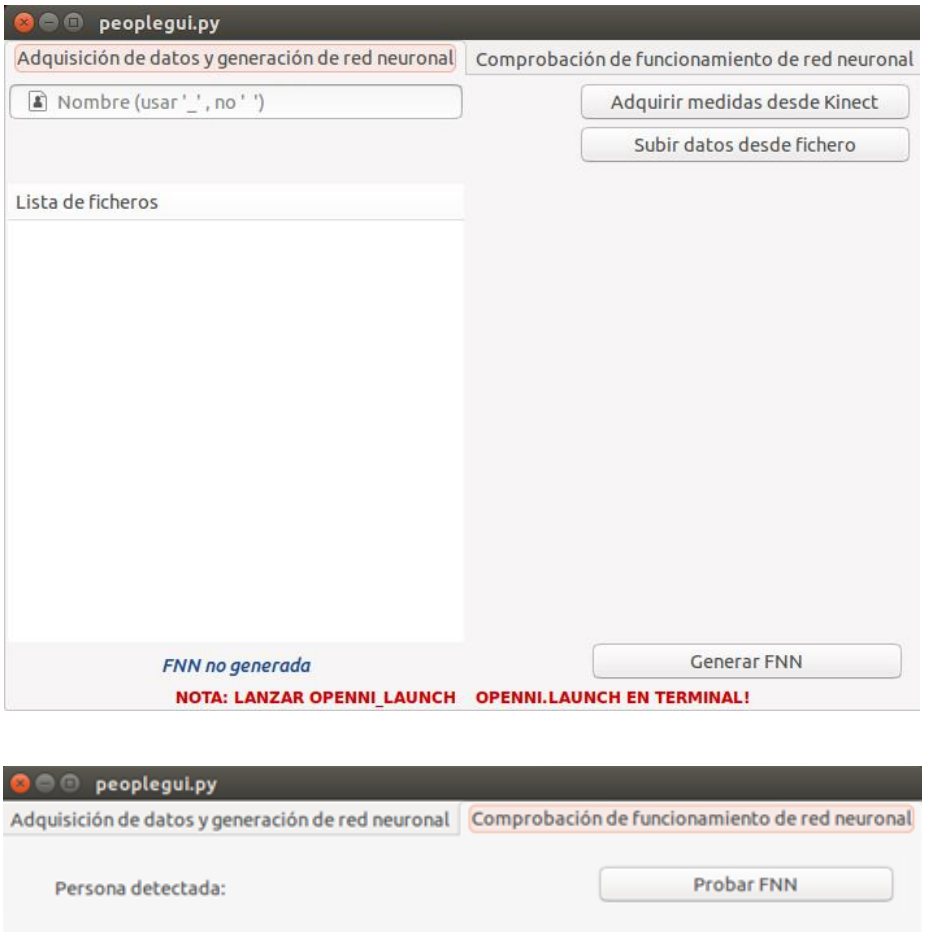


Figura 48: Interfaz gráfica para ayudar a generar la red neuronal

Dicha interfaz gráfica constará de una ventana (Figura 48) con dos pestañas claramente diferenciadas, que se corresponden con las dos fases en las que ya se subdividió la programación de los algoritmos para la identificación de una persona mediante características fisiológicas: adquisición y generación de red neuronal; y comprobación de su funcionamiento.

Para la parte de adquisición y generación, se han incluido los siguientes elementos:

- Caja de texto, en la que se escribirá el nombre de la persona cuyos datos se vayan a tomar. Tal y como se especifica, si el nombre es compuesto, debe usarse como separador el **guión bajo**, y en ningún caso el espacio; lo más recomendable es introducir nombres simples.
- El botón “Adquirir medidas desde Kinect” comenzará el proceso de adquisición de medidas fisionómicas, utilizando para ello la cámara Kinect según lo explicado en el flujograma de la Figura 44. Se han incluido

comprobaciones en el código que evitan que dicho botón, así como cualquier botón en general de la interfaz, pueda pulsarse repetidas veces, para así evitar que se intente lanzar el mismo nodo en múltiples ocasiones, lo que daría lugar a errores.

- Si sólo se desea ampliar el conjunto de personas que van a participar del experimento, no es necesario volver a efectuar una toma de datos de todos y cada uno de los participantes en el mismo, sino sólo de aquellos que sean nuevos en el grupo; para ello, se contempla la posibilidad de **cargar ficheros ya existentes** en la base de datos que luego servirá para entrenar la red neuronal, haciendo uso del botón “Subir datos desde fichero”. Los ficheros, del formato “<nombre>_joints.txt”, deberán introducirse **de uno en uno**.
- En el apartado “Lista de ficheros” aparecerá un listado indicando el nombre de los ficheros donde se hayan guardado los datos de la personas cuya calibración y toma de datos haya sido completada con éxito. Dicha lista, posteriormente, se introducirá como parámetro de entrada al *script* encargado de entrenar la red neuronal y guardarla en un fichero, para su posterior carga y utilización durante la adquisición de datos a tiempo real desde la cámara.
- El botón “Generar FNN” será el encargado de lanzar el código encargado de entrenar y guardar la red neuronal (Figura 46), con los ficheros de las personas presentes en la lista de ficheros anteriormente comentada.

En la segunda pestaña, se ofrece la posibilidad de que pueda comprobarse el correcto funcionamiento de la red neuronal, una vez haya sido entrenada y guardada. Para ello, se obtendrán características fisiológicas de un individuo a tiempo real, y se introducirán como parámetro de entrada en la red neuronal, que identificará a la persona y hará que, en la interfaz gráfica, se muestre su nombre.

Cabe destacar que cada uno de los diferentes procesos se lanzará en una terminal independiente cada vez que se ejecuten, para asegurar la independencia entre sí mismos y entre ellos y la interfaz, que se comporta como un “padre” que suministra los parámetros de entrada y recoge y almacena los parámetros de salida de cada uno de ellos; además, en el código de la propia interfaz, se han incluido

comprobaciones para evitar que, por ejemplo, se abra el mismo *script* dos veces a la vez, o para que todos los *scripts* abiertos se cierren antes de salir de la interfaz.

Con la explicación de uso de la interfaz se da por finalizada la etapa de explicación del código desarrollado para resolver las tareas necesarias para la detección, generación de coordenadas e identificación de una persona en un entorno determinado; acto seguido, se procederá a ilustrar y ejemplificar la utilidad y aplicabilidad de todo lo anterior en un caso de estudio concreto, que muestre el potencial de las herramientas desarrolladas.

5. Caso de estudio

En este punto de la memoria se pretende exponer un ejemplo práctico (también llamado **caso de estudio**) de una situación real en la que todas las utilidades anteriormente explicadas y desarrolladas, y que van encaminadas a obtener información de un entorno parcialmente desconocido para facilitar información a un robot sobre el mismo, se vean empleadas de forma conveniente. Para ello, se colaborará estrechamente con el proyecto de navegación autónoma, fijándose el procedimiento y las pautas a realizar para preparar el escenario de trabajo, así como las tareas concretas que ayuden a ejemplificar los objetivos perseguidos.

Los algoritmos desarrollados no tendrían sentido si no tuviesen una aplicabilidad y objetivos específicos. Para ilustrarlos y facilitar su comprensión, se propone el siguiente caso de estudio, cuyo objetivo principal es **"detectar a una persona en un área predefinida y movilizar la plataforma robótica hasta su posición para que la identifique e interactúe con ella"**.

El primer paso para definir el caso de estudio es describir adecuadamente el entorno de trabajo, que será un área despejada, de aproximadamente doce metros cuadrados de superficie, en la que tanto robot como personas podrán moverse libremente; se procurará que no existan grandes obstáculos que impidan el movimiento autónomo de la base robótica móvil PMB-2, prestándose especial atención a aquellos que quedan fueran del rango de visibilidad del láser, ya que este se encuentra a una cierta altura con respecto al suelo (10 -15 centímetros aproximadamente) y no es capaz de detectar ciertos objetos que se encuentran por debajo del sensor, lo que hace que exista riesgo de colisión. Además, la iluminación debe ser adecuada (no muy tenue ni tampoco muy intensa) para el correcto funcionamiento de las cámaras.

Siendo la cámara de tiempo de vuelo uno de los sensores empleados más delicados, cabe recordar que, en la medida de lo posible, conviene evitar la presencia de superficies reflectantes (ver apartado 3.2.2), ya que interfieren en la imagen de profundidad captada por la misma; a pesar de ello, se observó en el apartado 4.2 de esta memoria que el suelo del entorno de trabajo poseía un alto índice de reflectividad, con lo cual se decidió trabajar con información bidimensional para desarrollar el

algoritmo de detección de personas y prescindir de un algoritmo basado en la PCL y nubes de puntos, que necesitaba la presencia de un plano de suelo regular.

Ya que la cámara TOF debe ser capaz de detectar la presencia de personas en el **mayor rango posible**, se situará en una esquina de la escena, montada sobre el trípode y con el sensor IMU convenientemente colocado sobre ella. Como el resto de los elementos (robot móvil PMB-2, robot humanoide Meka y sensor Kinect) se encuentra formando un único bloque, se fijará una **situación de reposo**, que interfiera mínimamente con el desarrollo de los acontecimientos que formarán parte de la demostración final. Un esquema de lo que podría ser la situación inicial y final de todos los elementos que interactúan en el escenario del caso de estudio es el ilustrado en la Figura 49. En el mismo, se observa que la posición inicial del robot interfiere mínimamente con el desarrollo inicial del caso de estudio (persona caminando en escena); mientras que su posición final se define frente a la persona, entre ella y la cámara TOF de detección. La posible ruta que podría seguir el robot autónomo se dibuja en magenta en la Figura 49.

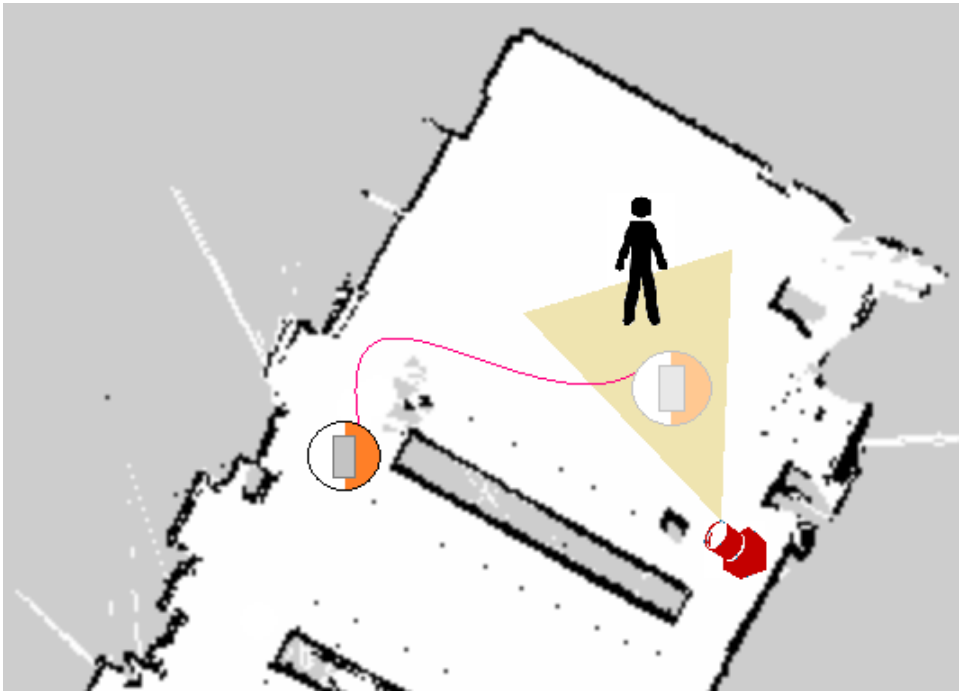


Figura 49: Situación de los elementos conformantes de la plataforma desarrollada en el entorno de trabajo: posición inicial y final del robot (círculo), persona y cámara TOF

Un esquema que resume brevemente las tareas principales que deben cumplir cada uno de los elementos del sistema, con sus características y el flujo de

información entre ellos, podría ser el mostrado en la Figura 50. En él, cada elemento realiza una tarea concreta, a la vez que comparte información con el resto:

- La cámara de tiempo de vuelo (TOF) se encarga de posicionar a los humanos en escena, tarea para la cual recibe asistencia del robot móvil PMB-2 para autolocalizarse (ver apartado 4.3.2) en el mapa que le proporciona.
- El sensor Kinect se encuentra fijado en el extremo superior del robot humanoide Meka, que le confiere una posición fija en el árbol de transformadas espaciales. Dicha cámara se emplea para identificar a la persona que se encuentra frente al robot, de forma que éste pueda interactuar con ella.
- Por su parte, los robots tiene una estrecha relación entre sí, ya que el robot Meka va montado sobre la base móvil PMB-2, que dota al primero de movilidad traslacional (y no sólo articular), así como le otorga una posición concreta en el mapa.

Para realizar estas tareas todos los elementos, tanto sensores (cámaras) como actuadores (robots) intercambian información entre sí y con la plataforma software de control y comunicación empleada para gestionar el sistema, que no es otra que **ROS**.

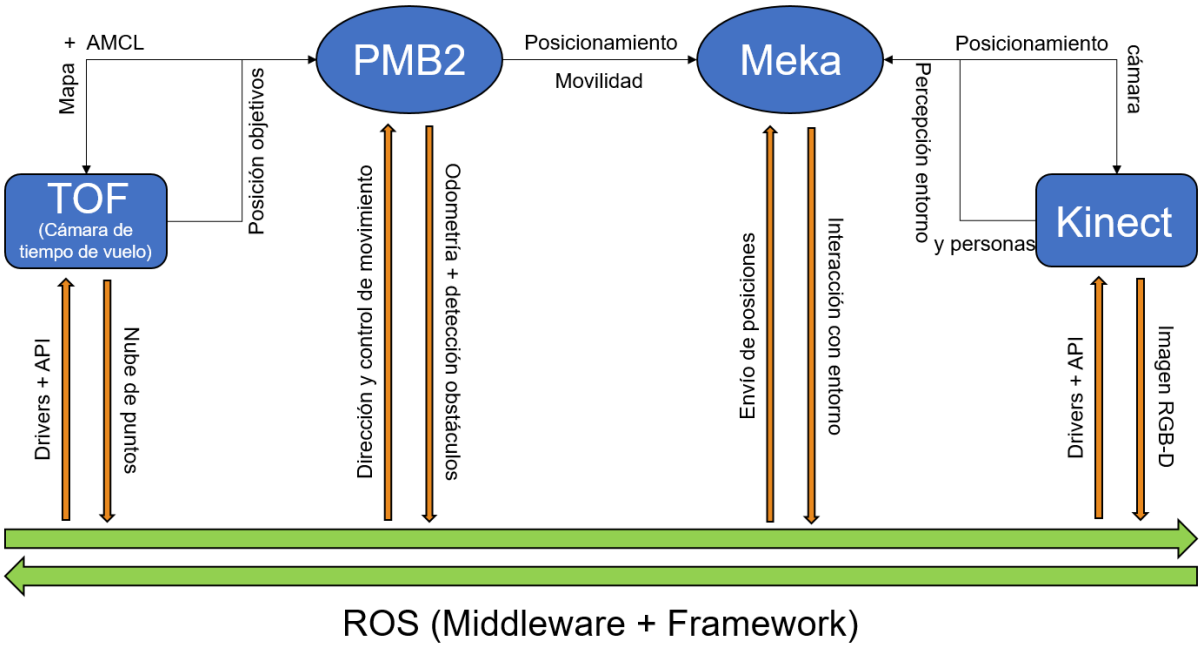


Figura 50: Esquema del caso de estudio

Una vez explicado el flujograma de intercambio de información y reparto de tareas entre los diferentes elementos que conforman la plataforma robótica, se procede a tratar aspectos como el conexionado de red entre los diferentes ordenadores, robots y sensores, así como la forma en la que se suministra energía de alimentación a los mismos. Para ello, se ha tenido en cuenta que el robot móvil debe tener una independencia total de movimiento, lo cual obliga a que tanto la plataforma móvil PMB-2 como el robot humanoide Meka (y su ordenador) deban alimentarse exclusivamente con baterías, así como la cámara Kinect. Además, tendrá que existir una conexión inalámbrica de red entre este bloque y el resto de elementos de la plataforma, que es generada por el propio robot PMB-2.

Con todo esto en mente, un esquema que refleja el diagrama de interconexionado y alimentación de todos los elementos es el que aparece en la :

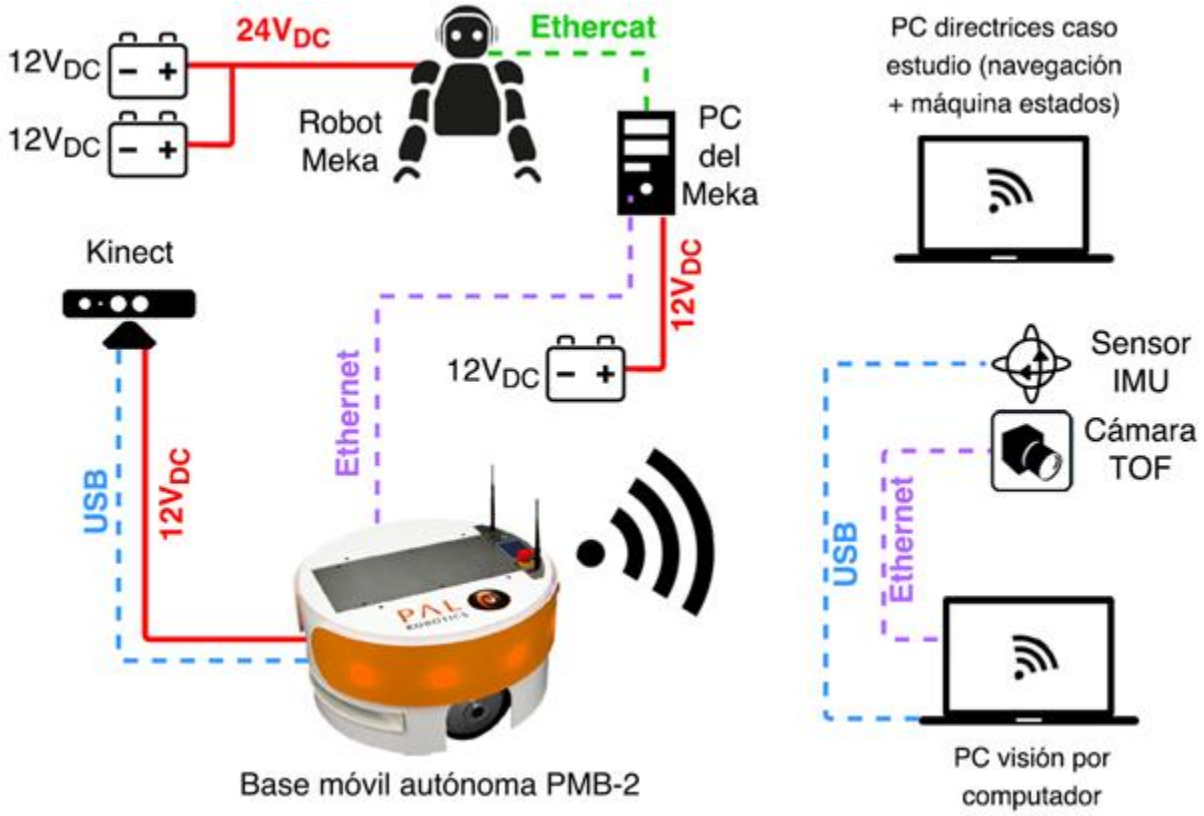


Figura 51: Esquema de conexionado de red y de alimentación de la plataforma robótica desarrollada

De un simple vistazo al esquema, se puede observar que el principal centro de conexiones es la base móvil autónoma PMB-2, puesto que se conecta por Ethernet al ordenador que controla el robot Meka y por USB a la cámara Kinect, además de

proporcionar la tensión e intensidad de alimentación necesarios a esta última, gracias a su batería de gran capacidad.

Los demás ordenadores participantes de la plataforma se conectarán a la red global mediante conexión inalámbrica WiFi, empleando el punto WiFi creado por el robot PMB-2. De acuerdo con el modelo OSI (*Open System Interconnection* o modelo de interconexión de sistemas abiertos [ISO/IEC 7498-1]), el conexionado mediante Ethernet o WiFi resulta ser indiferente. Esto es así porque el protocolo de red de arquitectura en capas, creado en 1980 por la ISO (*International Standards Organization)*, separa la capa física de las demás, lo que permite que la asignación de IPs sea idéntica, tanto por Ethernet como por WiFi.

Que todos los dispositivos se conecten de forma similar a la red no quiere decir que tengan la misma categoría jerárquica en ROS, como bien ilustra el esquema de la Figura 52.

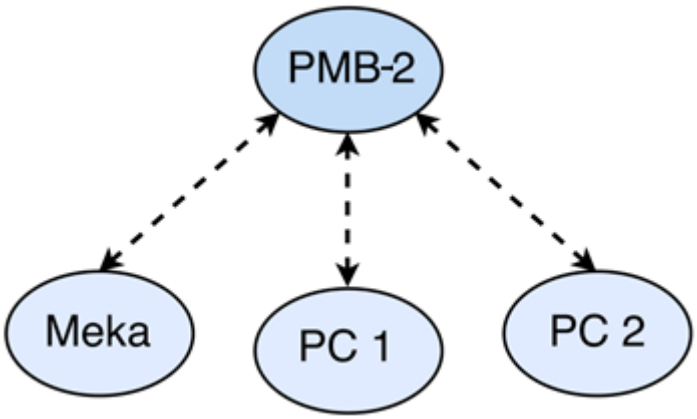


Figura 52: Esquema jerárquico de la plataforma desarrollada, con el robot PMB-2 como *master*

Como puede observarse, y a pesar de que cualquiera de los equipos conectado a la red podría ser el *rosmaster*, se selecciona para esta tarea el sistema con mayor poder de computación, y que esté más libre de carga (información consultada en [2]); dicho ordenador resulta ser el del PMB-2, que coincide con el centro neurálgico de la red de conexionado.

El resto de ordenadores deberán indicar en su *bash* que el *master* es el robot PMB-2, identificable mediante su dirección IP. Para ello, en cada uno de los ordenadores en los que se vayan a ejecutar nodos, deberán incluirse las siguientes líneas en sus archivos **.bashrc**:

```
$ export ROS_IP=<IP DEL PROPIO ORDENADOR>
$ export ROS_MASTER_URI=http://<IP DEL ORDENADOR PMB-2>:11311
```

Para que la aplicación final, descrita mediante una máquina de estados finitos en [1], pueda ejecutarse, debe seguirse el siguiente procedimiento de ejecución de tareas y de arranque:

- Entrenamiento de la red neuronal para identificar personas a partir de sus características fisionómicas.
- Mapeado del entorno.
- Autocalibrar la posición de la cámara TOF respecto del mapa y guardarla.
- Iniciar plataforma con archivos de lanzamiento alojados en el PC del *master* (PMB-2):

    - Navegación en PMB-2, carga de URDF (Meka + Kinect) y arranque del Meka [1].
    - Inicialización de la Kinect (*openni_launch* y *openni_tracker*) en el PMB-2, para gestionar la Kinect, así como el nodo *sound_play,* para poder escuchar las directrices de voz enviadas por el sistema mediante el altavoz acoplado al robot.
    - Lanzamiento de la posición de la TOF respecto al mapa y de nodos auxiliares al caso de estudio [1].

- Lanzamiento de la cámara TOF, junto con el sensor IMU y los nodos de generación de transformadas cámara – giroscopio – personas.
- Inicialización del nodo encargado de efectuar las tareas de visión por computador, empleando las imágenes proporcionadas por la cámara TOF, para detectar a las personas en escena a partir de su silueta.
- Carga y uso de red neuronal entrenada que identifique a los usuarios, haciendo uso del sensor Kinect.
- Script SMACH [1] que dirija la máquina de estados finitos encargada de supervisar la evolución en el caso de estudio.

Para que resulte lo más sencillo posible, el Anexo V recoge, paso a paso, todas las instrucciones necesarias para la puesta a punto de los elementos,

paquetes y programas necesarios para que, desde una instalación limpia de Ubuntu, pueda ejecutarse el caso de estudio sin mayores problemas en cualquier PC. Es importante, como ya se ha mencionado anteriormente, destacar **que no se recomienda el uso de máquinas virtuales** en el ordenador que vaya a ejecutar las tareas de visión por computador, porque suponen una gran carga para el sistema; además, y aunque en el momento de la ejecución el sensor Kinect vaya conectado al robot PMB-2, es importante saber que dicho sensor no funcionará en sistemas virtualizados.

6. Conclusiones

La realización del presente proyecto ha resultado, en líneas generales, satisfactoria, pudiéndose cumplir los objetivos marcados al inicio del mismo.

Las personas se han conseguido detectar en escena, pese a las dificultades marcadas por el propio entorno, gracias al desarrollo de algoritmos propios de visión por computador y a las bondades que ofrecen las cámaras de profundidad. No obstante, se plantea, como trabajo futuro, conjugar esta tarea con el uso de una cámara de mayor resolución, que permita extraer características de los contornos presentes en escena de una forma más precisa.

El proceso de autolocalización de la cámara en el entorno, una de las tareas más complejas de automatizar, se ha resulto satisfactoriamente gracias al empleo de la unidad sensorial de medición inercial, que ha permitido que el proceso se realice de forma sencilla y sin necesidad de realizar tediosas medidas en el mapa. Además, dicho sensor ha servido como punto de referencia para ubicar a las personas en escena, previa obtención de sus coordenadas con la cámara de profundidad.

El sensor Kinect ha resultado ser una herramienta excelente a la hora de identificar personas mediante métodos alternativos al reconocimiento facial, empleándose para ello características morfológicas; no obstante, se propone como tarea futura mejorar el algoritmo de detección, incluyendo más mediciones del cuerpo o efectuando análisis morfológicos del rostro de las personas para disponer de una herramienta de detección combinada más robusta.

Para finalizar, se pretende realizar una reflexión en voz alta, considerándose que, a pesar de la gran cantidad de dificultades encontradas durante el desarrollo del proyecto, como falta de *drivers* o carencias de programación, se han conseguido solventar satisfactoriamente, disponiéndose en la actualidad de una idea mucho más amplia acerca de lo que es ROS y de cómo se trabaja con los sensores para procesar los datos que ofrecen y transformalos en información útil para un robot. Los sensores utilizados en este proyecto simplemente constituyen una selección personal, y se propone, como tarea futura, ampliar el catálogo para dotar a la plataforma de nuevas e interesantes funcionalidades.

Anexo I : Información adicional sobre ROS

I. Instalación de ROS

Los pasos necesarios para la instalación de ROS en Ubuntu vienen explicados en la web oficial de la plataforma [2]; no obstante, se resumen aquí los pasos a seguir, teniéndose en cuenta que la versión de Ubuntu elegida es la 14.04LTS y la versión de ROS a instalar es la ***Indigo***:

- Configurar el ordenador para aceptar paquetes de *software* de ROS:

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

- Establecer las claves de ROS (sustituir por “hkp://pgp.mit.edu:80” ó “hkp://keyserver.ubuntu.com:80” si el comando falla):

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
--recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

- Actualizar el índice de paquetes Debian:

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

- Una vez finalizado todo lo anterior, se procede a instalar ROS. Se recomienda instalar la versión completa:

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

- Cuando la instalación finalice, y antes de poder utilizar ROS, inicializar el *rosdep*:

```
$ sudo rosdep init
$ rosdep update
```

- Añadir las variables del entorno de ROS automáticamente al ***bash*** cada vez que se abre una nueva terminal de comandos:

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

Para finalizar con este punto, y evitar posibles problemas en el futuro, es **importante** destacar el hecho de que las variables de entorno de los ***workspaces***

deben añadirse al *bash* de la terminal en la que se vayan a utilizar paquetes pertenecientes a dichos *workspaces*; para ello, se debe ejecutar el siguiente comando:

```
$ source <ruta_al_workspace>/devel/setup.bash
```

Si se desea automatizar este proceso, para que las variables de un *workspace* determinado se añadan al inicio de cada sesión en terminal, la línea de código anterior deberá incluirse en el archivo "**.bashrc"**, dentro de la carpeta personal; una forma fácil de editarlo en cualquier momento es:

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

II. Paquetes y workspaces

Los paquetes son la unidad básica más importante que maneja ROS. Internamente, deben tener una estructura específica, para evitar confusión al compartirlos con la comunidad para su posterior reutilización. La creación de los mismos resulta sencilla, gracias a herramientas proporcionadas por la plataforma, que con un simple comando permiten crear un paquete y declarar sus dependencias (que son fácilmente modificables):

```
$ catkin_create_pkg <nombre_paquete> <dependencia1> <dependencia2> <...>
```

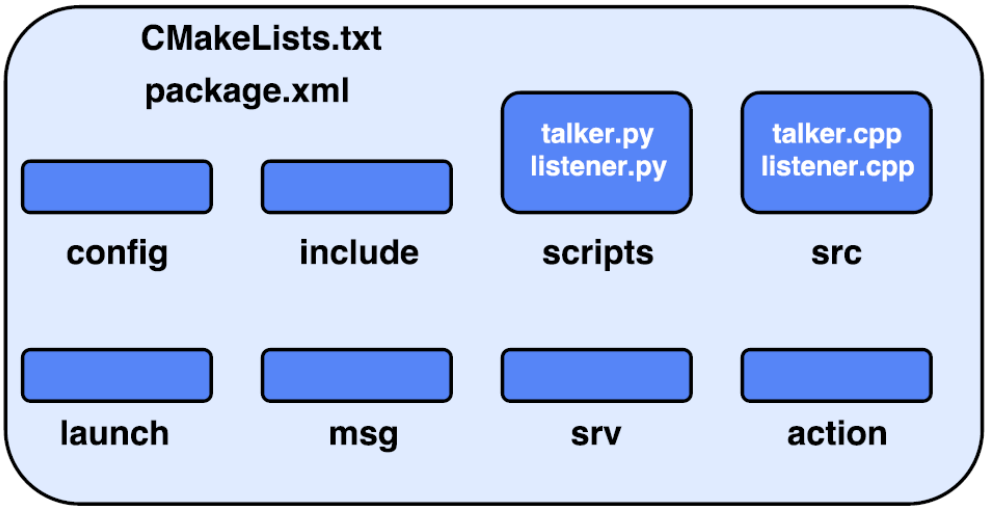


Figura 53: Estructura típica de un paquete ROS

El primer requisito para crear un paquete es crear un ROS catkin workspace, que es el directorio principal en el que compilar, enlazar librerías y ejecutables… a partir de otros archivos estructurados de ROS, como los paquetes. Para ello, se ejecutan los siguientes comandos:

```
$  mkdir -p ~/<nombre_workspace>/src
$  cd ~/<nombre_workspace>/src
$  catkin_init_workspace
```

Dentro de la carpeta “src” del workspace ya creado se ubicarán los paquetes de ROS, cuyos nodos se escriben en Python o C++; en el caso de Python, se ubicarán en la carpeta “scripts”, y sólo será necesario dar permisos de ejecución a dichos scripts “.py” una vez, mediante los siguientes comandos:

```
$  cd ~/<nombre_workspace>/src/<nombre_paquete>/scripts
$  sudo chmod u+x *.py
```

Si la programación se realiza en C++, además de la propia edición del código, se deberán editar los archivos “CmakeLists.txt” y “package.xml” del paquete, compilando después:

```
$  cd ~/<nombre_workspace>
$  catkin_make
```

Por último, se deben referenciar los workspaces en el “shell” de Ubuntu, para que el sistema de archivos de ROS reconozca los nodos y puedan ejecutarse; para que la referencia sea permanente, se deben escribir los siguientes comandos cada vez que se cree un workspace:

```
$  echo “source ~/<nombre_workspace>/devel/setup.bash” >> ~/.bashrc
$  source ~/.bashrc
```

III. TF

Un gran problema en la robótica que además afecta a multitud de tareas a realizar es la de la localización espacial de todos los elementos en acción. Para ello, debe haber una forma adecuada de gestionar y manejar el uso de coordenadas 3D en el tiempo, tara para la cual ROS incluye el paquete **TF** [27], básico para el correcto funcionamiento de cualquier plataforma robótica. Por lo tanto, usando TF no sólo se gestiona la dimensión espacial si no también la temporal. Ya que es un paquete de ROS, puede operar en un sistema distribuido, en diferentes unidades; es por ello por lo que la sincronización entre los relojes de todos los sistemas es crucial para evitar problemas de coordinamiento entre ellos.

TF consta de unos objetos, llamados *frames,* los cuales pueden referenciarse unos a otros mediante transformadas espaciales, que vienen definidas por una posición y una orientación concretas; por lo tanto, es muy importante saber la **jerarquía** existente entre dichos *frames*, ya de ello depende que el resultado final sea adecuado o no, según las necesidades pertinentes. Ya que la estructura es jerárquica, los elementos de cualquier árbol de transformadas deben tener un solo *frame* padre, pero pueden tener varios *frames* hijos; esto quiere decir que existe un sistema de referencia, en la parte superior del sistema jerárquico, al cual se referencian todos los puntos del sistema definido; esa es la forma que tiene TF de conocer la posición global de cualquier objeto en un instante determinado. En el caso de estudio tratado en esta memoria, debido a que el objetivo era facilitar la interacción en un entorno parcialmente desconocido entre robots y personas, se ha utilizado como punto de referencia global el mapa creado en [1].

Algunas de las relaciones existentes entre *frames* serán estáticas, como por ejemplo la posición que ocupa la cámara Kinect respecto al robot; dentro del contexto de este proyecto; otras, por el contrario, serán dinámicas, como la posición que ocupa la persona en un instante determinado respecto al propio robot.

Para visualizar el árbol jerárquico de las transformadas existentes en un momento concreto se utiliza la herramienta **rqt_tf_tree**, mientras que si se desean visualizar los frames de forma gráfica se empleará la herramienta **Rviz**.

Anexo II : Comandos para la inclusión del sensor Mesa SR4000 en la plataforma ROS

Para que este sensor pueda funcionar adecuadamente en ROS, se necesita convertir la información en bruto ofrecida por la TOF en mensajes entendibles por la *framework* en cuestión; para ello, se debe efectuar la instalación de la plataforma "Open_Ptrack" en ROS.

Teniéndose en cuenta que sólo se pretenden instalar los drivers de la Mesa SR4000, se deberán realizar varios pasos:

- Abrir una terminal de Ubuntu y crear un *workspace* de ROS, según los pasos indicados en el **Anexo I (II)**. No es necesaria la creación de ningún paquete, aunque sí es recomendable, como se explicaba en el anexo anteriormente mencionado, incluir la ruta del workspace en el archivo "~/.bashrc" de la carpeta personal.
- Navegar hasta la carpeta "src" (*source*) de dicho workspace:

```
$ cd ~/<nombre_workspace>/src
```

- Clonar los archivos fuente desde el repositorio de "GitHub" de "Open_PTrack":

```
$ git clone https://github.com/OpenPTrack/open_ptrack.git
```

(**Nota:** en caso de que, en el momento de la instalación, dicho repositorio se encuentre inactivo, se incluye una copia del código fuente en los archivos adjuntos de la presente memoria.)

- Navegar hasta el paquete clonado, de nombre "open_ptrack":

```
$ cd open_ptrack
```

- Eliminar directorios innecesarios para liberar memoria:

```
$ rm -rf bayes detection docs open_ptrack opt_calibration opt_msgs
opt_utils scripts tracking
```

- Dar permisos a los archivos de configuración:

```
$ chmod 755 swissranger_camera/cfg/SwissRanger.cfg
```

(**Nota:** podría darse el caso, tanto en este como en otros futuros comandos, de que al ejecutarlos se obtenga un error de permisos; en ese caso, añadir, al principio del comando, la palabra “sudo”, seguida de un espacio, para otorgar permisos de superusuario.)

- Navegar hasta el directorio raíz del workspace y compilar:

```
$ cd ../../..
$ catkin_make
```

Una vez compilado el paquete (puede tardar bastante), los drivers y la API de ROS para la cámara Mesa SR4000 están instalados; no obstante, debido a que el **caso de estudio** requiere de la modificación de algunos archivos fuente de Open_PTrack, se debe, si se desea trabajar en el caso de estudio, **reemplazar la carpeta** “swissranger_camera” instalada por defecto en el proceso por la presente en el CD adjunto a este proyecto, con el mismo nombre.

Anexo III : Comandos para la inclusión del sensor Kinect en la plataforma ROS

Antes de comenzar a explicar con el proceso de instalación, es necesario recalcar que, tal y como se indicó en la introducción de este trabajo, estos pasos están concebidos para su ejecución en un ordenador con Ubuntu instalado como sistema nativo en disco duro, y **no mediante una virtualización**; de no ser así, el sensor Kinect no podría funcionar correctamente.

En primer lugar, deberá instalarse el *framework* OpenNI:

```
$ sudo apt-get install libopenni0 libopenni-dev
```

Después, se procederá a instalar los diferentes paquetes que permiten la interacción entre OpenNI y ROS; para ello, teniendo en cuenta que se está usando la versión “Indigo”:

```
$ sudo apt-get install ros-indigo-openni-*
```

El uso del asterisco implica que se instalarán todos los paquetes de ROS cuyo nombre comience por “openni”, tales como **openni_launch** y **openni_tracker**.

Para continuar con la instalación, deberá descargarse el siguiente repositorio (nota: si el siguiente comando no funciona, instalar previamente el programa “git” mediante “apt-get”); también se incluye una copia del mismo en el CD adjunto:

```
$ git clone https://github.com/avin2/SensorKinect
```

Tras ello, navegar a la carpeta clonada y, dentro de ella, al directorio “Bin”. En este directorio se encuentran unos drivers específicos para el sensor Kinect, que deberán instalarse en función de si se dispone de un sistema operativo de 32 o 64 bits; para el caso de 64 bits:

```
$ tar xjf SensorKinect093-Bin-Linux-x64-v5.1.2.1.tar.bz2
$ cd Sensor-Bin-Linux-x64-v5.1.2.1
$ sudo ./install.sh
```

Finalmente, y para el correcto funcionamiento del *tracker* de estructura morfológica de personas, los *drivers* NiTE deberán ser descargados, descomprimidos e instalados mediante el comando:

```
$ sudo ./install.sh
```

La versión recomendada para NiTE es la 1.5.2.23, que puede ser encontrada en los archivos adjuntos de la presente memoria y descargada en el siguiente enlace: http://www.openni.ru/openni-sdk/openni-sdk-history-2/. Hay que destacar que, si se desea hacer un uso de la cámara Kinect en el **caso de estudio**, para identificar personas a partir de sus características morfológicas, es necesario instalar una **versión modificada** del paquete ***openni_tracker***; para ello, instalar en el ordenador desde el cual se vaya a ejecutar el contenido de dicho paquete la versión modificada de *openni_tracker*, copiando la carpeta de mismo nombre contenida en CD adjunto al presente documento en la carpeta *src* de un *workspace* cualquiera, y compilando dicho *workspace* con el comando “catkin_make”. Si se desea usar la Kinect acoplada al robot PMB-2, dicho paquete deberá renombrarse a *openni_tracker2*, para evitar incompatibilidades entre versiones.

Anexo IV : Proceso de instalación y puesta a punto de la plataforma Open_PTrack" en ROS para su utilización con la cámara Mesa SR4500

Open_PTrack es un proyecto *open-source* empleado para el seguimiento o *tracking* de personas en una escena, utilizando para ello algoritmos de visión por computador basados, fundamentalmente, en el tratamiento de información tridimensional del entorno (nubes de puntos) mediante el uso de la PCL. Ya que los sensores de Mesa Imaging proporcionan información tridimensional del entorno, Open_PTrack los considera en su lista de sensores compatibles y da soporte a uno de ellos, en concreto el modelo más reciente (SR4500). En esta breve guía, se explicarán los pasos que se siguieron para que Open_PTrack pudiese funcionar con dicho sensor en un entorno específico, que no es otro que aquel donde se desarrollaron las pruebas del caso de estudio redactado en esta memoria.

Para la instalación de Open_PTrack [38] deben seguirse los siguientes pasos:

- Clonar el repositorio desde GitHub:

```
$ sudo apt-get install git -y
$ git clone https://github.com/OpenPTrack/open_ptrack.git
```

- Instalar ROS (este paso puede omitirse si la instalación ya se ha realizado, que es lo más habitual). Si aún no se ha instalado, pueden consultarse los pasos en [38].
- Instalar Open_PTrack:

```
$ ./openptrack_install.sh
$ rm -rf ~/open_ptrack
$ ln -s  ~/workspace/ros/catkin/src/open_ptrack ~/open_ptrack
```

Una vez que Open_PTrack está instalado, se procede a conectar la cámara Mesa SR4500 a ROS, previa asignación de la IP según lo explicado en el apartado 3.2.4. Después, ejecutar:

```
$ roslaunch opt_calibration sr4500_calibration.launch device_ip:=192.168.48
  .141 size:=7x10 square:=0.0255 camera_id:=SwissRanger
```

Dicho comando contiene una serie de parámetros que deben explicarse; el primero de ellos es la IP de la cámara, anteriormente asignada. El segundo, llamado “size”, es el tamaño de la cuadrícula en tablero de ajedrez (Figura 54) que se va a utilizar para efectuar el proceso de calibrado de la cámara, teniéndose en cuenta que los números representan el número de aristas comunes en cada lado, es decir, el número de cuadros menos uno. El tercero, llamado “square”, representa el tamaño, en metros, de cada uno de los cuadros de dicha cuadrícula; por último, se indica el nombre de la cámara.

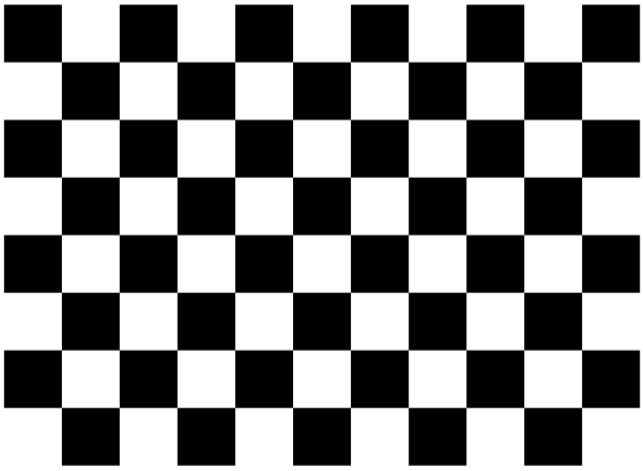


Figura 54: *Checkerboard* utilizado para calibrar la cámara TOF

Con todos estos pasos, aparecerá una interfaz para calibrar la cámara TOF, proceso el cual se llevará a cabo situando el tablero de ajedrez delante de la cámara y moviéndolo hacia adelante y detrás, hacia los lados y efectuando giros, para que la calibración sea lo más precisa posible. Cuando todos los indicadores de la Figura 55 estén en verde, se puede pulsar el botón de calibración y, acto seguido, el de guardado de datos, que generará un archivo llamado SwissRanger.yaml, localizado en la carpeta *openp_track/opt_configuration/camera_info*. Dicho archivo debe copiarse en la ruta *~/.ros/camera_info/SwissRanger.yaml*, para posteriormente ejecutar el comando que dará lugar al comienzo del proceso de *tracking* de personas:

```
$ roslaunch tracking detection_and_tracking_sr.launch
```

Figura 55: Interfaz de calibración de la cámara TOF

Es **muy importante** reseñar que, a menos que el suelo no sea reflectante, Open_PTrack no funcionará; el motivo de que esto ocurra es que la cámara de profundidad trabaja enviando haces de luz que luego recibe y con los que calcula la distancia al entorno, con lo que una superficie reflectante provoca que se falseen las mediciones y aparezca en escena un suelo irregular (Figura 29), que impide que la PCL encuentre un plano adecuado sobre el que detectar elementos, como personas. Esto mismo ocurría en el entorno del caso de estudio, por lo que para conseguir lanzar adecuadamente Open_PTrack el suelo tuvo que cubrirse con lonas antirreflectantes.

Una vez aclarado todo esto, y tras ejecutar el comando anteriormente citado, aparece una interfaz en Rviz en la que, cada vez que una persona entra en escena, se genera un “Pose” o posición en rotación y traslación, que representa la posición de la persona respecto al sistema de referencia de la propia cámara. En la Figura 56, cada ruta de color representa la traza que ha dejado una persona diferente al caminar por el escenario visible por la cámara, que es el que se ve en las imágenes de la izquierda.

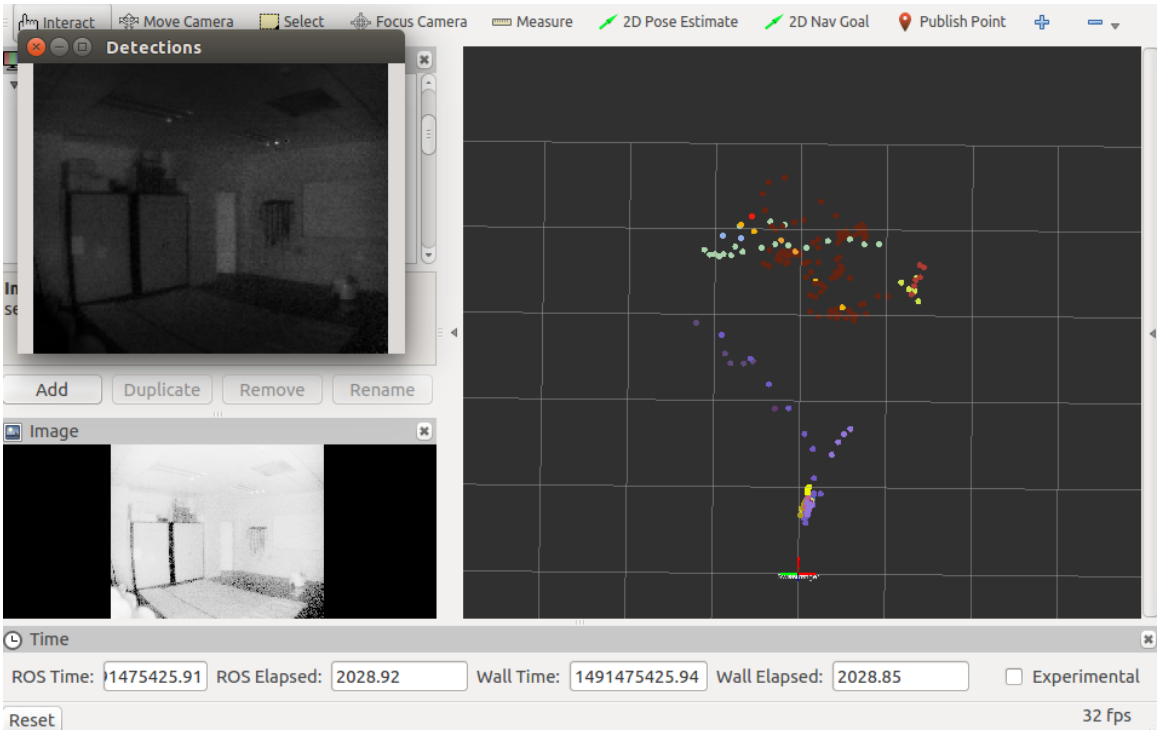


Figura 56: Datos generados por Open_PTrack sobre el seguimiento de personas y su representación en Rviz

Con esto, pues, queda explicado el procedimiento para la ejecución de Open_PTrack y su puesta en marcha para el sensor Mesa SR4500; en un entorno favorable, esta información puede utilizarse como base para desarrollar algoritmos de *tracking* más robustos y eficientes.

Anexo V: Puesta a punto de un ordenador con Ubuntu para la ejecución en el mismo de las aplicaciones desarrolladas para el caso de estudio

Los pasos a seguir para la instalación de todo lo necesario para ejecutar el caso de estudio en un PC con Ubuntu 14.04 instalado desde una partición limpia son:

1. Instalar ROS, tal y como se especifica en el Anexo I (I) de la presente memoria.
2. Instalar los *drivers* de Mesa Imaging, desde la web oficial del fabricante, como se explicó en el apartado 3.2.4. Para mayor comodidad, se incluyen en los archivos adjuntos del CD los *drivers* para Linux de 64 bits.
3. Instalar la herramienta *arp-scan*:

```
$ sudo apt-get install arp-scan
```

4. Conectar la cámara TOF por DHCP y efectuar un análisis de la red para localizar los dispositivos conectados a la misma, empleando la herramienta *arp-scan*, tal y como se describe en 3.2.4. Probar que la cámara arranca correctamente y envía datos, ejecutando el programa “libMesaSRTester”.
5. Instalar los paquetes necesarios para la inclusión del IMU en ROS, según lo explicado en el punto 3.4.2. No será necesario calibrar el sensor ni copiar a la carpeta “~/Arduino” los ficheros tal y como se explica en ese punto, puesto que se incluyen ya modificados en los archivos adjuntos del CD; lo único que debe hacerse es copiar el archivo “my_razor.yaml” a la carpeta *config* de la ruta del paquete “razor_imu_9dof”, usando para ello el navegador de archivos con permisos superusuario (“sudo nautilus”).
6. Instalar Python Visual con el siguiente comando:

```
$ sudo apt-get install python-visual
```

   Esto se utilizará para comprobar que el IMU funciona correctamente, mediante el lanzamiento del archivo “razor-pub-and-display.launch”, dentro del paquete “razor_imu_9dof”.

7. Copiar el workspace “fusion_sensorial_ws” al directorio personal, y compilar navegando hasta el mismo desde una terminal con la herramienta

"catkin_make". Añadir la ruta de dicho workspace al .bashrc, de la siguiente manera:

```
$ gedit ~/.bashrc
$ source ~/fusion_sensorial_ws/devel/setup.bash
```

8. Instalar los *drivers* para ROS de la cámara Kinect, según lo especificado en el Anexo III.
9. Deben darse permisos a todos los *scripts* de Python presentes en el *workspace* principal. Para ello, en cada carpeta en la que haya archivos "**.py"** (consultar Anexo externo 5) se ejecutará el siguiente comando:

```
$ sudo chmod u+x *.py
```

10. Para que la interfa gráfica pueda ser correctamente visualizada, instalar el paquete de Python *termcolor*:

```
$ sudo pip install termcolor
```

11. Por último, es importante señalar que la interfaz debe ejecutarse de la siguiente manera:

```
$ roscd kinect_skeleton_ident/graphic_interface
$ ./peoplegui.py
```

No debe, bajo ningún concepto, ejecutarse, puesto que de este modo no encuentra la ruta del archivo de la interfaz gráfica (.*glade*):

```
$ rosrun kinect_skeleton_ident peoplegui.py
```

Referencias

[1] Ortega Vázquez, Enrique Manuel (2017). *Inclusión del robot PMB-2 en la plataforma basada en ROS.*

[2] Web oficial de ROS: http://www.ros.org

[3] Fernández, E.; Sánchez, L.; Mahtani, A.; Martínez, A. (2015). *Learning ROS for Robotics Programming,* capítulo 1: "Getting started with ROS".

[4] Quigley, M.; Gerkey, B.; D., William (2015). *Programming Robots with ROS,* prefacio.

[5] Proyecto STAIR, Universidad de Stanford: http://stair.stanford.edu

[6] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 11: "ROS for Industrial Robots".

[7] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Why we prefer ROS for robots".

[8] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Why some do not prefer ROS for robots".

[9] Componentes básicos de ROS: http://www.ros.org/core-components/

[10] Núcleo de ROS: http://wiki.ros.org/roscore

[11] Mensajes de ROS: http://wiki.ros.org/msg

[12] Herramienta de visualización 3D Rviz: http://wiki.ros.org/rviz

[13] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Understanding the ROS file system level".

[14] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Understanding the ROS computation graph level".

[15] Hernández, S.; Herrero, F. (2015). *Multi-Master ROS Systems.*

[16] Tópicos de ROS: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

[17] Grzegorzek, M.; Theobalt, Ch.; Koch, R.; Kolb, A. (2013). *Time-of-Flight and Depth Imaging: Sensors, Algorithms, and Applications.*

[18] Li, Larry (2014). *Time-of-Flight Camera – An Introduction.*

[19] Mesa Imaging. *SR4000/SR4500 User Manual.*

[20] Página web oficial del proyecto *open-source* PCL: http://pointclouds.org/

[21] Descarga de drivers para Mesa SR4000: http://hptg.com/industrial/

[22] Repositorio oficial del proyecto de *tracking* de personas "Open_Ptrack": https://github.com/OpenPTrack/open_ptrack

[23] Shrestha, S.; Heide, F.; Heidrich, W.; Wetzstein, G. (2016). *Computational Imaging with Multi-Camera Time-of-Flight Systems.*

[24] Refael Z.; Andrew D.; Adrian A.; Michael J. (2010). *Multiple range imaging camera operation with minimal performance impact.*

[25] Especificaciones del sensor Kinect V1: https://developer.microsoft.com/es-es/windows/kinect/hardware

[26] Página oficial de OpenNI: https://structure.io/openni

[27] TF: http://wiki.ros.org/tf

[28] Paquete *razor_imu_9dof*: http://wiki.ros.org/razor_imu_9dof

[29] Web oficial de Arduino: https://www.arduino.cc

[30] Web oficial del proyecto open-source OpenCV: http://www.opencv.org

[31] Paquete ROS "cv_bridge": http://wiki.ros.org/cv_bridge

[32] Cascadas de Haar y cómo usarlas en conjunción con OpenCV: http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html

[33] Sinha, A.; Chakravarty, K.; Bhowmick, B. (2013). *Person Identification using Skeleton Information from Kinect.*

[34] Virginia O.; Ricardo M. (2014). *Full Body Person Identification Using the Kinect Sensor.*

[35] Zeng, X.; Deng, J.; Liu, M.; Wang, Y.; Zhang, X. (2016). *A Human Identity Recognition System Based on Kinect Skeletal Tracking and ANN Classifier.*

[36] Web oficial de PyBrain: http://pybrain.org/

[37] Introducción de "Scikit" al aprendizaje artificial mediante redes neuronales: http://scikit-learn.org/stable/tutorial/basic/tutorial.html

[38] Guía de instalación de la plataforma de seguimiento de personas Open_Ptrack: https://github.com/OpenPTrack/open_ptrack/wiki/Installation-Guide

[39] Bradski, G.; Kaehler, A. (2008). *Learning OpenCV*.