UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA
UNIVERSIDAD COMPLUTENSE DE MADRID

PROYECTO FINAL DE MÁSTER

MÁSTER EN INGENIERÍA DE SISTEMAS Y CONTROL

Integración de un robot autónomo en el sistema de laboratorios remotos

Daniel García García

Directores:
Dr. Luis de la Torre Cubillo
Dr. Dictino Chaos García
Dr. Jesús Chacón Sombría

Curso académico 2017/2018 – Convocatoria de Septiembre

PROYECTO FINAL DE MÁSTER

MÁSTER EN INGENIERÍA DE SISTEMAS Y CONTROL

Integración de un robot autónomo en el sistema de laboratorios remotos

Proyecto tipo A

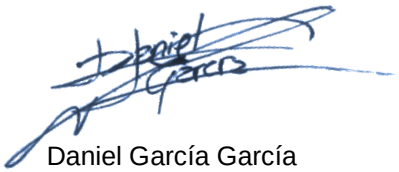
Daniel García García

Directores:
Dr. Luis de la Torre Cubillo
Dr. Dictino Chaos García
Dr. Jesús Chacón Sombría

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado:



Daniel García García

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO,

PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 5 de septiembre de 2018

Quien se suscribe:

Autor: Daniel García García

DNI: 71024375F

Hace constar que es el autor del trabajo:

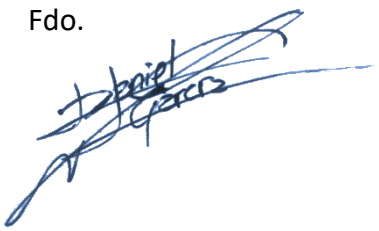
Integración de un robot autónomo en el sistema de laboratorios remotos

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente, en otra revista.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.



Resumen

El ejercicio de la docencia superior en un ámbito deslocalizado plantea retos y dificultades que los centros presenciales no tienen que afrontar, especialmente en cuanto a la transmisión de técnicas y habilidades de carácter eminentemente práctico. Esto imposibilita la implantación de titulaciones en las que la presencia del alumno es indispensable, como la medicina o la enfermería.

Ciencias como la robótica o la mecatrónica también precisan de la manipulación de elementos y dispositivos a los que un alumno que se plantee cursar a distancia una titulación de esta naturaleza difícilmente tendrá acceso en su lugar de residencia, por lo que desde la universidad se hace indispensable facilitarle el control del material necesario para la adquisición de las competencias asociadas a su campo de estudio.

El presente trabajo de fin de máster tiene como objetivo proporcionar el acceso y el control de un robot autónomo a los alumnos que así lo precisen, adaptando y tratando de mejorar el trabajo realizado por anteriores compañeros con la finalidad de implementarlo y desplegarlo en la plataforma UNILabs mediante el protocolo de comunicaciones estándar utilizado en el resto de laboratorios remotos.

Tanto el diseño hardware como el diseño software del robot autónomo utilizado en esta experiencia ya había sido previamente realizado por alumnos anteriores, pero para su correcta integración con el entorno de UNILabs ha sido necesario replantear e implementar de nuevo tanto la lógica interna de comunicación con el sistema Arduino encargado de los sensores y actuadores como las comunicaciones con el usuario final.

Por lo tanto, pese a que el objetivo principal de este trabajo es la de ofrecer al alumno la posibilidad de controlar y programar un robot autónomo, para ello ha sido necesario dotar al robot de las herramientas necesarias para poder actuar autónomamente, obtener datos del entorno y utilizarlos para la generación offline de mapas tanto bidimensionales como tridimensionales a través de diversos elementos sensoriales: encoders, cámaras de vídeo, sensores infrarrojos y un sensor Kinect™ capaz de realizar medidas de profundidad tridimensionales.

Adicionalmente, se ha procurado que el entorno remoto permita al alumno una comunicación fluida y usable, permitiendo el control manual del robot y haciendo que el código fuente pueda editarse y enviarse al robot desde la propia plataforma, guardarse en el espacio personal del alumno y obtener el resultado de las medidas obtenidas por el robot durante la experiencia.

Palabras clave: Robot autónomo, UNILabs, Kinect, localización, mapeo, laboratorio remoto, ArUco, OpenCV.

Abstract

The practice of higher education in a remote environment poses challenges and difficulties that does not appear in classroom, especially those regarding to the transmission of techniques and skills of practical nature. This makes it impossible to implement degrees in which the student's presence is essential, such as medicine or nursing.

Sciences such as robotics or mechatronics also require the manipulation of elements and devices to which a student considering taking a degree of this nature at a distance will find difficult to gain access to in his or her place of residence; because of that it is mandatory for the university to facilitate the control of the necessary devices for the acquisition of the skills associated with his or her field of study.

The aim of this master's project is to provide access and control of an autonomous robot to students who require it, adapting and trying to improve the work done by previous students in order to implement and deploy it on the UNILabs platform making use of the standard communications protocol implemented by other remote laboratories.

Both the hardware and the software design of the autonomous robot used in this experience had already been previously developed by previous students, but for its correct integration with the UNILabs environment it has been necessary to rethink and implement again both the internal communication logic with the Arduino system in charge of the sensors and actuators and the communications with the end user.

Therefore, despite the fact that the main objective of this work is to offer the student the possibility of controlling and programming an autonomous robot, it has been necessary to provide the robot with the necessary tools to be able to act autonomously, obtain data from the environment and use them for offline generation of both two-dimensional and three-dimensional maps through various sensory elements: encoders, video cameras, infrared sensors and a Kinect™ sensor capable of performing three-dimensional depth measurements.

In addition, the remote environment has been designed to allow the student to communicate fluently and usably, allowing manual control of the robot and making it

possible to edit the source code and send it to the robot from the platform itself, save it in the student's personal workspace and obtain the result of the measurements performed by the robot during the experience.

Keywords: Autonomous robot, UNILabs, Kinect, localisation, mapping, remote laboratory, ArUco, OpenCV.

Contenido

Índice de Figuras

Índice de Tablas

Índice de listados de código

1. Preámbulo

1.1. Introducción

El objetivo de este proyecto es el de integrar un robot autónomo en el entorno de laboratorios remotos UNILabs, de forma que pueda ser operado y programado remotamente por aquellos alumnos que lo precisen.

Para ello se ha partido del diseño hardware realizado por antiguos alumnos del Máster en Ingeniería de Sistemas y Control, concretamente en los trabajos de fin de máster *"Navegación de un robot autónomo utilizando balizas y luz estructurada"*[1] y *"Creación de mapas tridimensionales mediante robot teleoperado y dispositivo Kinect"*[2], de los cuales se han obtenido adicionalmente elementos conceptuales que formaban parte de los requisitos del presente trabajo.

1.2. Estado del Arte

1.2.1. Antecedentes de investigación

El término *robot* surge en la novela de ciencia-ficción *Rossum's Universal Robots* del escritor checoslovaco Karel Capek, cuyo significado es *siervo, fuerza de trabajo*. Posteriormente, es Isaac Asimov en 1950 quien populariza el término a través de su novela *"Yo, Robot"*, fundamentando las tres leyes de la robótica:

- Un robot no puede hacer daño a un ser humano por acción o por inacción.
- Un robot debe obedecer al ser humano excepto si ello hace que se contradiga la primera ley.
- Un robot debe proteger su existencia salvo que entre en conflicto con las dos leyes anteriores.

Desde los tiempos en los que la robótica era un mero concepto hasta nuestros días, en los que existen dos robots trabajando autónomamente en Marte, la evolución de las capacidades de estos dispositivos no ha hecho más que ir en aumento.

Los primeros robots con comportamientos autónomos de los que se tiene constancia son las llamadas *Tortugas de Bristol*, construidas en 1948 por W. Grey Walter, dos años antes de que Asimov enunciase sus famosas leyes. Estos dispositivos estaban compuestos por dos sensores (uno mecánico y otro lumínico) y dos motores (uno de avance y otro de giro).

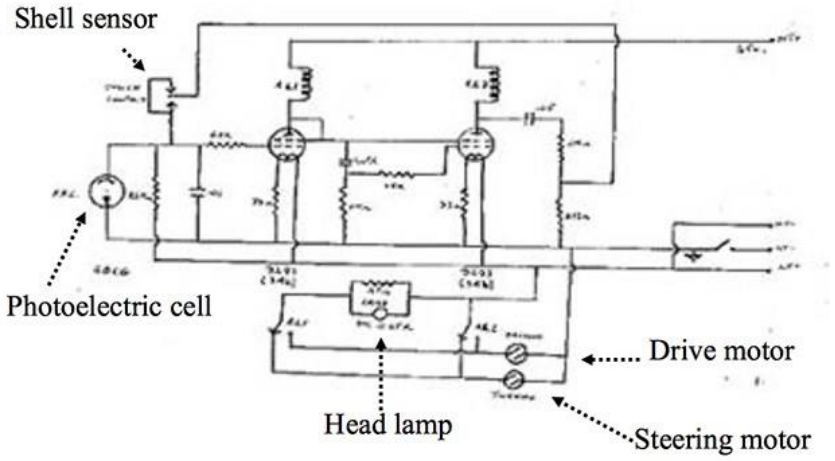


Figura 1 - Diseño de las tortugas de Bristol

Fuente: Bristol Robotics Laboratory.

El diseño estaba basado en la hipótesis que conforma a día de hoy la base de los agentes inteligentes: un sistema nervioso simple tiene la capacidad de generar comportamientos complejos e impredecibles.

Salvando su valor académico, estos dispositivos ni siquiera comulgaban con el propio concepto de "robot", puesto que no realizaban nada productivo más allá de huir de la luz fuerte y buscar fuentes de luz más tenues.

Un año más tarde, en 1951, Raymod Goertz diseña el primer brazo tele-operado, creando indirectamente el concepto de "control háptico".

En 1954, el ingeniero George Devol patenta el primer robot programable, creando la empresa *Unimation* para producir este tipo de dispositivos industriales. Por lo tanto, es en este año cuando los principales conceptos del presente proyecto (control teleoperado, funcionamiento autónomo y capacidad de programación) confluyen en una misma línea temporal.

En 1968 se construye el verdadero antecedente de los robots autónomos utilizados en ámbitos académicos, denominado *Shakey* y desarrollado por SRI, cuyo diseño, pese a estar muy alejado tecnológicamente de los robots de hoy en día, estaba íntimamente relacionado a nivel conceptual con los de los robots actuales: múltiples sensores y actuadores, capacidad de operar a través de órdenes recibidas por medios inalámbricos y pensados para ser utilizados en entornos ideales.

Los equipos de investigación tienden a día de hoy a alejarse de los diseños ideales para centrarse en resolver problemas reales, para lo cual es necesario desenvolverse en entornos comunes. No obstante, un comportamiento más complejo que actúe en tiempo real requiere de sensores más complejos, como los sensores LIDAR (*light detection and ranging*) utilizados en los actuales coches autónomos.

El sensor Kinect, sin ser una tecnología puntera a día de hoy, establece un hito en cuanto a las posibilidades que un robot autónomo puede llevar a cabo, permitiendo recabar información sensorial en tres dimensiones y extendiendo las posibilidades de estudio de robots autónomos a un nivel básico más allá de los sensores de ultrasonidos o infrarrojos utilizados habitualmente en estos laboratorios.

2. Desarrollo del proyecto

2.1. Diseño hardware

El robot autónomo utilizado en este proyecto es una evolución del diseñado y utilizado en proyectos anteriores [1][2], por lo que parte de su funcionalidad y diseño conceptual ha sido heredado de los trabajos previamente citados, siendo el objetivo del presente proyecto adaptar y facilitar al alumno su utilización a través de un laboratorio remoto implantado en la plataforma UNILabs.

No obstante, esto no implica que se haya realizado una simple adaptación de los trabajos anteriores, sino que la funcionalidad del robot se ha replanteado y recodificado desde cero haciendo uso exclusivamente del módulo de comunicación con los sensores y actuadores del robot, implementados en una placa Arduino Uno R3 que fueron implementados por Juan Ignacio Forcén Carvalho [1].

El robot a utilizar en el entorno virtual es teleoperado/automático de tipo móvil, no holonómico, compuesto por cuatro ruedas controladas por cuatro motores con un modelo cinemático de tipo diferencial (ver Figura 2).

Figura 2 - Robot teleoperado / automático

Las siguientes cotas definen las medidas del robot, cuya importancia es vital a la hora de realizar el cálculo odométrico:

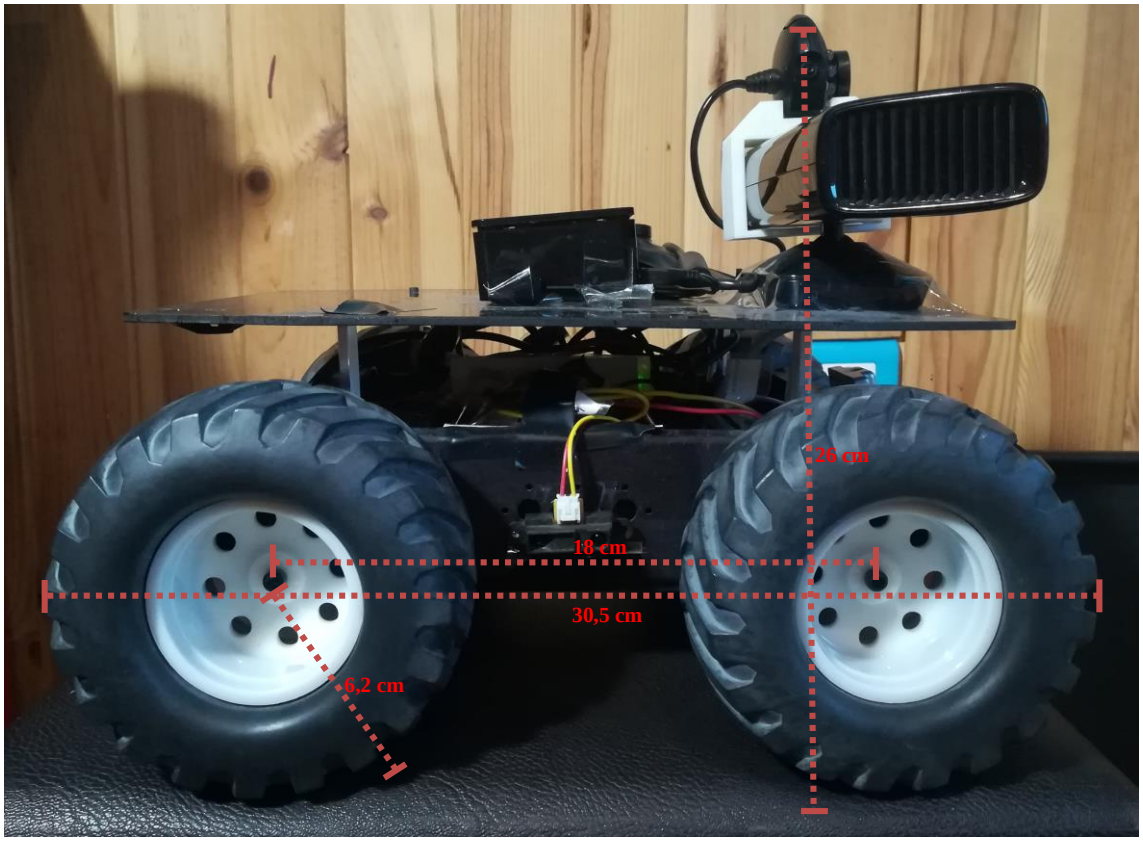


Figura 3 - Dimensiones y cotas del robot, perfil

Altura	26 cm
Longitud	30,5 cm
Distancia entre eje delantero y trasero	18 cm
Radio de la rueda	6,2 cm

Tabla 1 - Cotas del robot, perfil

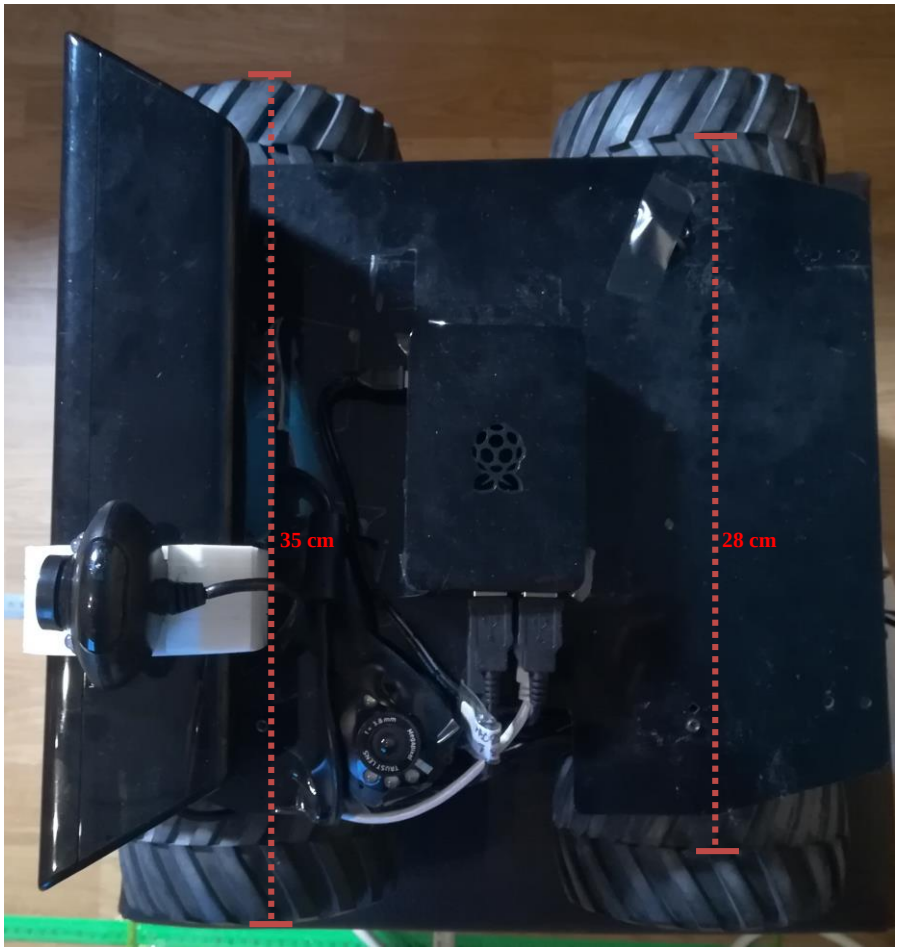


Figura 4 - Dimensiones y cotas del robot, planta

Anchura	35 cm
Distancia entre el centro de las ruedas	28 cm

Tabla 2 - Cotas del robot, planta

El robot es capaz de obtener datos de su entorno a través de los siguientes elementos controlados a través de una placa Arduino Uno R3:

- Un sensor de infrarrojos localizado en el frontal del vehículo.
- Un sensor de infrarrojos localizado en el flanco izquierdo.
- Un sensor de infrarrojos localizado en el flanco derecho.
- Dos encoders Lynxmotion Quadrature situados en las dos ruedas delanteras.

La placa Arduino Uno R3 es capaz, adicionalmente, de controlar los motores de las cuatro ruedas a través de un *Motor Shield R3* mediante modulación por ancho de pulsos, por lo que las órdenes enviadas al robot serán siempre de carácter discreto, con valores entre -255 y 255.

La carga computacional del robot se ha delegado en una placa Raspberry Pi 3 Model B, en la cual se ejecuta un sistema operativo Raspbian 4.9.80-v7, a la que se conectan los siguientes dispositivos mediante sus puertos USB:

- Placa Arduino UNO R3.
- Xbox Kinect v1441, colocado en la parte frontal del vehículo.
- Cámara Trust Spotlight 1.3MP colocada en la parte frontal del vehículo.
- Cámara Trust Spotlight 1.3MP colocada sobre el vehículo, apuntando hacia arriba.

El objetivo de cada uno de estos elementos es el siguiente:

Raspberry Pi 3 Model B: centraliza la operativa del robot, corriendo los servicios necesarios para comunicarse con todos los elementos del sistema, así como para establecer la comunicación con el usuario.

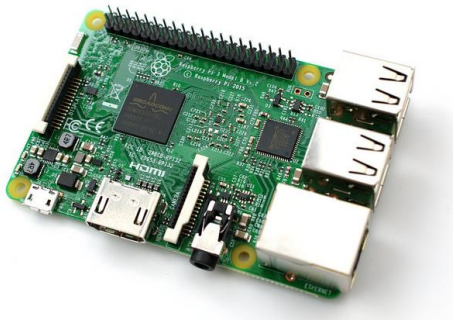


Figura 5 - Raspberry Pi Model B

Placa Arduino UNO R3 + Arduino R3 Motor Shield: obtención de datos de encoders, sensores infrarrojos y estado del motor. Envío de señales PWM a los motores para provocar el movimiento del robot.

Figura 6 - Arduino UNO R3 y Arduino Motor Shield R3

Xbox Kinect v1441: obtención de imágenes de profundidad bajo demanda a una resolución de 640x480px. Permite realizar un mapeo en profundidad del entorno del robot que permitirá realizar un postprocesamiento para su transformación a una nube de puntos, cuya suma compondrá un mapa tridimensional del entorno.

Figura 7 - Microsoft XBOX Kinect 1440

Cámara Trust Spotlight 1.3MP (frontal): realiza un streaming en tiempo real a una resolución de 320x240 de lo que se encuentra enfrente del robot, facilitando la teleoperación del mismo.

Figura 8 - Cámara Trust Spotlight 1.3MP

Cámara Trust Spotlight 1.3MP (superior): realiza capturas de imágenes bajo demanda de lo que se encuentra sobre el robot en ese momento. Permite detectar rotación y traslación de marcadores *ArUco*, cuya información puede utilizarse para complementar la información proporcionada por los encoders para posicionar al robot.

El siguiente esquema define la interconexión de los distintos elementos del robot entre sí:

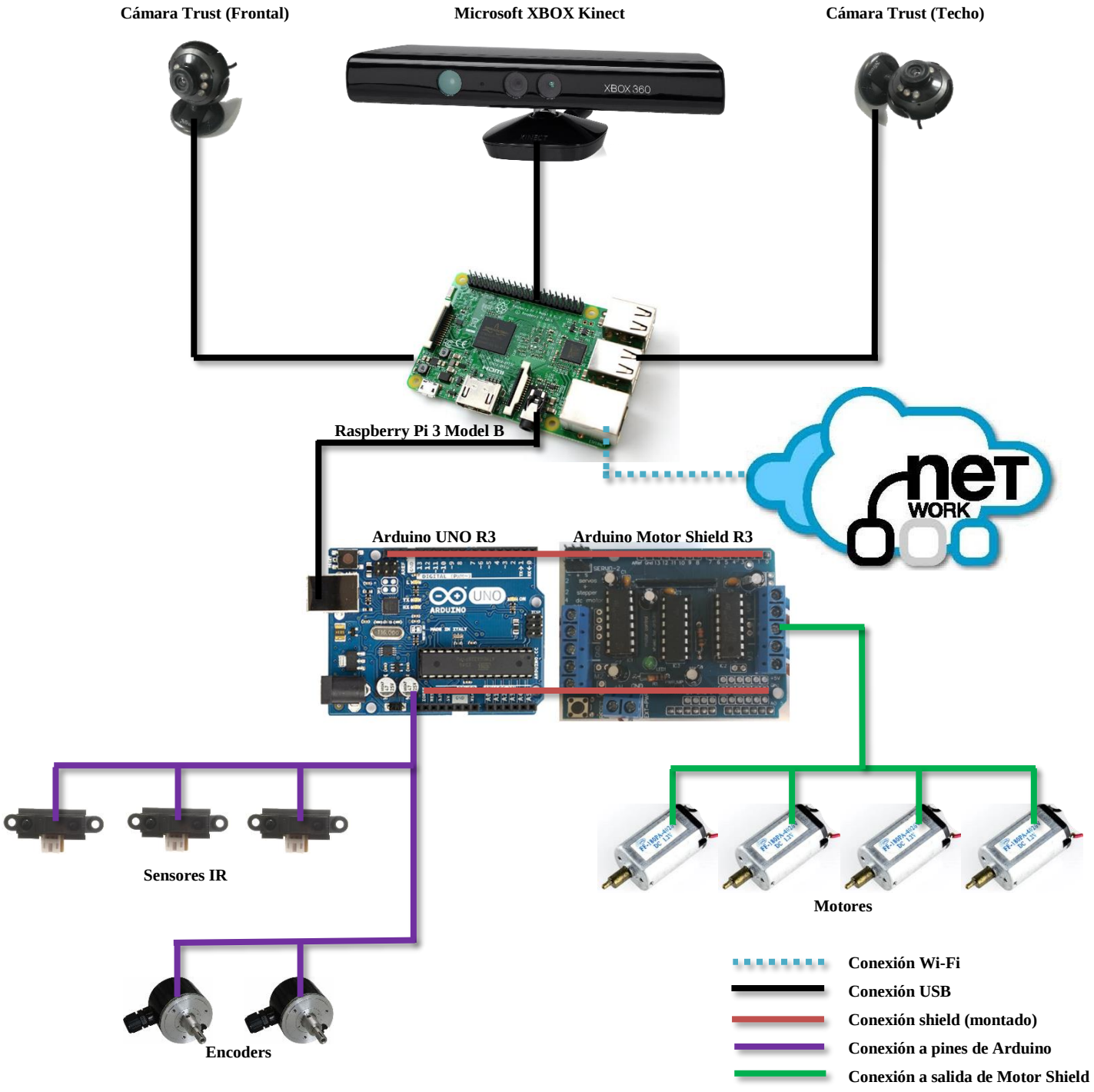


Figura 9 - Esquema de interconexión del sistema

La configuración hardware difiere de los proyectos [1] y [2] en la sustitución de la placa Raspberry Pi 2 por el modelo Raspberry Pi 3 Model B, que incorpora WiFi y Bluetooth de forma integrada, liberando de este modo uno de los puertos USB y facilitando la comunicación con la plataforma UNILabs.

Como única aportación al diseño hardware se ha diseñado e impreso en 3D un pequeño adaptador para fijar la cámara frontal al dispositivo Kinect, cuyo modelo e instalación se muestra a continuación en la Figura 10:

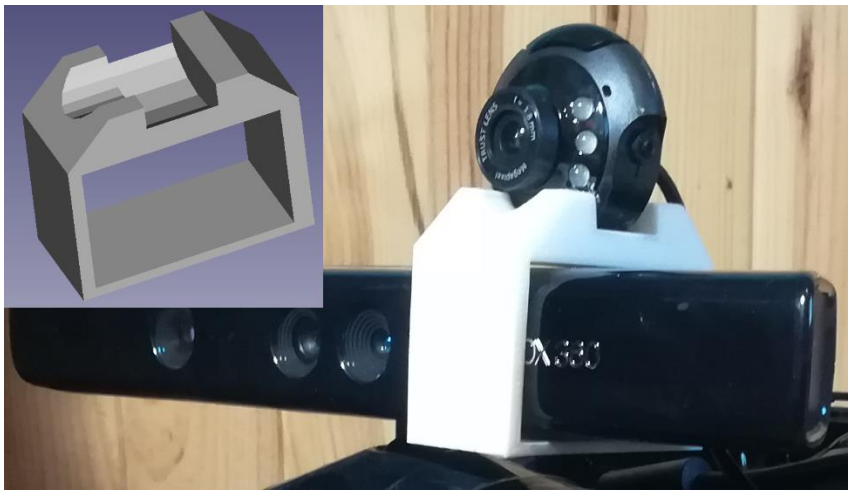


Figura 10 - Diseño 3D de la montura de la cámara frontal

El sistema necesita dos tomas de alimentación: una de 5V para la placas Raspberry Pi y Arduino y otra de 12V para alimentar el dispositivo Kinect y el Motor Shield. El desarrollo del proyecto se ha realizado mediante el uso de sendos transformadores a 5V y 12V conectados directamente a la red eléctrica, pero para un óptimo desempeño del robot, sería deseable su adaptación a un sistema autónomo de baterías.

2.2. Descripción de los elementos hardware

A continuación se describe un breve resumen de los componentes hardware utilizados en el robot, en los que se indicará su relevancia en el proyecto y cómo pueden ser utilizados para realizar mapas

tanto bidimensionales como tridimensionales. Dado que esta información ha sido ya previamente investigada y documentada en los trabajos previos en los que se basa este nuevo desarrollo, puede consultarse [2] para un detalle más exhaustivo de estos componentes.

2.2.1. XBOX Kinect

El dispositivo Kinect está compuesto por un proyector infrarrojo y dos cámaras: una infrarroja y otra estándar.

El proyector infrarrojo emite una nube de puntos invisible para el ojo humano que es detectada por el sensor CMOS de la cámara infrarroja. Esta proyección se genera de forma pseudoaleatoria y única en cada frame capturado de modo que el patrón detectado en su captura posterior sea diferente.

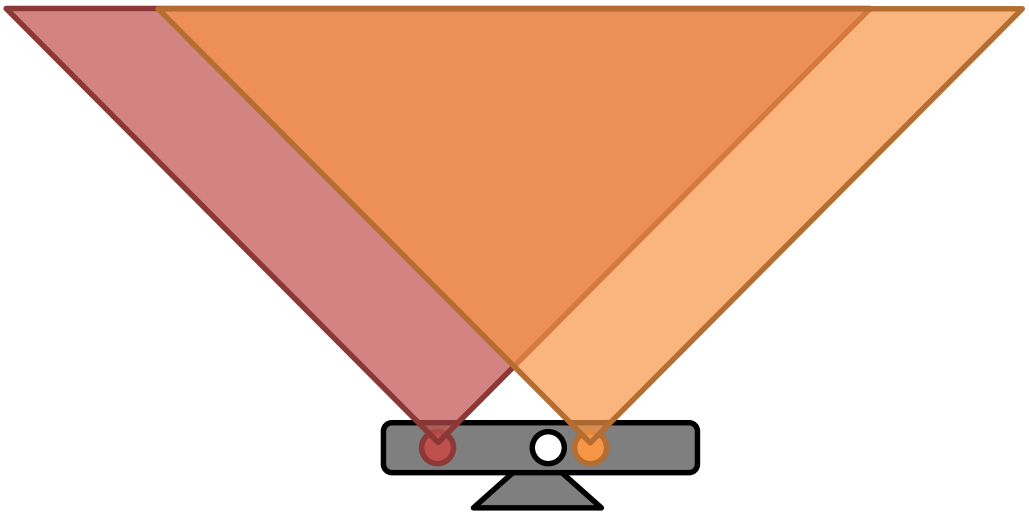


Figura 11 - Proyección y captura de malla de puntos de Kinect

El mapa de profundidad se calcula mediante la combinación de la técnica de triangulación y del algoritmo Light Coding [3], desarrollado por la empresa PrimeSense, ahora perteneciente a Apple Inc[4].

Para realizar la triangulación, Kinect compara la imagen proyectada con la capturada por el sensor infrarrojo, calculando su disparidad (diferencia de la localización de un mismo punto al ser proyectado desde distintas perspectivas), que será inversamente proporcional a la profundidad.

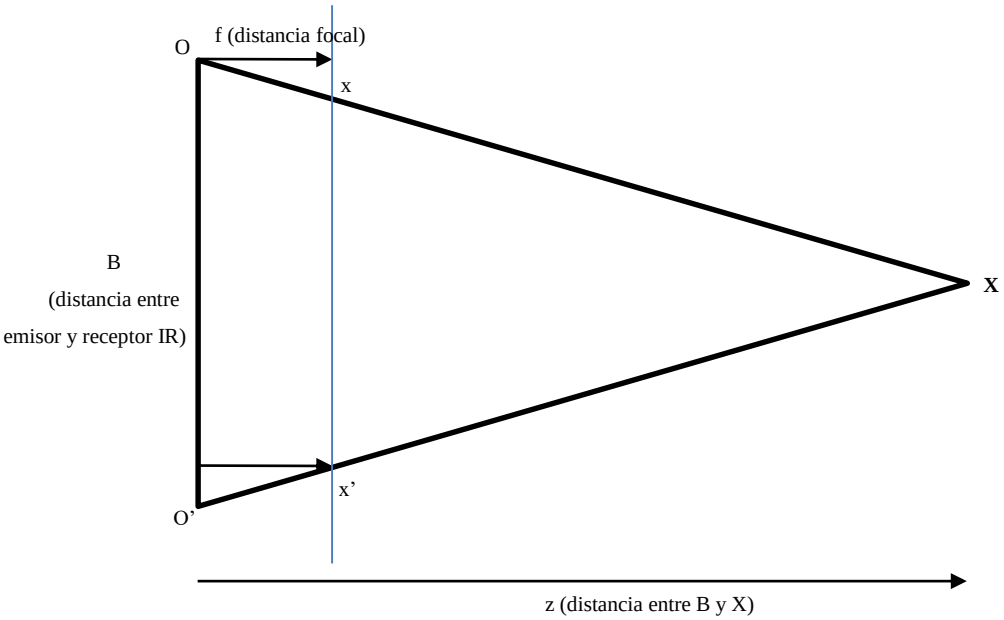


Figura 12 - Proceso de triangulación en Microsoft Kinect

De este modo, se calcula la disparidad D, y a partir de ella, la profundidad, que es inversamente proporcional a ésta:

$$D = x - x' = \frac{Bf}{z} \quad \rightarrow \quad z = \frac{Bf}{x - x'} \qquad (1)$$

A continuación, se correlaciona el patrón IR con el patrón de referencia proyectado, calculando la distancia desde cada píxel al sensor a través del cálculo de su disparidad.

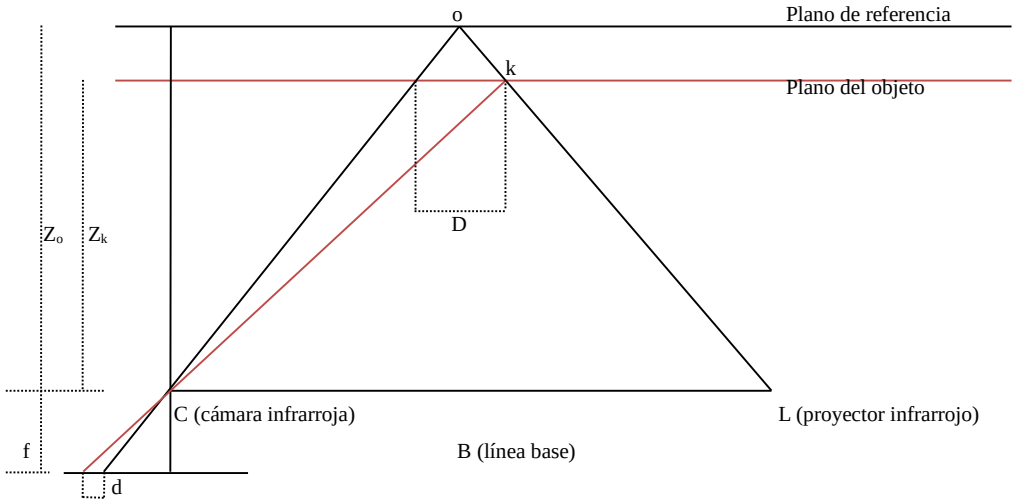


Figura 13 - Cálculo de la posición de cada píxel en Microsoft Kinect

La siguiente expresión haría uso de la ecuación (1) para el cálculo de la posición del píxel *k*:

$$Z_k = \frac{Z_0}{1 + \frac{Z_0}{Bf} d} \qquad (2)$$

Conocida la profundidad Z_k, es posible calcular las coordenadas X e Y del punto detectado, **siempre y cuando se conozcan los parámetros intrínsecos de la cámara**, correspondientes a la distancia focal f y a los parámetros de corrección de las lentes σ_x y σ_y:

$$X_k = \frac{Z_k}{f_x}(x_k - x_0 + \sigma_x) \qquad (3)$$

$$Y_k = \frac{Z_k}{f_y}(y_k - y_0 + \sigma_y) \qquad (4)$$

Siendo:

- **(X_k, Y_k):** coordenadas del punto k.
- **(x_0, y_0):** Offsets del punto.
- **(f_x, f_y):** Distancias focales de los ejes X, Y.
- **(σ_x, σ_y):** Factores de corrección de distorsión de las lentes.
- **Z_0**: Distancia entre el sensor y el plano de referencia
- **Z_k**: Distancia entre el sensor y el punto k.

El fabricante es muy críptico respecto a la especificación de Kinect, ya que pese a ser uno de los dispositivos más *hackeados* en robótica, su diseño únicamente había sido pensado para ser utilizado con la videoconsola XBOX 360. Debido a esto, salvo aplicaciones propietarias publicadas por Microsoft, nunca se han facilitado drivers ni documentación técnica que facilite el desarrollo mediante la utilización de este componente hardware.

Por ello, tanto la distancia focal como los parámetros intrínsecos de las lentes se han obtenido de forma experimental por diversos investigadores, siendo las más aceptadas las obtenidas por el equipo de desarrollo de *Robot Operating System (ROS)* [5], cuyos valores se indican a continuación [6] y que son los utilizados en el desarrollo de este trabajo para realizar los cálculos de las distancias obtenidas en las mediciones de profundidad.

- f_x = 594,21434211923247
- f_y = 591,04053696870778
- σ_x = 339,30780975300314
- σ_y = 242,73913761751615

Finalmente, para la traducción de una matriz de profundidad de 640x480 píxeles a centímetros, únicamente necesitaremos adaptar la fórmula (5) descrita en [5] a lo siguiente:

$$matrizCm = \left| \frac{100}{matriz \cdot -0.0030711016 + 3.3309495151} \right| \qquad (5)$$

2.2.2. Encoders

Los encoders proporcionan una forma directa de calcular la distancia recorrida por cada una de las ruedas, por lo que, aplicando la teoría del modelo cinemático diferencial, permitirán conocer la odometría del robot a partir de un punto original de referencia.

En el robot utilizado para este trabajo se utilizan, tal y como se explica en [2], dos encoders Lynxmotion Quadrature de 6000 pulsos por vuelta, que permite conocer tanto el número de pulsos recorridos como su orientación, tal y como se indicará más adelante en la sección *Modelo diferencial*.

Conociendo el número de pulsos y el radio de la rueda, podemos calcular la distancia recorrida a través del siguiente cálculo:

$$d = \frac{2\pi rc}{N} \qquad (6)$$

Siendo:

- **d**: distancia recorrida por la rueda.
- **r**: radio de la rueda (en este caso, 6,2 cm).
- **c**: número de cuentas detectadas por el encoder.
- **N**: número total de cuentas del encoder (en este caso, 6000).

En la descripción del modelo diferencial se indicará cómo se hace uso de estos datos y de la diferencia entre las cuentas leídas en cada encoder para realizar el cálculo de la orientación valiéndonos también de la distancia entre los puntos de apoyo de ambas ruedas.

2.2.3. Sensores infrarrojos

Otro de los sensores de los que dispone el robot es un conjunto de tres emisores/receptores infrarrojos Sharp GP2Y0A21 F55 IR situados en su parte frontal y en ambos flancos, y proporcionando un rango válido de lectura de entre 10 y 80cm.

Combinando la odometría con la lectura de estos sensores es posible generar un mapa de obstáculos bidimensional en el que reflejar la distancia a obstáculos mediante el registro de las lecturas de los sensores en distintas posiciones y orientaciones ya conocidas.

2.2.4. Cámaras Trust Spotlight 1.3MP

Con el objetivo de consumir el mínimo posible de capacidad de procesamiento obteniendo el máximo posible de información, se hace uso de dos webcams de baja resolución para obtener información visual de la parte frontal del robot (cuya información será enviada al usuario para su teleoperación) y del techo, donde se podrán colocar marcadores ArUco que podrán ser capturados y procesados para obtener información odométrica utilizándolos como balizas.

2.2.5. Arduino UNO R3 y Motor Shield R3

Estos dos elementos harware se comunican a través del puerto serie USB con la placa Raspberry Pi 3 Model B, y son capaces de proporcionar información de las lecturas de los sensores y operar sobre los motores.

Para su operación se hace uso del programa de aplicación desarrollado en [1] por Juan Ignacio Forcén Carvalho, en el que se opera en un lazo de tipo petición-respuesta mediante el envío de distintas órdenes de control mediante el puerto serie, cuyo significado se muestra en la siguiente tabla:

Mensaje	Significado
DXXX	Envía el valor XXX al motor derecho. XXX es un valor entero que debe estar comprendido entre -255 y 255.
IXXX	Envía el valor XXX al motor izquierdo. XXX es un valor entero que debe estar comprendido entre -255 y 255.
K	Mantiene la conexión evitando el *timeout*, que hace que el robot se pare automáticamente.
F	Envía el valor 155 a ambos motores, moviendo el vehículo hacia adelante.
B	Envía el valor -155 a ambos motores, moviendo el vehículo hacia atrás.
R	Envía los valores 215,-215 a los motores izquierdo y derecho, respectivamente, haciendo que el robot gire hacia la derecha.
L	Envía los valores -210,210 a los motores izquierdo y derecho, respectivamente, haciendo que el robot gire hacia la izquierda.
P	Para ambos motores

Tabla 3 - Mensajes interpretables por Arduino a través del puerto serie

Por lo tanto, el envío de los caracteres ASCII "`D100`" asignará un valor PWM de 100 a los motores derechos, mientras que la cadena "`I100D-100`" hará que los motores izquierdos giren hacia atrás con valor 100 mientras los motores derechos giran hacia adelante con valor 100.

En cuanto a la lectura de los datos de Arduino, éste escribirá en un buffer la información actual de los sensores siempre que reciba una orden, respondiendo con una cadena de texto parecida al formato JSON que proporcionará los siguientes datos:

```
{DT=XX, CD=XX, CI=XX, MP=XX, MS=XX, IrF=XX, IrR=XX, IrL=XX}
```

Siendo *XX* los valores obtenidos por los distintos elementos, que se corresponden con los siguientes:

- **DT**:diferencia de tiempo respecto a la última lectura.
- **CD**: número de cuentas leídas por el encoder derecho desde la última lectura.
- **CI**: número de cuentas leídas por el encoder izquierdo desde la última lectura.
- **MP**: estado del motor durante la lectura actual (1 = activo, 0 = inactivo)
- **MS**: estado del láser durante la lectura actual (1 = activo, 0 = inactivo)
- **IrF**: distancia en cm detectada por el sensor de infrarrojos frontal.
- **IrR**: distancia en cm detectada por el sensor de infrarrojos derecha.
- **IrL**: distancia en cm detectada por el sensor de infrarrojos izquierda.

Adicionalmente, Arduino puede responder con un número limitado de mensajes con el siguiente formato:

```
{Mensaje: "XXXX"}
```

Siendo XXXX una de las siguientes cadenas de texto:

- ***ESPERANDO ORDENES***: la comunicación ha sido establecida correctamente y Arduino está esperando para recibir órdenes a través del puerto serie.

- ***Comando desconocido***: se ha enviado una secuencia de caracteres no reconocida por Arduino.
- ***PARADO POR TIMEOUT***: no se han enviado órdenes durante un segundo, por lo que se paran automáticamente los motores.
- ***LASER APAGADO POR TIMEOUT***: no se han enviado órdenes durante un segundo, por lo que se apaga el láser.
- ***NO DA TIEMPO A EJECUTAR EL BUCLE***: no se ha podido ejecutar la orden en el período de muestreo del control interno del Arduino.

Estos mensajes y medidas serán posteriormente capturadas y tratadas en capas superiores, transformándolas a formato JSON y almacenándolas en distintas estructuras de datos para que puedan ser tratadas por el usuario.

2.2.6. Raspberry Pi 3 Model B

El principal elemento computacional del robot es una placa Raspberry Pi 3 Model B, que sustituye al modelo anterior (Raspberry Pi 2 Model B) incrementando sus prestaciones y componentes [7][8] tal y como se puede observar en la siguiente tabla:

Característica	Raspberry Pi 3 Model B	Raspberry Pi 2 Model B
CPU	Quad-Core ARM64 BCM2837	Quad-Core ARM32 Cortex-A7
Velocidad CPU	1,2 GHz	900 MHz
RAM	1 GB	1 GB
Puertos USB	4	4
Salida de video	HDMI	HDMI
Ethernet	100 Base T	100 Base T
WiFi	Integrada	No
Bluetooth	Integrado	No

Tabla 4 - Comparativa entre Raspberry Pi 2 y Raspberry Pi 3

Como podemos observar, las principales diferencias con la versión anterior radican en un aumento de 300 MHz en la señal de reloj del procesador, un cambio de arquitectura de 32 a 64 bits y la integración de Wi-Fi y Bluetooth que libera de facto dos de los puertos USB, normalmente utilizados para conectar un adaptador Wi-Fi y un *dongle* Bluetooth para la conexión conjunta de ratón y teclado inalámbrico.

El sistema operativo utilizado en la Raspberry Pi es Raspbian Stretch 4.14, una versión de Debian adaptada a Raspberry Pi instalada sobre una tarjeta microSD de 16GB.

2.3. Estructura del proyecto

El objetivo principal del proyecto es su explotación en el entorno UNILabs, por lo que se ha hecho especial hincapié en realizar una estructura robusta, documentada y mantenible, de modo que futuros alumnos o el propio equipo docente pueda modificar, corregir y ampliar el sistema de forma sencilla.

El siguiente diagrama de bloques muestra la estructura conceptual del proyecto:

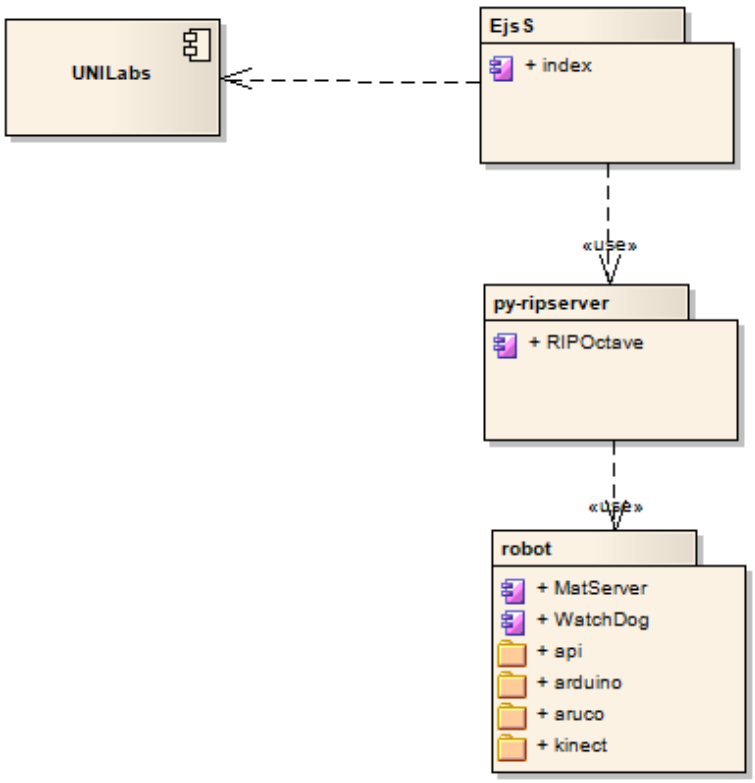


Figura 14 - Estructura global del proyecto

La experiencia de usuario se ha diseñado siguiendo el protocolo estándar de los laboratorios remotos, que se componen de dos elementos comunes:

- Una interfaz web desarrollada mediante la herramienta Easy Java Simulations (EjsS), que proporciona un paquete HTML + CSS + Javascript capaz de integrarse directamente en la plataforma UNILabs.
- Un servidor RIP encargado de realizar la comunicación entre EjsS y el robot, cuya API se proporciona por el equipo docente y que es implementada por el alumno para lograr el intercambio dinámico de datos entre la aplicación desarrolada en EjsS e integrada en UNILabs y el proyecto en desarrollo.

La interfaz en EjsS ofrece la siguiente funcionalidad al usuario:

- Conexión / desconexión al robot.
- Teleoperación manual del robot.
- Solicitud de la información sensorial del robot.
- Visualización directa en *streaming* de la cámara del robot.
- Visualización del mapa de profundidad 3D capturado por el dispositivo Kinect.
- Visualización de los fotogramas del techo capturados por la cámara montada sobre el chasis y procesada mediante realidad aumentada para proporcionar información de las balizas *ArUco*.
- Editor *online* para la edición de código *octave* interpretable por el robot.
- Visualización del log de la aplicación.
- Visualización *online* de la descripción de los métodos de la API del robot.
- Almacenamiento de código fuente en el espacio de trabajo del usuario.
- Carga de código fuente desde el espacio de trabajo del usuario.
- Descarga de los resultados de la última experiencia en formato .mat (Matlab/Octave).

El servidor RIP realiza la tarea de actuar de *middleware* entre EjsS y el robot, encargándose de las siguientes funciones:

- Inicialización de la sesión de usuario y arranque de los servicios necesarios para su correcto funcionamiento.

- Envío de órdenes al robot y solicitudes de servicios (como la generación y solicitud de descarga del fichero .mat).
- Actualización de la información actual obtenida a través de la lectura de los distintos sensores del robot.
- Transformación de la información binaria obtenida de las cámaras en formatos interpretables en HTML.

Por último, el módulo del robot es el encargado de implementar toda la lógica de servicios necesarios para ofrecer al usuario la posibilidad de operar el robot y obtener información de sus sensores.

Los principales componentes y servicios ofrecidos por el robot son los siguientes:

- Una API de programación en lenguaje Octave, que permite al usuario programar funciones específicas del robot.
- Un servicio de comunicación con el dispositivo Arduino, que permite la lectura de la información de sus sensores y operar los motores del robot.
- Un servicio de captura y tratamiento de imágenes de profundidad a través del dispositivo Kinect.
- Un servicio de captura y tratamiento de marcadores *ArUco*, que actúan a modo de balizas en el techo, proporcionando información sobre orientación y posición y que puede ser trasladada a la propia imagen mediante realidad aumentada.
- Un servicio de gestión de información de sesión, que permite el almacenamiento temporal de la información capturada por el robot durante las sesiones de los distintos usuarios.
- Un servicio *watchdog* que detecta inactividad en distintos servicios y los reinicia en caso de que no se encuentren operativos.

2.4. Diseño software

El software se ha estructurado en una colección de servicios que gestionan las posibles solicitudes realizadas desde el cliente hacia los distintos elementos que componen el robot. Pese a que hacen uso de diversas tecnologías codificadas en diversos lenguajes de programación, toda operación realizable

por el robot está centralizada en el entorno Octave, que se encarga de la comunicación con el resto de elementos del entorno.

2.4.1. Casos de uso

A la hora de realizar el diseño del robot se han identificado los siguientes casos de uso:

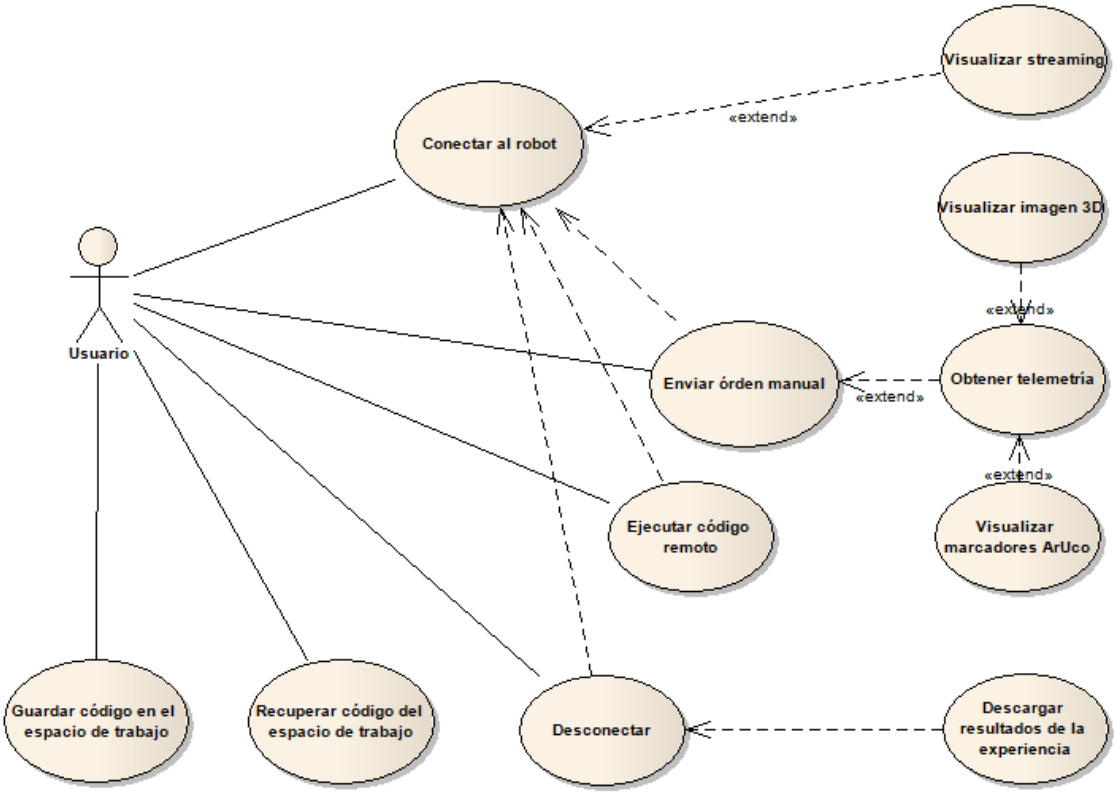


Figura 15 - Casos de uso generales

Una vez que el usuario accede a la plataforma UNILabs deberá conectarse al robot, momento en el cual comenzará a visualizarse un streaming en tiempo real de la cámara del robot. Del mismo modo, se comenzarán a solicitar a intervalos regulares de tiempo imágenes 3D capturadas a través de Kinect y composiciones de las imágenes del techo aumentadas con la información de los marcadores ArUco detectados en cada instante.

El usuario puede guardar y cargar código desde su espacio de trabajo en cualquier momento, que será mostrado en el editor *online* de la interfaz. Ese código puede ser enviado al robot para su ejecución, que será ejecutado automáticamente, y cuyos resultados podrán ser descargados una vez que se produzca la desconexión.

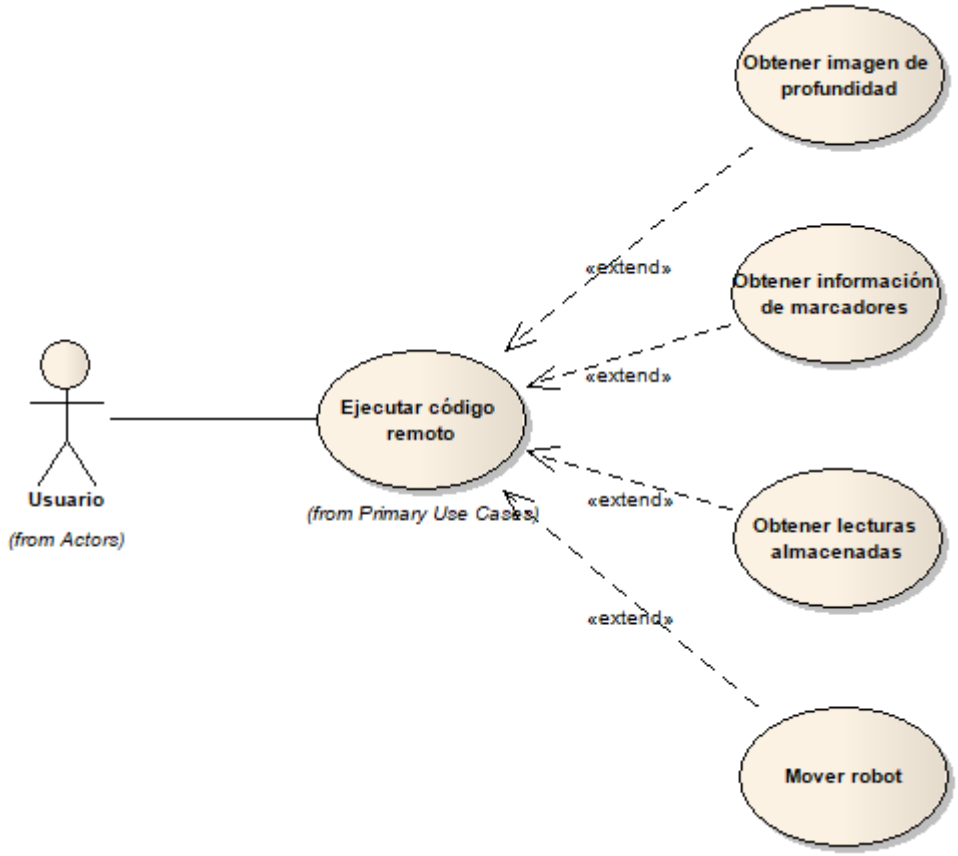


Figura 16 - Casos de uso relacionados con la ejecución de código remoto

La ejecución de código remoto permite efectuar diversas operaciones sobre el robot:

- Obtener y tratar imágenes de profundidad a través del dispositivo Kinect.
- Obtener información de los marcadores ArUco.
- Obtener información sobre las lecturas almacenadas por Arduino.
- Efectuar movimientos del robot.

La descripción de los casos de uso se puede consultar más adelante en la sección *Diseño dinámico*.

2.4.2. Diseño estático

A continuación, se detalla el diagrama estático del proyecto, así como su organización en paquetes, clases y métodos:

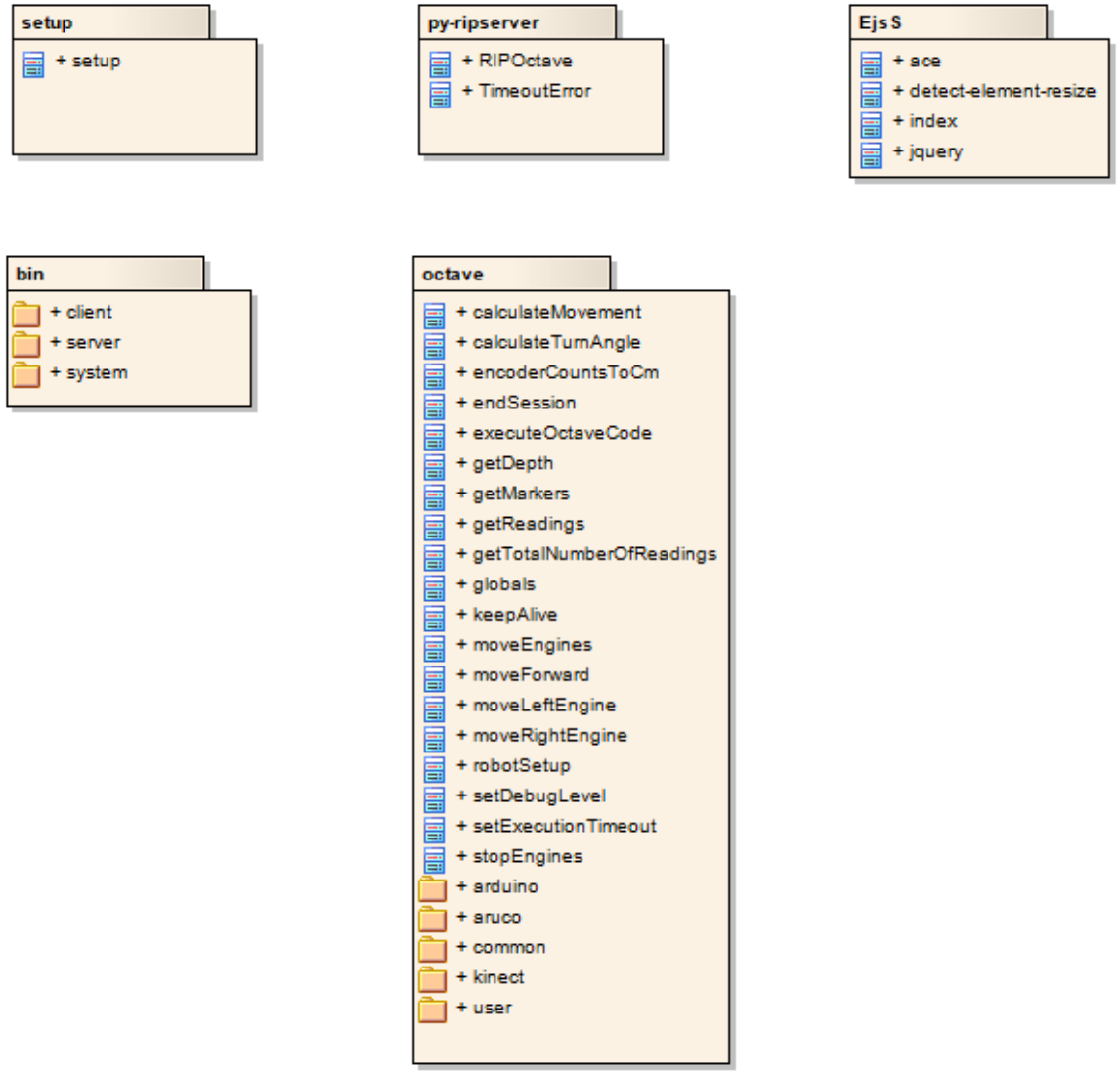


Figura 17 - Diagrama de paquetes

El contenido de cada paquete se detalla a continuación, así como una caracterización de cada una de sus clases y métodos.

2.4.2.1. *setup*

Almacena el script de instalación de las dependencias y de la propia aplicación del robot sobre una Raspbian limpia, tal y como se indica en la sección *ANEXO: Instalación y configuración*.

2.4.2.2. *EjsS*

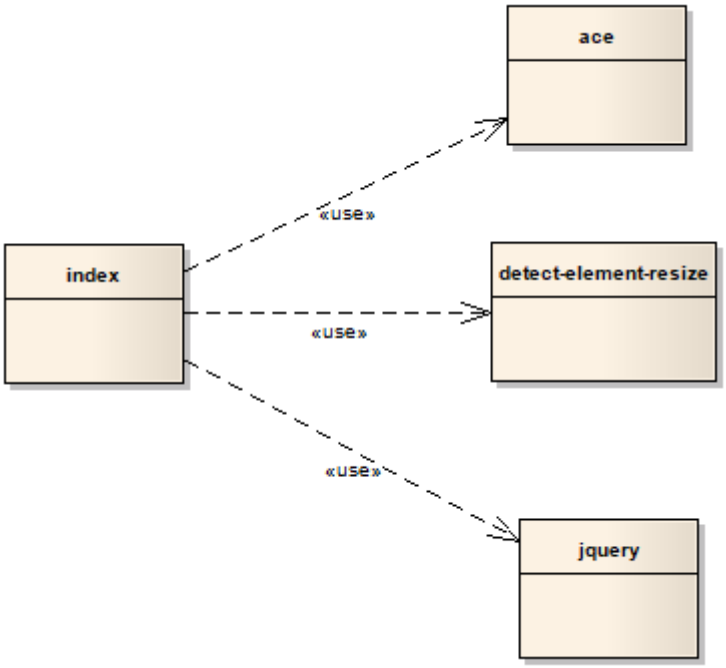


Figura 18 - Estructura del paquete EjsS

Contiene el proyecto desarrollado en Easy Java Simulations que hará las veces de interfaz entre el usuario y el RIPServer y que será embebido en la plataforma UNILabs.

Hace uso de las siguientes bibliotecas de terceros, que son embebidas o directamente referenciadas a través de su CDN:

- Ace Code editor [9]
- jQuery [10]
- Bootstrap [11]
- Font Awesome [12]
- Detect Element Resize. [13]

La siguiente tabla muestra una colección con los métodos implementados en el cliente EjsS:

Método	moveForward
Parámetros de entrada	
Retorno:	
Descripción:	
Asigna el valor 'F' a la variable 'currentAction' mediante la ejecución del método *set* expuesta por RIPOctave, haciendo que el robot se mueva hacia adelante.	

Método	releaseForward
Parámetros de entrada	
Retorno:	
Descripción:	
Restaura el estilo del botón utilizado para mover el robot hacia adelante tras finalizar la pulsación.	

Método	moveBackward
Parámetros de entrada	
Retorno:	
Descripción:	
Asigna el valor 'B' a la variable 'currentAction' mediante la ejecución del método *set* expuesta por RIPOctave, haciendo que el robot se mueva hacia atrás.	

Método	releaseBackward
Parámetros de entrada	
Retorno:	
Descripción:	
Restaura el estilo del botón utilizado para mover el robot hacia atrás tras finalizar la pulsación.	

Método	turnLeft
Parámetros de entrada	
Retorno:	
Descripción:	
Asigna el valor 'L' a la variable 'currentAction' mediante la ejecución del método *set* expuesta por RIPOctave, haciendo que el robot gire hacia la izquierda.	

Método	releaseLeft
Parámetros de entrada	
Retorno:	
Descripción:	
Restaura el estilo del botón utilizado para mover el robot hacia la izquierda tras finalizar la pulsación.	

Método	turnRight
Parámetros de entrada	
Retorno:	
Descripción:	
Asigna el valor ‘R’ a la variable ‘currentAction’ mediante la ejecución del método *set* expuesta por RIPOctave, haciendo que el robot gire hacia la derecha.	

Método	releaseRight
Parámetros de entrada	
Retorno:	
Descripción:	
Restaura el estilo del botón utilizado para mover el robot hacia la izquierda tras finalizar la pulsación.	

Método	stopEngines
Parámetros de entrada	
Retorno:	
Descripción:	
Asigna el valor ‘S’ a la variable ‘currentAction’ mediante la ejecución del método *set* expuesta por RIPOctave, haciendo que el robot detenga los motores.	

Método	releaseStop
Parámetros de entrada	
Retorno:	
Descripción:	
Restaura el estilo del botón utilizado para detener el robot tras finalizar la pulsación.	

Método	keepAlive
Parámetros de entrada	
Retorno:	
Descripción:	
Asigna el valor ‘K’ a la variable ‘currentAction’ mediante la ejecución del método *set* expuesta por RIPOctave, haciendo que el robot mantenga activos los motores.	

Método	releaseKeepAlive
Parámetros de entrada	
Retorno:	
Descripción:	
Restaura el estilo del botón utilizado para mantener activos los motores del robot tras finalizar la pulsación.	

Método	sendOctaveCode

Parámetros de entrada	
Retorno:	
Descripción:	
Asigna el contenido del editor de texto a la variable 'octaveCode' mediante la ejecución del método *set* expuesta por RIPOctave, haciendo que sea enviado al robot, escrito en el fichero octave/user/usercode.m y a continuación, ejecutado.	

Método	customizeStyles
Parámetros de entrada	
Retorno:	
Descripción:	
Asigna estilos css de forma dinámica a los elementos de la interfaz.	

Método	sleep
Parámetros de entrada	ms:int
Retorno:	
Descripción:	
Realiza una pausa de *ms* milisegundos, siempre y cuando la función que invoca el método sea declarada como *async*.	

Método	addCustomEvents
Parámetros de entrada	
Retorno:	
Descripción:	
Añade handlers de eventos a los elementos de la interfaz, permitiendo realizar las siguientes operaciones: - Redimensionar las dos mitades de la interfaz cuando cambia el tamaño de la ventana. - Añadir una referencia a la biblioteca jQuery - Añadir una referencia a la biblioteca Ace Editor	

Método	createDynamicElements
Parámetros de entrada	
Retorno:	
Descripción:	
Genera elementos HTML no incluidos inicialmente en el diseño estático y los inyecta dentro de la interfaz: - *Slider* de conexión/desconexión al servidor. - Textarea con el log del sistema. - Popup modal con la ayuda de la API.	

Método	showHelp
Parámetros de entrada	
Retorno:	
Descripción:	
Muestra el pop-up con la ayuda de la API del robot.	

Método	fixSarlabStyles
Parámetros de entrada	
Retorno:	
Descripción:	
Mitiga la sobrecarga de estilos entre los css de la aplicación EjsS y el entorno UNILabs en el que se embebe.	

Método	getUserId
Parámetros de entrada	
Retorno:	String
Descripción:	
Obtiene el ID del usuario actual y lo devuelve como valor de retorno.	

Método	configureEditor
Parámetros de entrada	
Retorno:	
Descripción:	
Configura las propiedades y estilos del editor ace destinado a albergar el código fuente del robot.	

Método	uploadCodeToWorkspace
Parámetros de entrada	
Retorno:	
Descripción:	
Almacena el código contenido en el editor de texto en el espacio de trabajo del alumno.	

Método	downloadCodeFromWorkspace
Parámetros de entrada	
Retorno:	
Descripción:	
Carga en el editor de texto ace el fichero seleccionado en el espacio de trabajo del alumno.	

Método	updateLog
Parámetros de entrada	text: string
Retorno:	
Descripción:	
Actualiza la caja de texto que almacena el log de la sesión.	

Método	setHelpText
Parámetros de entrada	
Retorno:	
Descripción:	
Configura el contenido de la ayuda de la API del robot.	

Tabla 5 – Descripción de los métodos de EjsS

2.4.2.3. py-ripserver

Contiene la implementación específica de RIPOctave con la capacidad de comunicarse tanto con la interfaz desarrollada en EjsS como con las distintas funciones del robot, así como el entorno virtual configurado para la plataforma ARM64.

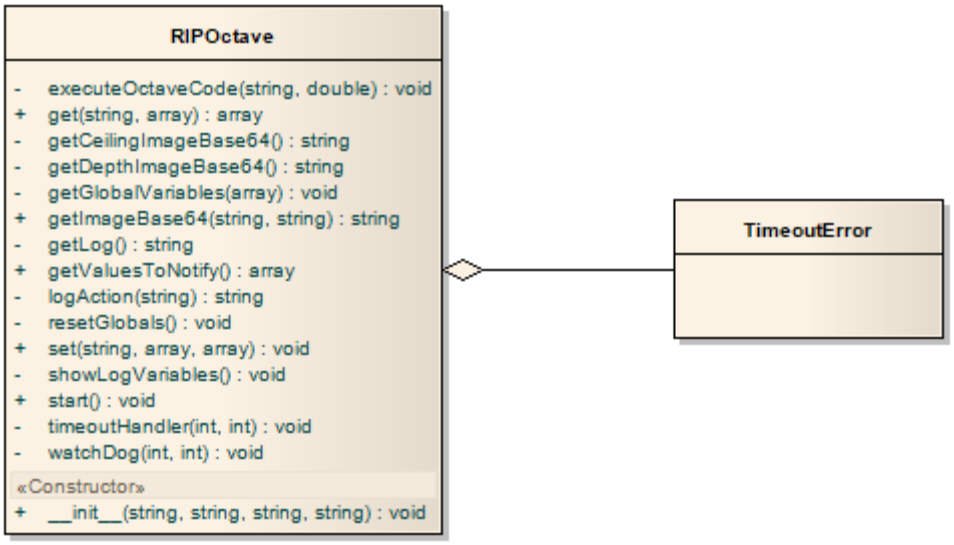


Figura 19 - Contenido del paquete py-ripserver

La siguiente tabla muestra la descripción de los métodos de la clase RIPOctave, encargada de gestionar la comunicación entre EjsS y el robot:

Método	executeOctaveCode
Parámetros de entrada	code: string

	timeout: int
Retorno:	-
Descripción:	
Escribe el código *code* en el fichero *usercode.m* en la carpeta /octave/user del robot y hace que Octave lo ejecute. Si transcurridos *timeout* segundos la ejecución del código no ha finalizado, se aborta automáticamente la ejecución.	

Método	get
Parámetros de entrada	expid: string
Parámetros de salida:	variables: array<string>
Retorno:	dictionary<string, object>
Descripción:	
Recupera de la sesión octave las variables pasadas como parámetro en el array *variables* y devuelve sus valores en una colección de pares clave/valor	

Método	getCeilingImageBase64
Parámetros de entrada	
Retorno:	ceilingImageBase64: string
Descripción:	
Invoca el método getMarkerInfo() de Octave, realizando una captura de la webcam del techo y aumentándola con la información de los marcadores detectados. A continuación, se codifica en formato base64 mediante el método getImageBase64() para facilitar su envío a EjsS través del método getValuesToNotify().	

Método	getDepthImageBase64
Parámetros de entrada	
Retorno:	
Descripción:	
Invoca el método getDepthImage() de Octave, generando una imagen a partir de la matriz de profundidad obtenida por Kinect. A continuación, se codifica en formato base64 mediante el método getImageBase64() para facilitar su envío a EjsS través del método getValuesToNotify().	

Método	getGlobalVariables
Parámetros de entrada	result: array<double>
Retorno:	
Descripción:	
Almacena en las variables TS, DT, CD, CI, MP, MS, IrF, IrR e IrL los valores del array *result*, que debe haber sido recuperado desde Octave mediante la función *updateGlobals*.	

Método	getImageBase64
Parámetros de entrada	imagePath: string

	lockFilePath: string
Retorno:	imageBase64: string
Descripción:	
Comprueba que el archivo *lockFilePath* no exista y a continuación lee la imagen almacenada en la ruta *imagePath* y la convierte a formato string base64. El fichero *lockFilePath* es un archivo de bloqueo generado por los servicios encargados de capturar imágenes al comienzo del proceso, siendo eliminado cuando la imagen está lista para su utilización.	

Método	getLog
Parámetros de entrada	
Retorno:	logData: string
Descripción:	
Extrae la información almacenada por el log del sistema, extrae sus líneas y las filtra para su posterior envío al cliente.	

Método	getValuesToNotify
Parámetros de entrada	
Retorno:	returnValue: array[[array<string>,array<object>]]
Descripción:	
Este método es invocado desde EjsS periódicamente para obtener información en tiempo real del estado del robot. La función recupera los últimos valores de los sensores desde octave invocando el método *updateGlobals* y recupera en formato base64 una imagen de profundidad del dispositivo Kinect y una imagen aumentada del techo mediante la webcam superior en una de cada tres invocaciones al método para no penalizar computacionalmente la experiencia. A continuación, obtiene el log del sistema desde la última llamada. Finalmente, almacena los resultados en dos arrays que simulan un conjunto de pares clave valor y que contiene los siguientes elementos: time, TS, DT, CD, CI, MP, MS, IrF, TSM, Mensaje, KinectImageBase64, CeilingImageBase64, Ready, octaveLog.	

Método	logAction
Parámetros de entrada	action: string
Retorno:	actionName: string
Descripción:	
Transforma un mnemónico interpetable por arduino (‘K’, ‘P’, ‘D100’, …) en una cadena de texto interpretable por un ser humano para su inclusión en el log del sistema.	

Método	resetGlobals
Parámetros de entrada	

Retorno:	
Descripción:	
Establece a cero el valor de los atributos TS, DT, CD, CI, MP, MS, IrF, TSM.	

Método	set
Parámetros de entrada	expid: string variables: array<string> values: array<object>
Retorno:	
Descripción:	
Asigna un valor a una variable. Este método es invocable desde la interfaz EjsS y sirve para transmitir información al robot desde el lado de cliente. Las tres posibles variables que se manejan en este método son las siguientes: - **currentAction**: se corresponde con una acción del robot ('K', 'F', 'B', 'R', 'L') y son enviadas a Arduino mediante la invocación del método *arduinoProcessInputs()*. - **octaveCode**: se corresponde con el código que debe ser enviado y ejecutado en el robot, por lo que se invoca el método *executeOctaveCode()*. - **userId**: se corresponde con el identificador del usuario, que será utilizado para hacer única la sesión y establecer el fichero de caché donde se almacenarán los valores capturados por el usuario.	

Método	showLogVariables
Parámetros de entrada	
Retorno:	
Descripción:	
Almacena en el log del sistema la información de los atributos TS, DT, CD, CI, MP, MS, IrF, TSM en un formato legible para el ser humano.	

Método	start
Parámetros de entrada	
Retorno:	
Descripción:	
Método que será invocado al iniciar la sesión de usuario, y que lleva a cabo las siguientes acciones: - Genera un fichero de caché identificado por el ID de usuario obtenido desde EjsS y por la marca de tiempo de inicio de sesión. - Invoca el método de Octave *robotSetup*, que inicia los servicios de streaming, Arduino, ArUco y Kinect. - Realiza una primera petición de datos al robot para obtener su estado inicial.	

Método	timeoutHandler
Parámetros de entrada	signum: int frame: int

Retorno:	
Descripción:	
Lanza una excepción en caso de que se detecte un timeout en la ejecución del código octave.	
Método	watchDog
Parámetros de entrada	signum: int frame: int
Retorno:	
Descripción:	
Handler que responde a la señal SIGUSR2 enviada por robotwatchdog leyendo un número entero contenido dentro de un fichero, incrementando su valor y volviéndolo a escribir, indicando así al watchdog que el proceso sigue vivo. En caso de que el watchdog detecte que el dato escrito en el fichero no ha sido incrementado, asumirá que el proceso ha dejado de responder y forzará su reinicio automáticamente.	
Método	__init__
Parámetros de entrada	
Retorno:	
Descripción:	
Constructor de la clase. Declara las variables globales de Octave, carga los *paths* de las funciones a utilizar por el robot e indica a EjsS las variables disponibles para su lectura y escritura.	

Tabla 6 - Descripción de los métodos de la clase RIPOctave

2.4.2.4. bin

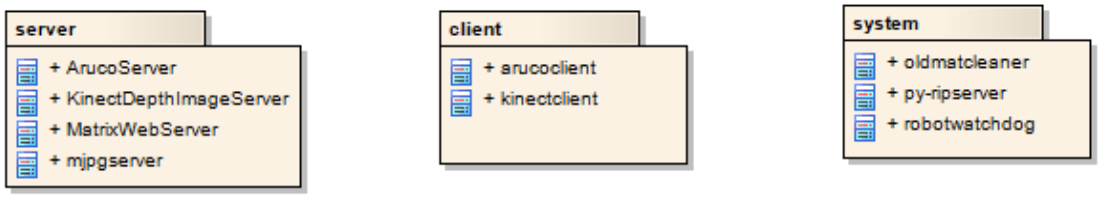


Figura 20 – Contenido del paquete "bin"

Implementa los distintos servicios y clientes externos que son invocados desde Octave para proporcionar funciones que éste no puede prestar por sí mismo, como la captura de imágenes de profundidad o el streaming de video.

ArucoServer

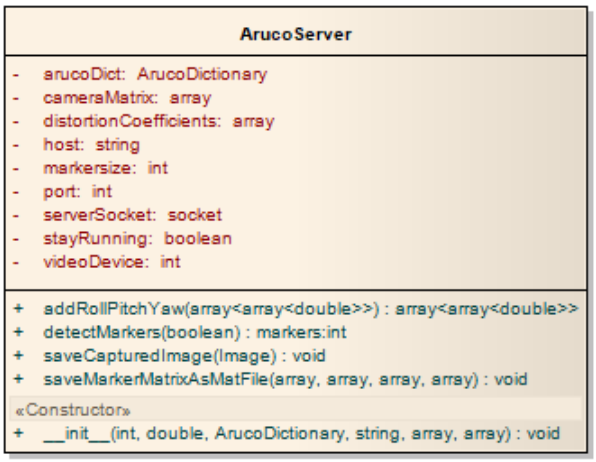


Figura 21 - Clase ArucoServer

Clase encargada de capturar las imágenes a través de la cámara que apunta al techo, así como de obtener la información sobre los marcadores detectarlos y aumentar las imágenes capturadas añadiéndole dicha información.

Para ello se lanza como un servicio escuchando en el puerto 12501, al cual deberán conectarse los clientes para solicitarle alguna de las siguientes operaciones:

- *quit*: finaliza la ejecución del servicio.
- *status:* devuelve el valor "0" si el servicio está activo
- *detect:* obtiene la información sobre las esquinas, vectores de rotación y vectores de traslación de los marcadores detectados.
- *capture:* obtiene la información sobre las esquinas, vectores de rotación y vectores de traslación de los marcadores detectados y almacena una imagen aumentada con esa información en una ruta predefinida.

Método	addRollPitchYaw
Parámetros de entrada	array<array<array<double>>>
Retorno:	array<array<array<double>>>
Descripción:	
Función interna que añade la información de alabeo, cabeceo y guiñada a la matriz de rotación.	

Devuelve una matriz con seis elementos: los tres primeros se corresponderán con los componentes de la matriz de rotación en notación eje-ángulo, mientras que los tres últimos se corresponderán con los ángulos de alabeo, cabeceo y guiñada, respectivamente.

Método	detectMarkers
Parámetros de entrada	capture: boolean
Retorno:	markers:int

Descripción:

Captura un frame de la cámara que apunta al techo y detecta si existen marcadores ArUco en la imagen.

En caso afirmativo, extrae la información de la localización de sus esquinas, vectores de rotación y vectores de traslación y los almacena en un fichero .mat en una localización predeterminada para su posterior procesamiento por Octave.

Si el parámetro *capture* es *True*, adicionalmente guardará el frame capturado, aumentándolo con la información de posición y orientación de los marcadores detectados, si procede.

Método	saveCapturedImage
Parámetros de entrada	image: Image
Retorno:	

Descripción:

Redimensiona la imagen a una resolución de 320x240 y la almacena en una ruta predefinida.
Este método se utiliza en RIPOctave para permitir al cliente visualizar las capturas del techo en la interfaz gráfica tras transformar la imagen en una cadena base64.

Método	saveMarkerMatrixAsMatFile
Parámetros de entrada	ids: array<double> corners: array<double> rvecs: array<double> tvecs: array<double>
Retorno:	

Descripción:

Almacena en una ruta predefinida un fichero *.mat* procesable por Octave con la información de los *markers* capturados por la cámara cuya información se pasa como parámetro. Cada fila de los arrays pasados como parámetro se corresponderá con la información de un marcador diferente identificado en la imagen, cuyas estructuras se corresponden con los siguientes datos:

- ids: array con los identificadores de los marcadores
- corners: array con las coordenadas de las esquinas de los marcadores
- rvecs: array con los vectores de rotación de los marcadores.
- tvecs: array con los vectores de traslación de los marcadores.

La información, recuperada en bruto, se tratará con Octave en el método *getMarkerInfo()*.

Método	`__init__`
Parámetros de entrada	videoDevice: int markersize: double arucoDict: ArucoDictionary matImagePath: string cameraMatrix: array<array<double>> distortionCoefficients: array<double>
Retorno:	
Descripción:	

```
Constructor de la clase ArucoServer.
Configura los parámetros necesarios para la correcta ejecución del servidor:
    - Número de dispositivo de vídeo a utilizar.
    - Tamaño de los marcadores ArUco (cm)
    - Diccionario ArUco a utilizar.
    - Ruta donde se almacenará la matriz con la información de los marcadores
    - Matriz de calibración de la cámara
    - Coeficientes de distorsión de la cámara.
Una vez que el servidor ha sido configurado, lanza un servicio que se mantiene a la
espera de peticiones a través del puerto 12501.
```

Tabla 7 – Descripción de los métodos de la clase ArucoServer

KinectDepthImageServer

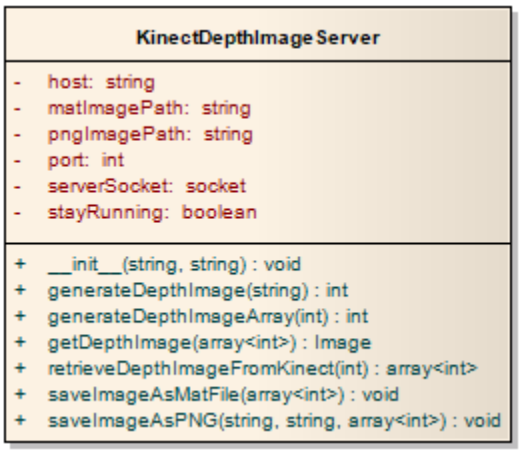


Figura 22 - Clase KinectDepthImageServer

Clase encargada de comunicarse con el dispositivo Kinect y obtener las imágenes de profundidad, tanto en formato de imagen como de matriz de datos.

Para ello se lanza como un servicio escuchando en el puerto 12500, al cual deberán conectarse los clientes para solicitarle alguna de las siguientes operaciones:

- *quit*: finaliza la ejecución del servicio.
- *status:* devuelve el valor "0" si el servicio está activo
- *depth:* realiza una captura de la profundidad detectada por Kinect en formato matricial, con una resolución de 640x480 píxeles en formado de entero de 16 bits, obteniendo valores entre 0 y 2047, guardando el resultado en una ruta predeterminada.
- *image:* realiza una captura de la profundidad detectada por Kinect en formato de imagen PNG, con una resolución de 320x240, guardando el resultado en una ruta predeterminada.
- *all:* realiza las operaciones *depth* e *image*.

Método	generateDepthImage
Parámetros de entrada	imageFilter: string
Retorno:	int
Descripción:	
Obtiene una lectura del sensor Kinect y genera una imagen en formato PNG en la ruta indicada por el atributo *pngImagePath*, aplicándole el mapa de color pasado como parámetro, que será *jet* por defecto. Devuelve 0 si la operación es correcta y un valor mayor que 0 en caso de error.	

Método	generateDepthImageArray
Parámetros de entrada	depth: int
Retorno:	
Descripción:	
Obtiene una lectura del sensor Kinect y genera un array de enteros de 16 bits en la ruta indicada por el atributo *matImagePath*. Opcionalmente puede reducirse el tamaño de los enteros de 16 a 8 bits indicando este último valor como parámetro de entrada, pero implicaría una pérdida de información, por lo que sólo se utiliza de forma interna para generar la imagen PNG. Devuelve 0 si la operación es correcta y un valor mayor que 0 en caso de error.	

Método	getDepthImage
Parámetros de entrada	
Retorno:	image
Descripción:	
Obtiene una lectura del sensor Kinect y genera una imagen en formato PNG con una resolución de 640x480 píxeles, que será devuelta como valor de retorno.	

Método	retrieveDepthImageFromKinect
Parámetros de entrada	depth: int
Retorno:	array<int>
Descripción:	
Solicita una matriz de profundidad al dispositivo Kinect, que será devuelta como un array de enteros de 16 u 8 bits, dependiendo del valor pasado como parámetro.	
Método	saveImageAsMatFile
Parámetros de entrada	depthArray: array<int>
Retorno:	
Descripción:	
Almacena el array pasado como parámetro en un fichero .mat con el nombre 'depthImage' en la ruta especificada en el atributo matImagePath.	
Método	saveImageAsPNG
Parámetros de entrada	path: string imageFilter: string arrayImage: array<int>
Retorno:	
Descripción:	
Almacena el array pasado como parámetro en un fichero .png con una resolución de 320x240px en la ruta especificada en el artibuto pngImagePath con el colormap indicado en el parámetro imageFilter, que por defecto será *jet*.	
Método	__init__
Parámetros de entrada	matImagePath: string pngImagePath: string
Retorno:	
Descripción:	
Constructor de la clase KinectDepthImageServer. Configura los parámetros necesarios para la correcta ejecución del servidor: - Ruta en la que se almacenará la matriz generada por el dispositivo Kinect. - Ruta en la que se almaenará la imagen generada por el dispositivo Kinect. Una vez que el servidor ha sido configurado, lanza un servicio que se mantiene a la espera de peticiones a través del puerto 12500.	

Tabla 8 - Descripción de los métodos de la clase KinectDepthImageServer

mjpgserver

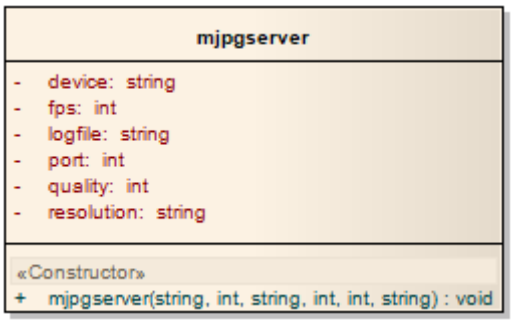


Figura 23 - Clase mjpgserver

Clase encargada de lanzar el servicio de streaming de la cámara frontal.

Método	mjpgserver
Parámetros de entrada	device: string fps: int resolution: string quality: int port: int logfile: string
Retorno:	
Descripción:	
Habilita el servicio de streaming del dipositivo *device* (por defecto, /dev/video0) en el puerto *port* (por defecto, 8081). El video tendrá una tasa de refresco de *fps* frames por segundo (por defecto, 5), una calidad de un *quality%* (por defecto, 30) y una resolución de *resolution* píxeles (por defecto, 320x240), registrando los posibles problemas detectados durante la ejecución en la ruta *logfile*.	

Tabla 9 - Descripción de la clase mjpgserver

MatrixWebServer

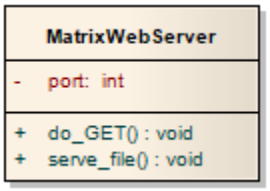


Figura 24 - Clase MatrixWebServer

Genera un fichero .mat a partir de la caché de datos almacenados por el usuario y lo envía al usuario a través del puerto 8080.

Método	do_GET
Parámetros de entrada	
Retorno:	
Descripción:	
Escucha en el puerto 8080 una ruta con el formato http://127.0.0.1/XXXX, siendo XXXX un identificador numérico correspondiente al usuario de la plataforma. Una vez conectado al puerto 8080, se invocan los métodos Octave *getLastCachedDataFilePath* pasándole el ID de usuario como parámetro y *saveEnvironment* para generar el fichero .mat que será enviado al usuario como resultado de la experiencia.	
Método	serve_file
Parámetros de entrada	
Retorno:	
Descripción:	
Realiza un HTTP RESPONSE al usuario como respuesta de la petición GET, proporcionándole un archivo de nombre *robot.mat* con los resultados de la experiencia.	

Tabla 10 – Descripción de la clase MatrixWebServer

arucoclient

Permite controlar el servidor ArucoServer mediante el envío de una de las siguientes órdenes de control:

- *start*: inicia el servicio.
- *restart*: reinicia el servicio.
- *stop*: finaliza la ejecución del servicio.
- *status:* devuelve el valor "0" si el servicio está activo, "1" en caso contrario.
- *detect:* obtiene la información sobre las esquinas, vectores de rotación y vectores de traslación de los marcadores detectados.
- *capture:* obtiene la información sobre las esquinas, vectores de rotación y vectores de traslación de los marcadores detectados y almacena una imagen aumentada con esa información en una ruta predefinida.

kinectclient

Permite controlar el servidor KinectDepthImageServer mediante el envío de una de las siguientes órdenes de control:

- *start*: inicia el servicio.
- *restart*: reinicia el servicio.
- *stop*: finaliza la ejecución del servicio.
- *status:* devuelve el valor “0” si el servicio está activo, “1” en caso contrario.
- *depth :* realiza una captura de profundidad en un archivo .mat.
- *image:* realiza una captura de profundidad en una imagen png.
- *all:* realiza una captura de profundidad en .mat y como imagen .png

2.4.2.5. *octave*

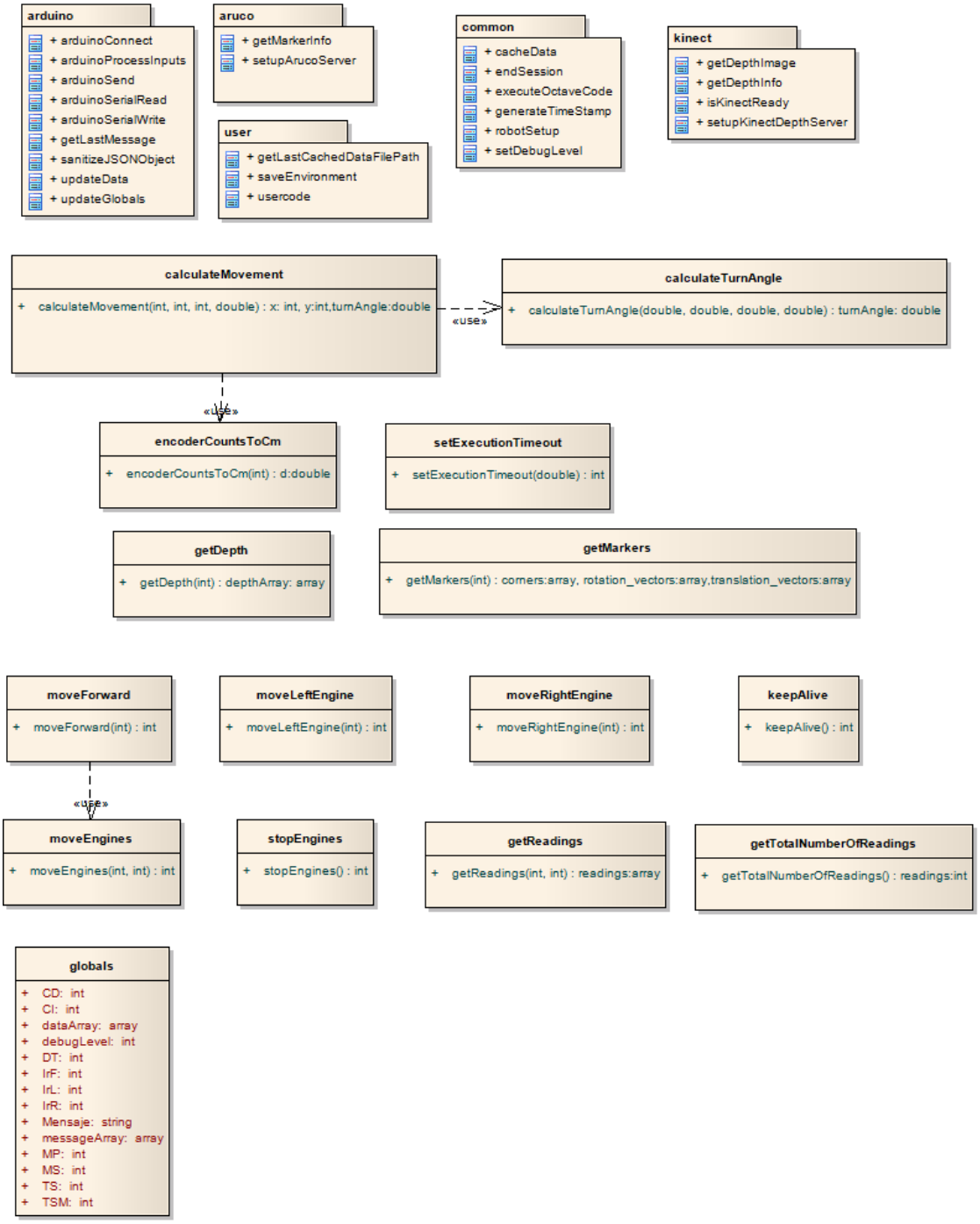


Figura 25 - Diagrama con los métodos Octave

Implementa la API de usuario y los métodos de gestión de la aplicación capaces de preparar el entorno, comunicarse con los distintos elementos del sistema e intercambiar información con el RIPServer.

Las funciones de los paquetes *arduino, aruco, common, kinect y user* tienen carácter interno y no se ofrecen directamente al usuario, mientras que los métodos de primer nivel pueden ser utilizados directamente y se ofrecen como API al estudiante.

A continuación, se describirán las funciones de dichos paquetes, dejando la descripción de la API del robot para la sección siguiente.

common

Este paquete contiene un conjunto de métodos generales comunes a todos los módulos, y contiene los siguientes métodos:

Método	cacheData
Parámetros de entrada	varname: string data: array
Retorno:	
Descripción:	
Almacena la variable de nombre *varname* y cuyo valor es *data* en la caché de usuario. Esta caché no es más que un fichero .mat compuesto por el identificador de usuario y un *timestamp* en el que se van concatenando todas las variables de las lecturas del robot siguiendo la siguiente nomenclatura: - aTIMESTAMP: lecturas de Arduino. - kTIMESTAMP: lecturas de Kinect. - xTIMESTAMP: localizaciones de las esquinas de los marcadores ArUco - rTIMESTAMP: vectores de rotación de los marcadores ArUco. - tTIMESTAMP: vectores de traslación de los marcadores ArUco. Siendo TIMESTAMP único para cada variable, excepto para la información de ArUco, cuyos datos de esquinas, rotaciones y traslaciones compartirán TIMESTAMP de forma que puedan identificarse como realizados en una misma medición. Mediante el método *saveEnvironment* se recorrerá este fichero concatenando todas las variables del mismo tipo en un mismo array, devolviendo ese fichero como resultado de la experiencia del usuario.	

Método	generateTimeStamp
Parámetros de entrada	asDouble: int
Retorno:	ts: string|double
Descripción:	

Genera una marca de tiempo con el formato yyyymmddHHMMSSFFF correspondiente al momento de ejecutar el método.

El valor de retorno será de tipo double si se pretende almacenar como valor y se indica con el valor 1 en el parámetro de entrada *asDouble*, o de tipo string en caso contrario, para ser utilizado como nombre de variable en el método *cacheData*.

Método	robotSetup
Parámetros de entrada	
Retorno:	status: int
Descripción:	

Configura el entorno del robot, arrancando los siguientes servicios:
- Servicio de streaming mjpgserver
- Servicio de conexión a Arduino
- Servicio de captura de marcadores ArUco
- Servicio de captura de imágenes de profundidad Kinect

Método	endSession
Parámetros de entrada	
Retorno:	status: int
Descripción:	

Finaliza la sesión, cerrando los siguientes servicios:
- Servicio de streaming mjpgserver
- Servicio de conexión a Arduino
- Servicio de captura de marcadores ArUco
- Servicio de captura de imágenes de profundidad Kinect

Método	executeOctaveCode
Parámetros de entrada	code: string
Retorno:	
Descripción:	

Ejecuta el código almacenado en user/usercode.m.
En caso de que code no sea una cadena vacía, almacena su contenido en ese fichero antes de ejecutar el fichero. No obstante, el fichero usercode.m debería ser escrito por el servicio RIPOctave, por lo que normalmente este parámetro no será necesario.

Método	setDebugLevel
Parámetros de entrada	debugLvl: int
Retorno:	
Descripción:	
Ajusta el nivel de detalle de las trazas entre 1 (sólo errores críticos) y 5 (mayor nivel de detalle).	

Tabla 11 - Descripción de los métodos del módulo octave.common

arduino

Este paquete contiene los métodos necesarios para hacer uso de la placa Arduino. A continuación, se detalla una tabla con la descripción de los métodos utilizados para ello.

Método	arduinoConnect
Parámetros de entrada	port: string baud: int
Retorno:	connected: int
Descripción:	
Abre el puerto serie y declara la globalmente el handler *serialStream* que será utilizado por aquellos métodos que requieran leer o escribir en la placa Arduino. El primer parámetro indica el puerto en el que se encuentra el puerto serie al que conectar (por defecto /dev/ttyACM0), mientras que el segundo parámetro permitirá indicar la velocidad de conexión, que deberá coincidir con la configurada en Arduino (por defecto, 9600). En caso de error, la función devolverá "0", mientras que un valor de "1" querrá decir que la conexión se ha realizado correctamente.	

Método	arduinoProcessInputs
Parámetros de entrada	action:string waitForResponse: double
Retorno:	result: int
Descripción:	
Escribe la acción *action* en Arduino mediante la función *arduinoSend* y actualiza los arrays globales que contienen los datos (*dataArray, messageArray*) con la información devuelta por *arduinoSend*, devolviendo finalmente el número de registros obtenidos.	

El valor waitForResponse indica el número de segundos que deben esperarse para realizar una lectura del puerto serie después de la escritura con el objetivo de no saturar las comunicaciones. Por defecto, su valor será 0.

Método	arduinoSend
Parámetros de entrada	data: string readAll: int waitForResponse: double
Retorno:	json: string
Descripción:	

Escribe la cadena *data* en el puerto serie invocando el método *arduinoSerialWrite* y a continuación solicita una lectura de los datos mediante *arduinoSerialRead* una vez transcurridos *waitForResponse* segundos desde la escritura.

Si readAll es 0, se solicitará únicamente el primer mensaje de la cola. En caso contrario, se realizará la lectura de todos los datos pendientes en el buffer de Arduino.

El valor de retorno *json* es una cadena en formato JSON con los datos obtenidos desde Arduino, cuyo formato es el siguiente:

```
{"DT":XX, "CD":XX, "CI":XX, "MP":XX, "MS":XX, "IrF":XX, "IrR":XX, "IrL":XX}
```

Método	arduinoSerialRead
Parámetros de entrada	readAll: int
Retorno:	json: string
Descripción:	

Solicita una lectura de los datos pendientes en el buffer de mensajes de Arduino.

Si readAll es 0, se solicitará únicamente el primer mensaje de la cola. En caso contrario, se realizará la lectura de todos los datos pendientes en el buffer de Arduino.

El valor de retorno *json* es una cadena en formato JSON con los datos obtenidos desde Arduino, cuyo formato es el siguiente:

```
{"DT":XX, "CD":XX, "CI":XX, "MP":XX, "MS":XX, "IrF":XX, "IrR":XX, "IrL":XX}
```

Método	arduinoSerialWrite
Parámetros de entrada	serialData: string
Retorno:	writtenBytes: int
Descripción:	

Escribe la cadena *serialData* en el puerto serie y devuelve el número de bytes escritos en la operación.

Método	getLastMessage
Parámetros de entrada	
Retorno:	message: string
Descripción:	

Devuelve el último mensaje escrito en el array global messageArray.

Método	sanitizeJSONObject
Parámetros de entrada	rawJSON: string
Retorno:	sanitizedJson: string
Descripción:	

Transforma el *pseudo-json* recibido como lectura de Arduino en un JSON válido.
Por ejemplo, dado el siguiente valor de *rawJSON*:

```
{DT=0, CD=17, CI=2, MP=0, MS=0, IrF=3, IrR=9, IrL=58}
```

Se obtendría el siguiente valor como retorno:

```
{"DT":0, "CD":17, "CI":2, "MP":0, "MS":0, "IrF":3, "IrR":9, "IrL":58}
```

Método	updateData
Parámetros de entrada	json: string
Retorno:	dataCount: int, messageCount: int
Descripción:	

Traduce un fichero JSON válido obtenido de Arduino y tratado mediante *sanitizeJSONObject()* en un conjunto de filas que serán incorporadas a los arrays *dataArray* y *messageArray*.

Por ejemplo, dado el siguiente valor de json:

```
[{"TS":86070809780785, "Mensaje": "READY"}, {"TS":123849072563436, "DT":0, "CD":17, "CI":2, "MP":0, "MS":0, "IrF":3, "IrR":9, "IrL":58}, {"TS":37486463457, "DT":10, "CD":167, "CI":28, "MP":30, "MS":30, "IrF":3656, "IrR":59, "IrL":518},{"TS":53447574556735, "Mensaje": "GENERAL ERROR"}]
```

Generaría los siguientes registros en dataArray y messageArray, respectivamente:

```
dataArray =  1.2385e+014   0   17   2   0   0   3   9   58
             3.7486e+010  10  167  28  30  30 3656 59  518
messageArray = {
    [1,1] =
    {
      [1,1] = 53447574556735
      [2,1] = READY
    }
    [1,2] =
    {
      [1,1] = 86070809780785
      [2,1] = GENERAL ERROR
    }
  }
```


Método	json2row
Parámetros de entrada	json: string
Retorno:	currentData: array, currentMessage: cell
Descripción:	
Traduce la cadena json en una fila de datos o en una celda con un mensaje. La cadena debe estar compuesta por un único mensaje. Este método es invocado por cada mensaje desde *updateData* una vez dividido todo el contenido de los datos obtenidos desde Arduino en mensajes independientes.	

Método	removeJsonDecorators
Parámetros de entrada	json: string
Retorno:	str: string
Descripción:	
Función auxiliar que elimina los delimitadores JSON (llaves, corchetes y comas) para realizar el *parseo* de los distintos mensajes obtenidos desde Arduino.	

Método	updateGlobals
Parámetros de entrada	data: array<double> message: cell
Retorno:	result: array<double>
Descripción:	
Extrae la información de la fila *data* y la vuelca en las variables globales TS, DT, CD, CI, MP, MS, IrF, IrR e IrL. Extrae la información de la celda *message* y la vuelca en las variables globales *TSM* y *Mensaje*. Devuelve el array [TS, DT, CD, CI, MP, MS, IrF, IrR, IrL], obtenido a partir de esas mismas variables globales.	

Tabla 12 - Descripción de los métodos del módulo octave.arduino

aruco

Este paquete contiene la funcionalidad necesaria para configurar y obtener información de los marcadores ArUco obtenidos a través de la cámara situada en la parte superior del robot comunicándose con el servidor *ArucoServer* a través del cliente *arucoclient*. En él sólo son necesarios dos métodos: uno para la configuración del servidor y otro para la obtención de datos de los marcadores.

Método	getMarkerInfo
Parámetros de entrada	captureImage: int
Retorno:	corners: array<double> rotations: array<double> translations: array<double>

Descripción:

Realiza una captura con la cámara situada sobre el robot y devuelve tres arrays con la información de los marcadores ArUco detectados:

- corners: cada fila contiene diez elementos que se corresponden con los siguientes datos:
  - 1: Timestamp de la captura.
  - 2: Identificador del marcador ArUco.
  - 3,4: Coordenadas (x,y) de la esquina inferior izquierda del marcador.
  - 5,6: Coordenadas (x,y) de la esquina inferior derecha del marcador.
  - 7,8: Coordenadas (x,y) de la esquina superior derecha del marcador.
  - 9,10: Coordenadas (x,y) de la esquina superior izquierda del marcador.
- rotations: cada fila contiene ocho elementos con la siguiente información:
  - 1: Timestamp de la captura.
  - 2: Identificador del marcador ArUco.
  - 3-5: Vector de rotación en representación eje-ángulo.
  - 6: Ángulo de alabeo expresado en radianes.
  - 7: Ángulo de cabeceo expresado en radianes.
  - 8: Ángulo de guiñada expresado en radianes.
- translations: cada fila contiene cinco elementos que se corresponden con los siguientes datos:
  - 1: Timestamp de la captura.
  - 2: Identificador del marcador ArUco.
  - 3: Coordenada X del vector de traslación.
  - 4: Coordenada Y del vector de traslación.
  - 5: Coordenada Z del vector de traslación.

La forma más sencilla de hacer uso de estos datos es calcular la posición mediante la media de las posiciones (x, y) de las esquinas y utilizar el campo rotations[8] (guiñada) como orientación, ya que el robot está diseñado para ser utilizado en una superficie plana paralela al techo.

La forma de calcular la posición (x,y) sería, por lo tanto:

```
x = (corners(3) + corners(5) + corners(7) + corners(9))/4;
y = (corners(4) + corners(6) + corners(8) + corners(10))/4;
```

Si el parámetro *captureImage* es 1, el robot realizará adicionalmente una captura en formato PNG en una localización determinada.

Método	setupArucoServer
Parámetros de entrada	shutdown: int
Retorno:	ready: int

Descripción:

Inicializa el servidor de captura de marcadores Aruco (*ArucoServer*) en caso de que no se encuentre en ejecución.

En caso de que se envíe el valor *shutdown=1*, forzará que el servidor se cierre.

Método	sendToAruco
Parámetros de entrada	operation: string
Retorno:	result: int output: string
Descripción:	
Envía la orden *operation* a *arucoclient*, devolviendo el código de salida del proceso (*result*) y su salida de consola (*output*). Las posibles operaciones a enviar son las indicadas en la descripción de la clase *arucoclient*.	

Tabla 13 – Descripción de los métodos del módulo octave.aruco

kinect

Este paquete contiene los métodos necesarios para configurar y obtener información del sensor de profundidad Kinect, comunicándose con *KinectDepthImageServer* a través del cliente *kinectclient*.

Método	getDepthImage
Parámetros de entrada	filter: string
Retorno:	Retrieved: int
Descripción:	
Solicita una imagen en formato PNG a Kinect, almacenándola en una ruta temporal determinada. En caso de que ya existiera una imagen previa, el método comprueba que la imagen solicitada no es la que ya existía, reintentando el proceso en caso de fallo. Este método hace uso de *kinectclient* para solicitar la imagen al servicio *KinectDepthImageServer*. En caso de querer un *colormap* distinto a *jet*, puede indicarse en el parámetro *filter*. Devuelve el valor 1 si la imagen se ha obtenido correctamente o 0 en caso contrario.	

Método	getDepthInfo
Parámetros de entrada	depth: int saveArray: int
Retorno:	depthArray: array<array<int8|int16>>

Descripción:
Solicita una matriz de profundidad a Kinect de 640x480, almacenándola en una ruta temporal determinada y devolviéndola como valor de retorno. Este método hace uso de *kinectclient* para solicitar el array al servicio *KinectDepthImageServer*. Si el parámetro *depth* recibe el valor *8*, devolverá una matriz de *int8* en lugar de *int16,* pero se perderá gran parte de la información. No obstante, este parámetro se permite para generar imágenes PNG en las que la resolución de la profundidad no es tan necesaria como en una medición real. Si el parámetro *saveArray* recibe el valor 1, la matriz se almacenará en la caché de usuario.

Método	isKinectReady
Parámetros de entrada	
Retorno:	ready: int
Descripción:	
Comprueba si el servidor KinectDepthImageServer está activo y aceptando peticiones, devolviendo 1 en tal caso o 0 en el contrario.	

Método	setupKinectServer
Parámetros de entrada	shutdown: int
Retorno:	ready: int
Descripción:	
Inicializa el servidor de captura de datos de profundidad mediante Kinect (*KinectDepthServer*) en caso de que no se encuentre en ejecución. En caso de que se envíe el valor *shutdown=1*, forzará que el servidor se cierre.	

Método	sendToKinect
Parámetros de entrada	operation: string
Retorno:	result: int output: string
Descripción:	
Envía la orden *operation* a *kinectclient*, devolviendo el código de salida del proceso (*result*) y su salida de consola (*output*). Las posibles operaciones a enviar son las indicadas en la descripción de la clase *kinectclient*.	

Tabla 14 - Descripción de los métodos del módulo octave.kinect

user

El módulo *user* contiene la funcionalidad básica para almacenar los datos que el usuario podrá descargarse, así como un fichero marco para realizar la ejecución del código del robot.

Método	getLastCachedDataFilePath
Parámetros de entrada	userId: string sourceDir: string
Retorno:	filePath: string
Descripción:	
Obtiene el último fichero de caché del usuario cuyo ID es *userId* de la ruta *sourceDir*. Este fichero se genera por el método *cacheData* como un conjunto de variables independientes por cada dato recuperado del robot, tanto de sus sensores como de Kinect y ArUco, y será tratado al finalizar la sesión por el método *saveEnvironment* aglutinando todos los datos del mismo tipo en una única matriz. El objetivo de separar los datos y escribirlos en disco nada más generarlos se debe al requisito de no perder los datos de la sesión en caso de desconexión, reinicio o cualquier otro problema. De este modo, si el usuario se desconecta fortuitamente del entorno, podrá generar el fichero .mat a partir del archivo de caché volviendo a acceder a la experiencia y pulsando el botón de descarga. No obstante, si el usuario vuelve a pulsar la pestaña de conexión, se generará un nuevo archivo de caché que ocultará el anterior. No obstante, los archivos de caché no se sobrescriben, sino que permanecen en la memoria del robot durante siete días, haciendo que puedan ser recuperables por el equipo docente en caso de necesidad.	

Método	saveEnvironment
Parámetros de entrada	source: string target: string
Retorno:	result: int
Descripción:	
Genera una matriz con toda la información recopilada por el usuario durante la sesión a partir del fichero de caché. Para ello, carga la matriz en la ruta *source* obtenida mediante *getLastCachedDataFilePath* e itera sobre todas sus variables, almacenándolas en una matriz dependiendo de la naturaleza del dato:	

- **robotData**: <nx9 double>: almacena las lecturas del robot. En esta matriz se insertarán aquellas variables con el formato ayyyymmddHHMMSSFFF. El significado de cada posición es la siguiente:
    - 1: TS - Marca de tiempo en formato yyyymmddHHMMSSFFF
    - 2: DT – Tiempo desde la última lectura
    - 3: CD – Número de cuentas del encoder derecho
    - 4: CI – Número de cuentas del encoder izquierdo
    - 5: MP – Estado del motor (1=encendido, 0=apagado)
    - 6: MS – Estado del láser (1=encendido, 0=apagado)
    - 7: IrF – Distancia en cm detectada por el sensor infrarrojo frontal
    - 8: IrR - Distancia en cm detectada por el sensor infrarrojo derecho
    - 9: IrL - Distancia en cm detectada por el sensor infrarrojo izquierdo

- **robotDepthImages**: <nx480x640 double>: almacena las matrices de profundidad capturadas por Kinect.

- **robotDepthImageTimestamps**: <nx1 double>: almacena las marcas de tiempo de las capturas de profundidad, que se correlacionan por el índice. Esto es, a la captura robotDepthImages(5,:,:) le corresponderá la marca de tiempo robotDepthImageTimestamps(5).

- **cornerData**: <nx10 double>: almacena la información de las esquinas de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición y el resto de campos con el valor NaN. Los índices tienen el siguiente significado:
    - 1: Timestamp de la captura en formato yyyymmddHHMMSSFFF.
    - 2: Identificador del marcador ArUco.
    - 3,4: Coordenadas (x,y) de la esquina inferior izquierda del marcador.
    - 5,6: Coordenadas (x,y) de la esquina inferior derecha del marcador.
    - 7,8: Coordenadas (x,y) de la esquina superior derecha del marcador.
    - 9,10: Coordenadas (x,y) de la esquina superior izquierda del marcador.

- **rotationData**: <nx8 double>: cada fila contiene ocho elementos con la información referente a la rotación de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición y el resto de campos con el valor NaN. Los índices tienen el siguiente significado:
    - 1: Timestamp de la captura.
    - 2: Identificador del marcador ArUco.
    - 3-5: Vector de rotación en representación eje-ángulo.
    - 6: Ángulo de alabeo expresado en radianes.
    - 7: Ángulo de cabeceo expresado en radianes.
    - 8: Ángulo de guiñada expresado en radianes.

- **translationData**: <nx5>: cada fila contiene cinco elementos con la información referente a la traslación de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un

registro con la marca de tiempo de la petición y el resto de campos con el valor NaN. Los índices tienen el siguiente significado: - 1: Timestamp de la captura. - 2: Identificador del marcador ArUco. - 3: Coordenada X del vector de traslación. - 4: Coordenada Y del vector de traslación. - 5: Coordenada Z del vector de traslación. La información obtenida por ArUco puede correlacionarse mediante (*timestamp, id)*, ya que una misma captura compartirá estos dos valores en las tres matrices proporcionadas por ArUco (cornerData, rotationData y translationData).

Método	usercode
Parámetros de entrada	
Retorno:	
Descripción:	
Este fichero alojará el código generado por el usuario, y será escrito en disco cuando el usuario pulse la opción de realizar el envío al robot.	

Tabla 15 – Descripción de los métodos del módulo octave.user

2.4.3. API del robot

Pese a que las siguientes descripciones forman también parte del diseño estático de la solución, la API del robot se detalla en una sección aparte para facilitar su localización y facilitación a los posibles usuarios del robot.

Las funciones de la API se encuentran en la carpeta raíz (/octave), y permiten al usuario interactuar directamente con el robot además de usar los métodos propios de Octave. Las funciones que el usuario puede utilizar directamente son las siguientes:

Método	calculateMovement
Parámetros de entrada	rightCounts: int leftCounts: int currentX: double currentZ: double currentOrientation: double
Retorno:	x: double z: double orientation: double xDistance: double zDistance: double turnAngle: double
Descripción:	

Calcula la posición, orientación y distancia recorrida por el robot aplicando la teoría del modelo diferencial aplicando una distancia entre centros de 28 cm.

Para ello, se le debe proporcionar a la función los siguientes datos:
- rightCounts: cuentas leídas por el encoder derecho.
- leftCounts: cuentas leídas por el encoder izquierdo.
- currentX: coordenada X de la posición actual del robot (derecha-izquierda)
- currentZ: coordenada Z de la posición actual del robot (atrás-adelante)
- currentOrientation: ángulo actual de la orientación del robot

Es importante tener en cuenta que el eje de coordenadas en esta función está adaptado a Kinect, en el que la profundidad está dada por el eje Z, por lo que el eje X se determinará en la dirección derecha-izquierda y el eje Y en la dirección abajo-arriba.

Como resultado del cálculo, se proporcionan los siguientes datos:
- x: coordenada x del robot tras el movimiento
- z: coordenada z del robot tras el movimiento
- orientation: orientación tras el movimiento
- xDistance: distancia recorrida en el eje X
- zDistance: distancia recorrida en el eje Z
- turnAngle: ángulo girado al efectuar la trayectoria

Este método tiene como objetivo facilitar al alumno una forma de comprobar de forma teórica las posiciones y orientaciones que el robot debería adquirir en un entorno ideal. No obstante, debido a las imprecisiones de las medidas y de la ejecución del movimiento (por ejemplo, derrape de una de las ruedas), será necesario implementar métodos más complejos por su parte para adaptarlo a un entorno realista.

Tabla 16 - Descripción del método de la API calculateMovement

Método	calculateTurnAngle
Parámetros de entrada	innerRadius: double outerRadius: double innerArcLength: double outerArcLength: double
Retorno:	turnAngle: double
Descripción:	
Este método se utiliza de forma interna por el método calculateMovement, calculando el ángulo de giro a partir de los siguientes parámetros obtenidos a través de la teoría del modelo diferencial: - innerRadius: radio de la circunferencia trazada por la rueda interior respecto a un centro de giro imaginario. - outerRadius: radio de la circunferencia trazada por la rueda exterior respecto a un centro de giro imaginario. - innerArcLength: distancia recorrida por la rueda interior. - outerArcLength: distancia recorrida por la rueda exterior. El valor de retorno se corresponderá con el ángulo de giro del robot.	

Tabla 17 - Descripción del método de la API calculateTurnAngle

Método	depthMatToCm
Parámetros de entrada	depthArray: <480x640 double>
Retorno:	<480x640 double>
Descripción:	
Transforma los valores de profundidad de una matriz capturada por Kinect (cuyo rango se encuentra entre 0 y 2047) a distancias reales expresadas en centímetros. Los valores inválidos, representados por el valor 2047 en la matriz de profundidad adquirirán en su transformación a centímetros un valor de 0.	

Tabla 18 - Descripción del método de la API depthMatToCm

Método	encoderCountsToCm
Parámetros de entrada	counts: int
Retorno:	d: double
Descripción:	
Calcula la distancia recorrida por una rueda a partir del número de cuentas leídas por el encoder asumiendo un radio de 6,2 cm.	

Tabla 19 - Descripción del método de la API encoderCountsToCm

Método	getDepth
Parámetros de entrada	saveArray: int
Retorno:	<480x640 double>
Descripción:	

Solicita una imagen de profundidad al dispositivo Kinect, devolviendo una matriz de 480 filas por 640 columnas con valores comprendidos entre 0 y 2047, siendo 2047 el valor equivalente a "lectura incorrecta".

Opcionalmente, es posible no almacenar la matriz en la caché de usuario asignándole el valor 0 al parámetro saveArray. En caso contrario, el contenido de la matriz será almacenado en caché para su posterior descarga por parte del alumno.

Tabla 20 - Descripción del método de la API getDepth

Método	getMarkers
Parámetros de entrada	saveArrays: int
Retorno:	corners: <nx10 double> rotation_vectors: <nx8 double> translation_vectors: <nx5 double>
Descripción:	

Solicita una lectura a la cámara situada en la parte superior del robot, obteniendo tres arrays con n filas cada uno, siendo n el número de marcadores ArUco detectados.

Las matrices devueltas como valor de retorno son las siguientes:

- **corners**: <nx10 double>: almacena la información de las esquinas de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición, almacenando el valor NaN en el resto de campos. Los índices tienen el siguiente significado:
  - 1: Timestamp de la captura en formato yyyymmddHHMMSSFFF.
  - 2: Identificador del marcador ArUco.
  - 3,4: Coordenadas (x,y) de la esquina inferior izquierda del marcador.
  - 5,6: Coordenadas (x,y) de la esquina inferior derecha del marcador.
  - 7,8: Coordenadas (x,y) de la esquina superior derecha del marcador.
  - 9,10: Coordenadas (x,y) de la esquina superior izquierda del marcador.

- **rotation_vectors**: <nx8 double>: cada fila contiene ocho elementos con la información referente a la rotación de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición, almacenando el valor NaN en el resto de campos. Los índices tienen el siguiente significado:

- 1: Timestamp de la captura en formato yyyymmddHHMMSSFFF.
- 2: Identificador del marcador ArUco.
- 3-5: Vector de rotación en representación eje-ángulo.
- 6: Ángulo de alabeo expresado en radianes.
- 7: Ángulo de cabeceo expresado en radianes.
- 8: Ángulo de guiñada expresado en radianes.

- **translation_vectors**: <nx5>: cada fila contiene cinco elementos con la información referente a la traslación de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición, almacenando el valor NaN en el resto de campos:
  - 1: Timestamp de la captura en formato yyyymmddHHMMSSFFF.
  - 2: Identificador del marcador ArUco.
  - 3: Coordenada X del vector de traslación.
  - 4: Coordenada Y del vector de traslación.
  - 5: Coordenada Z del vector de traslación.

La información obtenida por ArUco puede correlacionarse mediante (*timestamp, id)*, ya que una misma captura compartirá estos dos valores en las tres matrices proporcionadas por ArUco (corners, rotation_vectors y translation_vectors).

La forma más sencilla de hacer uso de estos datos es calcular la posición mediante la media de las posiciones (x, y) de las esquinas y utilizar el campo rotation_vectors(8) (yaw/guiñada) como orientación, ya que el robot está diseñado para ser utilizado en una superficie plana paralela al techo en el que no debería producirse alabeo ni cabeceo.

La forma de calcular la posición (x,y) sería, por lo tanto:

```
x = (corners(3) + corners(5) + corners(7) + corners(9))/4;
y = (corners(4) + corners(6) + corners(8) + corners(10))/4;
```

Opcionalmente, es posible no almacenar las matrices en la caché de usuario asignándole el valor `0` al parámetro *saveArrays*. En caso contrario, el contenido de las matrices será almacenado en caché para su posterior descarga por parte del alumno.

Tabla 21 - Descripción del método de la API getMarkers

Método	getReadings
Parámetros de entrada	from: int to: int
Retorno:	readings: <nx9 double>
Descripción:	

Obtiene los registros obtenidos de Arduino comprendidos entre 'from' y 'to'.

A continuación, se indica el significado de cada columna de la matriz:

- 1: TS - Marca de tiempo en formato yyyymmddHHMMSSFFF.
- 2: DT – Tiempo desde la última lectura.
- 3: CD – Número de cuentas del encoder derecho.
- 4: CI – Número de cuentas del encoder izquierdo.
- 5: MP – Estado del motor (1=encendido, 0=apagado).
- 6: MS – Estado del láser (1=encendido, 0=apagado).
- 7: IrF – Distancia en cm detectada por el sensor infrarrojo frontal.
- 8: IrR - Distancia en cm detectada por el sensor infrarrojo derecho.
- 9: IrL - Distancia en cm detectada por el sensor infrarrojo izquierdo.

En caso de no proporcionar ningún parámetro, se devolverá un array con todos los registros obtenidos desde Arduino.

En caso de no proporcionar el parámetro 'to', se devolverán todas las lecturas desde 'from'.

En caso de proporcionar un valor <= 1 para 'from', se devolverán todas las lecturas hasta 'to'.

En caso de proporcionar un valor superior al número total de registros para 'to', se devolverán todas las lecturas desde 'from' hasta 'to'.

Ejemplos:
- getReadings(): devolverá todos los registros.
- getReadings(10): devolverá todos los registros a partir del décimo.
- getReadings(0,10): devolverá los diez primeros registros.
- getReadings(5,10): devolverá los registros comprendidos entre el 5 y 10.

Tabla 22 - Descripción del método de la API getReadings

Método	getTotalNumberOfReadings
Parámetros de entrada	
Retorno:	readings: int
Descripción:	
Devuelve el número total de lecturas realizadas hasta el momento por Arduino.	

Tabla 23 - Descripción del método de la API getTotalNumberOfReadings

Método	keepAlive
Parámetros de entrada	
Retorno:	result: int
Descripción:	
Indica a Arduino que mantenga el motor encendido, evitando la parada por timeout. Adicionalmente, solicita a Arduino una lectura, almacenando la información de los sensores en el array de datos. Devuelve como valor de retorno el número de bytes escritos en la placa Arduino.	

Tabla 24 - Descripción del método de la API keepAlive

Método	moveEngines
Parámetros de entrada	signalLeft: int signalRight: int delay: double
Retorno:	result: int
Descripción:	
Envía la señal *signalLeft* al motor izquierdo y la señal *signalRight* al motor derecho, debiendo estar ambos valores entre -255 y 255. El parámetro *delay* configura el tiempo transcurrido entre el envío de la orden de escritura y la posterior orden de lectura de los sensores. Si no se proporciona este parámetro, adquirirá el valor de max(signalLeft, signalRight)/150 segundos. En caso de querer omitir la orden de lectura, habrá que pasar el valor 0. La ejecución de este método hará que el robot se mueva según la señal enviada a cada rueda. **Ejemplos:** *moveEngines(210, 210): hace que el robot se desplace hacia adelante.* *moveEngines(-210, -210, 0): hace que el robot se desplace hacia atrás, omitiendo la lectura de los sensores.* *moveEngines(-210, 210, 1): hace que el robot gire hacia la izquierda, dejando un segundo entre la orden de escritura y la orden de lectura.*	

moveEngines(210, -210): hace que el robot gire hacia la derecha. Devuelve como valor de retorno el número de bytes escritos en la placa Arduino.

Tabla 25 - Descripción del método de la API moveEngines

Método	moveForward
Parámetros de entrada	signal: int delay: double
Retorno:	result: int
Descripción:	
Envía la señal *signal* a ambos motores. Equivale a ejecutar el método *moveEngines(signal, signal)*. El parámetro *delay* configura el tiempo transcurrido entre el envío de la orden de escritura y la posterior orden de lectura de los sensores. Si no se proporciona este parámetro, adquirirá el valor de signal/150 segundos. En caso de querer omitir la orden de lectura, habrá que pasar el valor 0. La ejecución de este método hará que el robot se mueva según la señal enviada a cada rueda. **Ejemplos:** *moveForward(210): hace que el robot se desplace hacia adelante.* *moveForward(-210,0): hace que el robot se desplace hacia atrás, omitiendo la orden de lectura tras la escritura en el puerto serie.* *moveForward(): equivale a ejecutar moveForward(100).* Devuelve como valor de retorno el número de bytes escritos en la placa Arduino.	

Tabla 26 - Descripción del método de la API moveForward

Método	moveLeftEngine
Parámetros de entrada	signal: int delay: double
Retorno:	result: int
Descripción:	
Envía la señal *signal* al motor izquierdo. Equivale a ejecutar el método *moveEngines(signal, 0)*.	

El parámetro *delay* configura el tiempo transcurrido entre el envío de la orden de escritura y la posterior orden de lectura de los sensores. Si no se proporciona este parámetro, adquirirá el valor de signal/150 segundos. En caso de querer omitir la orden de lectura, habrá que pasar el valor 0.

La ejecución de este método hará que el robot se mueva según la señal enviada a cada rueda.

Ejemplos:
moveLeftEngine(210): hace que el robot gire hacia la derecha.
moveLeftEngine(-210, 0): hace que el robot gire hacia la izquierda, omitiendo la orden de lectura de los valores de los sensores.

NOTA: si se envía una señal a un motor con suficiente potencia, es probable que la tracción haga que se produzca movimiento en la otra rueda, pese a no aplicarle ninguna señal de forma activa.

Devuelve como valor de retorno el número de bytes escritos en la placa Arduino.

Tabla 27 - Descripción del método de la API moveLeftEngine

Método	moveRightEngine
Parámetros de entrada	signal: int delay: double
Retorno:	result: int
Descripción:	

Envía la señal *signal* al motor derecho. Equivale a ejecutar el método *moveEngines(0, signal*).

El parámetro *delay* configura el tiempo transcurrido entre el envío de la orden de escritura y la posterior orden de lectura de los sensores. Si no se proporciona este parámetro, adquirirá el valor de signal/150 segundos. En caso de querer omitir la orden de lectura, habrá que pasar el valor 0. La ejecución de este método hará que el robot se mueva según la señal enviada a cada rueda.

Ejemplos:
moveRightEngine(210): hace que el robot gire hacia la izquierda.
moveRightEngine(-210,0): hace que el robot gire hacia la derecha, omitiendo la orden de lectura de los valores de los sensores

NOTA: si se envía una señal a un motor con suficiente potencia, es probable que la tracción haga que se produzca movimiento en la otra rueda, pese a no aplicarle ninguna señal de forma activa.
Devuelve como valor de retorno el número de bytes escritos en la placa Arduino.

Tabla 28 - Descripción del método de la API moveRightEngine

Método	setExecutionTimeout
Parámetros de entrada	secs: int
Retorno:	
Descripción:	
Debido a que la ejecución del código Octave tiene un timeout por defecto de 60 segundos, es posible que el timeout se produzca antes de que el código haya terminado de ejecutarse. Añadiendo una llamada a *setExecutionTimeout* en la primera línea del código hará que el timeout se modifique para la ejecución actual, pudiendo ampliarla hasta un máximo de 900 segundos (15 minutos). NOTA: es necesario que la llamada a setExecutionTimeout se realice en la primera línea de código para que su ejecución sea efectiva.	

Tabla 29 - Descripción del método de la API setExecutionTimeout

Método	stopEngines
Parámetros de entrada	
Retorno:	result: int
Descripción:	
Indica a Arduino que detenga los motores. Adicionalmente, solicita a Arduino una lectura, almacenando la información de los sensores en el array de datos, por lo que puede utilizarse para forzar la actualización de los datos de los sensores. Devuelve como valor de retorno el número de bytes escritos en la placa Arduino.	

Tabla 30 - Descripción del método de la API stopEngines

2.4.4. Diseño dinámico

A continuación, se muestra una serie de diagramas de secuencia con el desarrollo de los casos de uso definidos en la sección correspondiente, *Casos de uso*:

2.4.4.1. Conectar al robot

Una vez que el usuario realiza la conexión a UNILabs y accede a la experiencia, se cargará la interfaz EjsS que permite realizar una conexión al robot.

Cuando el usuario se conecta al robot, el servidor RIPOctave recoge la petición y ejecuta el método *start()*, invocando el método *octave.common.robotSetup*, encargado de levantar los correspondientes servicios de streaming, captura de imágenes de profundidad y de marcadores Aruco.

Adicionalmente, se crea un fichero de caché para el ID de usuario y el timestamp de inicio de la conexión en el que se volcará toda la información correspondiente a la sesión actual.

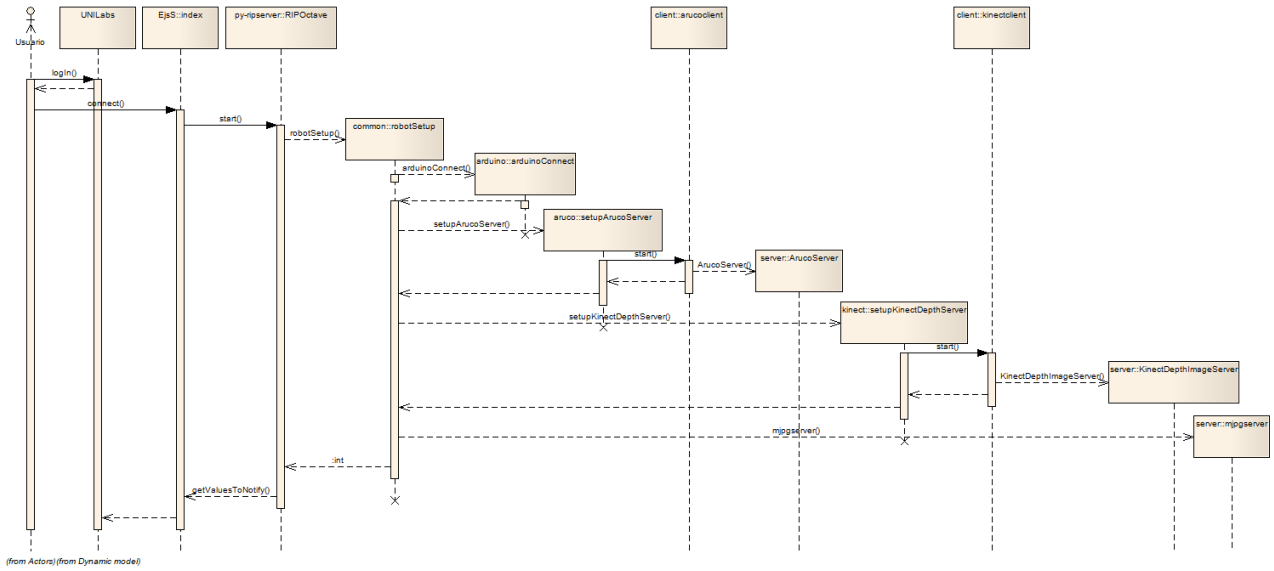


Figura 26 - Diagrama de secuencia del caso de uso 'Conectar al robot'

2.4.4.2. *Obtener telemetría*

El envío de una orden manual a través de las flechas de dirección del robot hace que se ejecute el método *set* de RIPOctave enviándole un valor a la variable *currentAction*.

Esto hará que se ejecute el método *octave.arduino.arduinoProcessInputs* enviándole *currentAction* como parámetro, provocando el movimiento del robot y la recuperación de la telemetría por parte de éste, que es actualizada por el método *getValuesToNotify* de RIPOctave y enviada al usuario, actualizando la información visualizada en el cuadro de control y en el log del sistema.

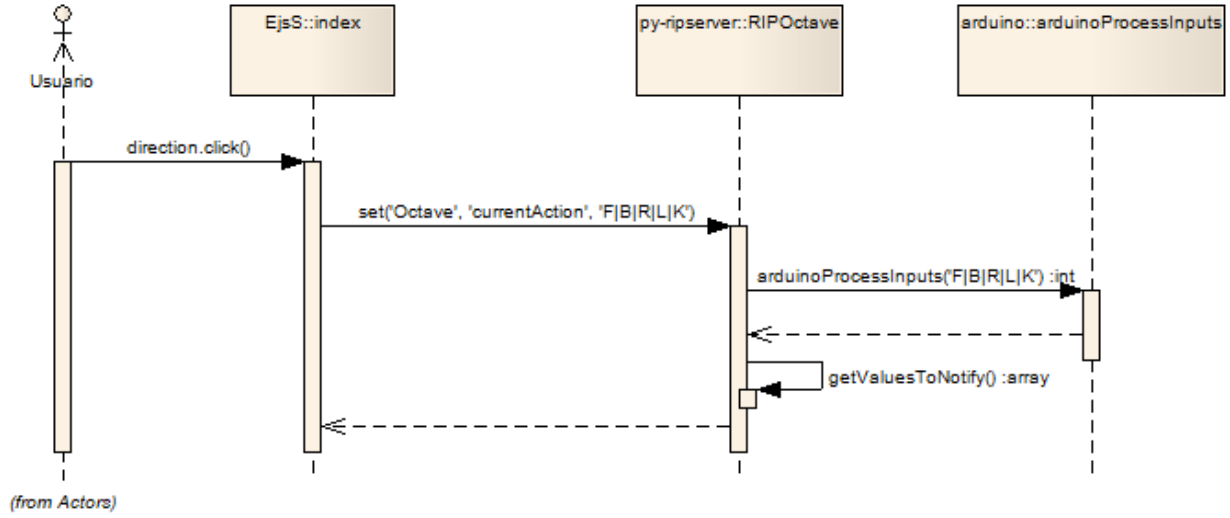


Figura 27 – Diagrama de secuencia del caso de uso "Obtener telemetría"

2.4.4.3. *Visualizar imagen 3D*

En una de cada tres solicitudes de datos a *RIPServer*, el robot solicitará a Kinect una imagen de profundidad invocando el método *octave.kinect.getDepthImage*, que hará uso de *kinectclient* para indicarle a *KinectDepthImageServer* que almacene la imagen en una ruta temporal, la cual será codificada en formato base64 a través del método *getDepthImageBase64* y enviada al usuario en el array de valores de retorno de *getValuesToNotify* con la clave *KinectImageBase64*, donde será renderizada en la interfaz como una imagen de 320x240px junto al streaming principal.

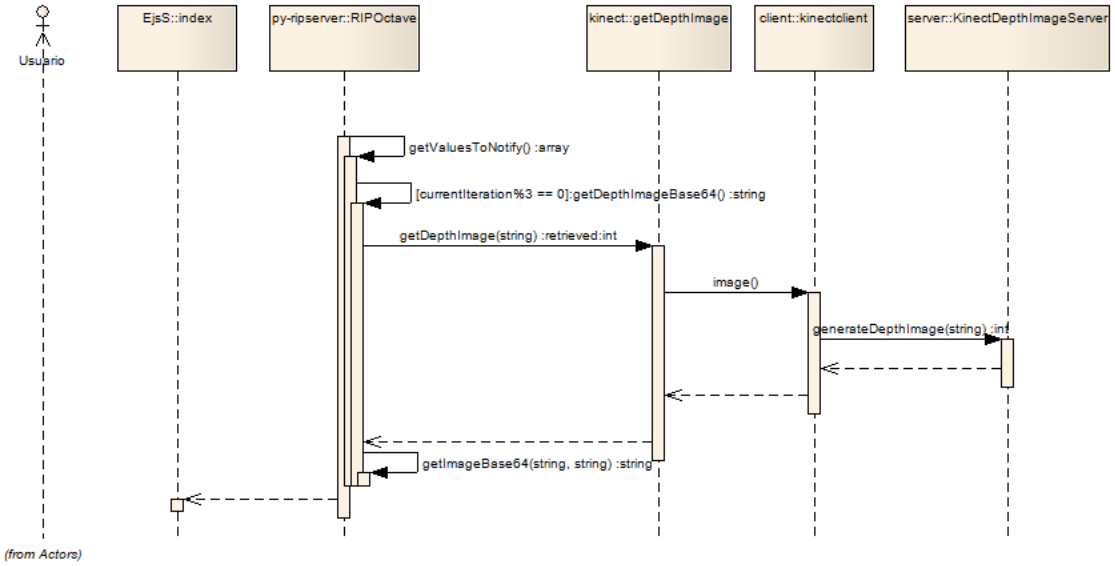


Figura 28 – Diagrama de secuencia del caso de uso 'Visualizar imagen 3D'

2.4.4.4. *Visualizar marcadores ArUco*

En una de cada tres solicitudes de datos a *RIPServer*, el robot solicitará a la cámara de la parte superior del robot una captura junto a la información de los posibles marcadores ArUco invocando el método *octave.aruco.getMarkerInfo*, que hará uso de *arucoclient* para indicarle a *ArucoServer* que almacene la imagen en una ruta temporal, la cual será codificada en formato base64 a través del método *getDepthImageBase64* y enviada al usuario en el array de valores de retorno de *getValuesToNotify* con la clave *CeilingImageBase64*, donde será renderizada en la interfaz como una imagen de 320x240px junto al streaming principal.

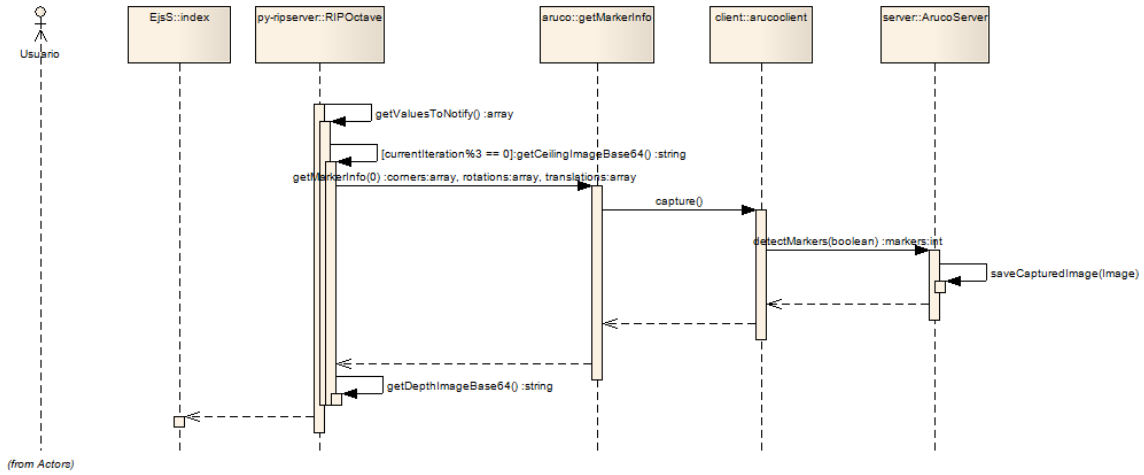


Figura 29 – Diagrama de secuencia del caso de uso 'Visualizar marcadores ArUco'

2.4.4.5. *Guardar código en el espacio de trabajo*

Mediante la pulsación del botón correspondiente de la interfaz EjsS se invocará el método *uploadCodeToWorkspace*, que invocará a su vez el método *_saveText* de UNILabs, permitiendo almacenar en el espacio de trabajo el contenido actual del editor de texto.

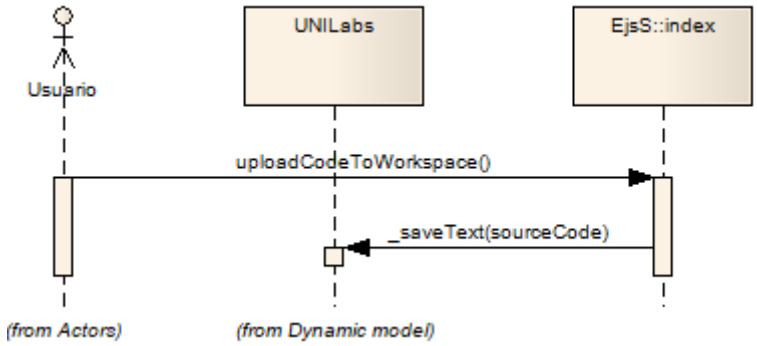


Figura 30 - Diagrama de secuencia del caso de uso 'Guardar código en el espacio de trabajo'

2.4.4.6. *Recuperar código del espacio de trabajo*

Mediante la pulsación del botón correspondiente de la interfaz EjsS se invocará el método *downloadCodeFromWorkspace*, que invocará a su vez el método *_readText* de UNILabs, permitiendo cargar en el editor de texto el contenido de un fichero previamente almacenado en el espacio de trabajo del alumno.

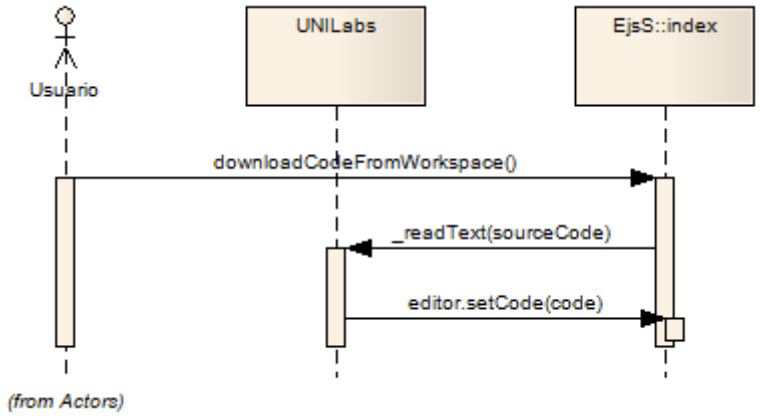


Figura 31 - Diagrama de secuencia del caso de uso 'Recuperar código del espacio de trabajo'

2.4.4.7. Descargar resultados de la experiencia

Una vez que el usuario se desconecte de la sesión, se visualizará un botón que permitirá generar y descargar en formato *.mat* el resultado de las lecturas realizadas durante la experiencia. Este fichero *.mat* contendrá un conjunto de variables, cuya estructura y contenido puede consultarse en la descripción del método *saveEnvironment*, localizada en la *Tabla 15 – Descripción de los métodos del módulo octave.user*.

La obtención de estos datos es única por usuario y sesión, dado que el fichero temporal de caché a partir del cual se genera se define por el identificador del usuario de UNILabs y por la marca de tiempo en la que se inicia la sesión.

Si el usuario cierra la sesión y no descarga inmediatamente el fichero de resultados podrá hacerlo sin problema accediendo nuevamente a la experiencia y pulsando el botón de descarga de resultados, ya que éste fichero se generará siempre a partir de los datos de la última experiencia del usuario. No obstante, si el usuario pulsa el botón *conectar*, se generará un nuevo fichero de caché que ocultará el contenido de la sesión anterior, por lo que será inaccesible para el usuario. En cualquier caso, el equipo docente podrá recuperar el archivo de caché (por defecto, */var/robot/cache*) durante los siete días siguientes a la ejecución de la experiencia, momento a partir del cual el fichero de caché será eliminado automáticamente del sistema por el servicio *oldmatcleaner*.

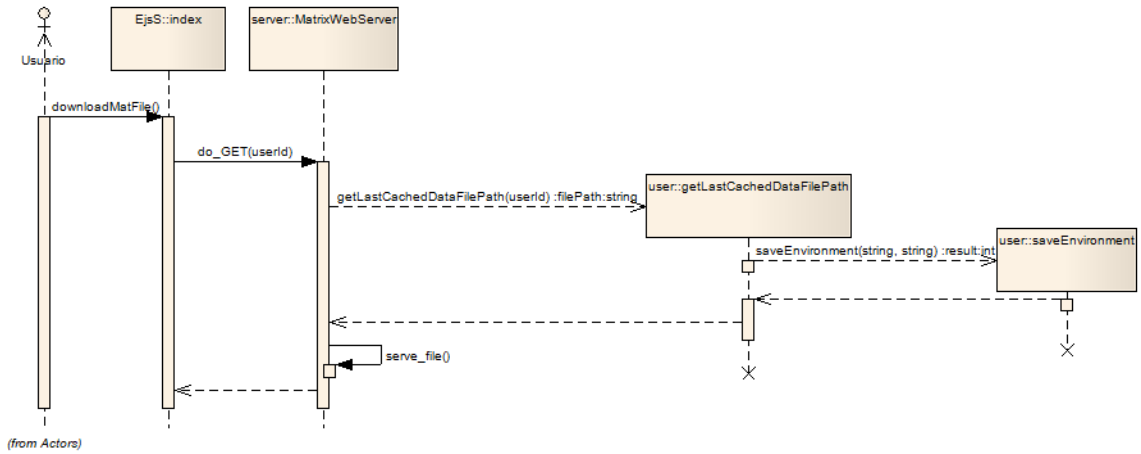


Figura 32 - Descripción del caso de uso 'Descargar resultados de la experiencia'

2.4.4.8. *Ejecutar código remoto*

La pulsación del botón de ejecución de código enviará el contenido del editor de texto a RIPOctave dentro de la variable *octaveCode*, haciendo que éste se almacene en */opt/robot/octave/user/usercode.m*, invocando a continuación el método correspondiente a la ejecución de código a través de octave.

Esta operación es bloqueante, por lo que durante la ejecución dejará de recibirse información actualizada del robot hasta su completa finalización (exceptuando el streaming de vídeo, que sí se mostrará en tiempo real).

La ejecución del código tiene un *timeout* de 60 segundos, ampliables hasta 900 segundos indicando como primera operación del código el método *setExecutionTimeout(segundos)*. No obstante, esta operación debe utilizarse con cautela, ya que en caso de codificar una situación de bloqueo dejará inoperativo el robot durante el tiempo especificado como parámetro.

Una vez finalizada la ejecución, el *timeout* volverá a situarse en 60 segundos, por lo que deberá volver a incluirse en la primera línea de código para que sea aplicable.

La ejecución hará que las capturas de datos se almacenen en la caché, pudiendo ser descargadas tras la desconexión pulsando en el botón correspondiente, tal y como se indica en la *Figura 31 - Diagrama de secuencia del caso de uso 'Recuperar código del espacio de trabajo'*.

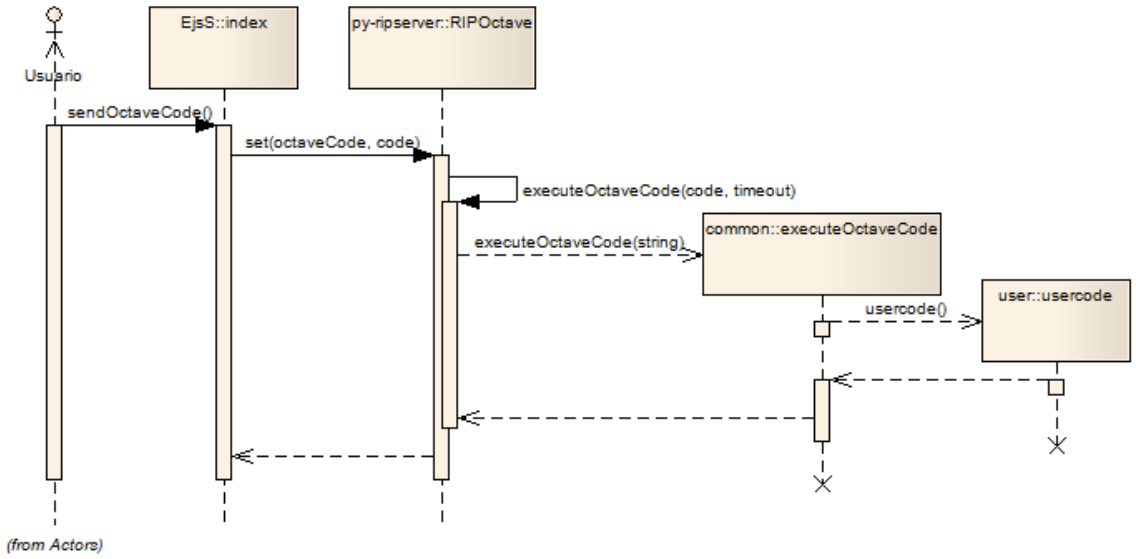


Figura 33 - Descripción del caso de uso 'Ejecutar código remoto'

El código enviado al robot puede, adicionalmente, realizar los siguientes casos de uso:

2.4.4.9. Mover robot

Provoca el movimiento del robot a través de uno de los siguientes métodos:

- moveEngines(signalLeft, signalRight)
- moveForward(signal)
- moveLeftEngine(signal)
- moveRightEngine(signal)
- keepAlive()
- stopEngines()

Cada uno de estos métodos realiza una operación cinemática sobre el robot. Estos métodos pueden consultarse en la sección *API del robot*.

El diagrama de secuencia de cada uno de estos métodos tiene un aspecto similar al detallado a continuación para el método *moveEngines*:

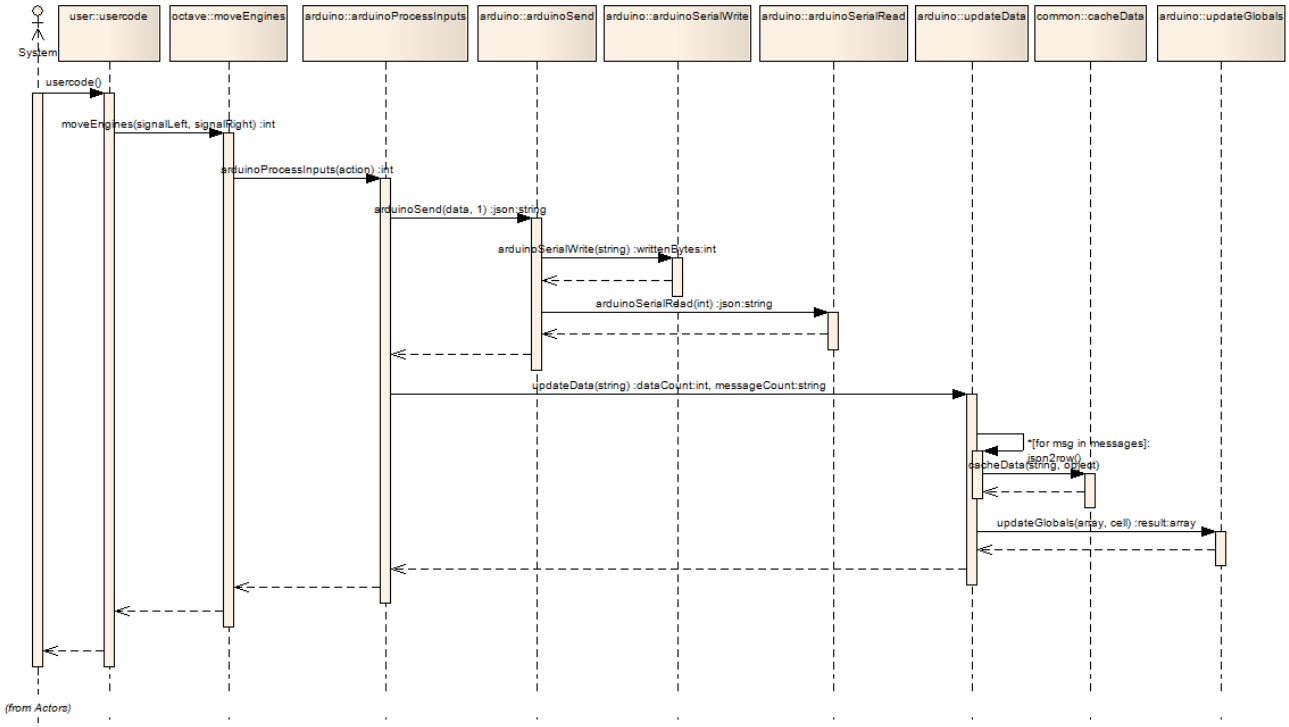


Figura 34 - Descripción del caso de uso 'Mover robot'

El proceso de movimiento del robot es, como puede observarse, el de mayor complejidad de la aplicación, ya que afecta a múltiples elementos del sistema.

El escenario comienza con la invocación del método que efectúa el movimiento (en este caso, *moveEngines*), al que se le indican los valores de las señales PWM que se enviarán a los motores. Esto se hace mediante la invocación de *arduinoProcessInputs*, que a su vez ejecuta el método *arduinoSend* pasándole la cadena de caracteres que se enviará a Arduino mediante el método *arduinoSerialWrite*.

Una vez ejecutada la secuencia, se invoca el método *serialRead* tras una pausa de *señal/150* segundos para obtener la lectura de los datos generados por la placa arduino en formato JSON (modificable indicando el retardo como parámetro, o *0* para no realizar la lectura) y se envían al método *updateData* para su conversión a una fila de la matriz global de datos y su cacheo en el fichero de sesión. Hecho esto, se actualizan los valores de las variables globales con las lecturas obtenidas desde Arduino.

2.4.4.10. Obtener información de marcadores

Permite la captura de información de marcadores ArUco situados en el techo mediante la cámara situada sobre el robot.

El método *getMarkers()* invoca el método *getMarkerInfo()*, ejecutando *arucoclient* con el parámetro *detect*, que será enviado a *ArucoServer*, el cual realizará la captura y tratamiento de los datos de las esquinas, rotaciones y traslaciones.

Una vez que los datos han sido capturados, el método de la API *getMarkers()* añadirá a la caché de usuario aquellos datos obtenidos de la cámara.

En caso de que no se detecte ningún marcador, se guardará igualmente un registro de cada tipo (esquinas, vectores de rotación y vectores de traslación), pero con todos sus valores a NaN, exceptuando la marca de tiempo. Esto simbolizará que en el momento TS no existía (o al menos, no se detectó) ningún marcador, contribuyendo con esta información al proceso de localización del robot.

El proceso de captura de la información se realiza mediante el almacenamiento de los datos en un fichero .mat, que será leído por las funciones de octave una vez que se hayan generado.

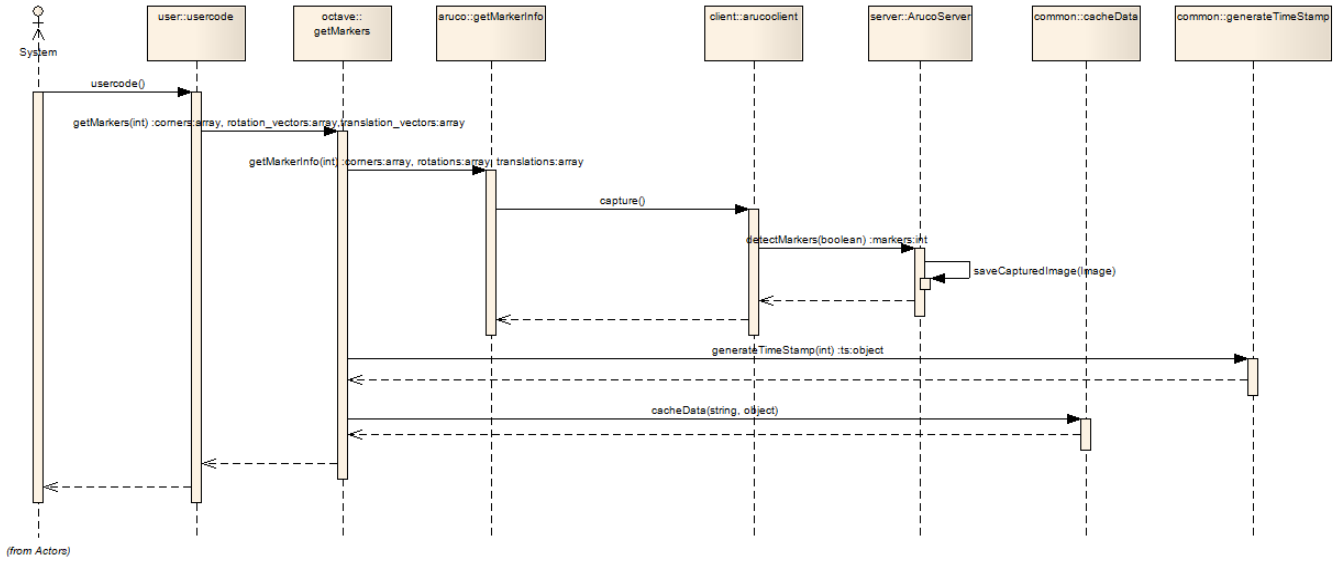


Figura 35 - Descripción del caso de uso 'Obtener información de marcadores'

2.4.4.11. *Obtener imagen de profundidad*

Permite la captura de imágenes de profundidad desde el dispositivo Kinect, almacenándola como una matriz de 480 filas x 640 columnas en formato *double*, con valores comprendidos entre 0 y 2047, simbolizando éste último valor un fallo en la lectura de la posición.

El proceso de captura de la información se realiza mediante el almacenamiento de los datos en un fichero .mat, que será leído por las funciones de octave una vez que se hayan generado.

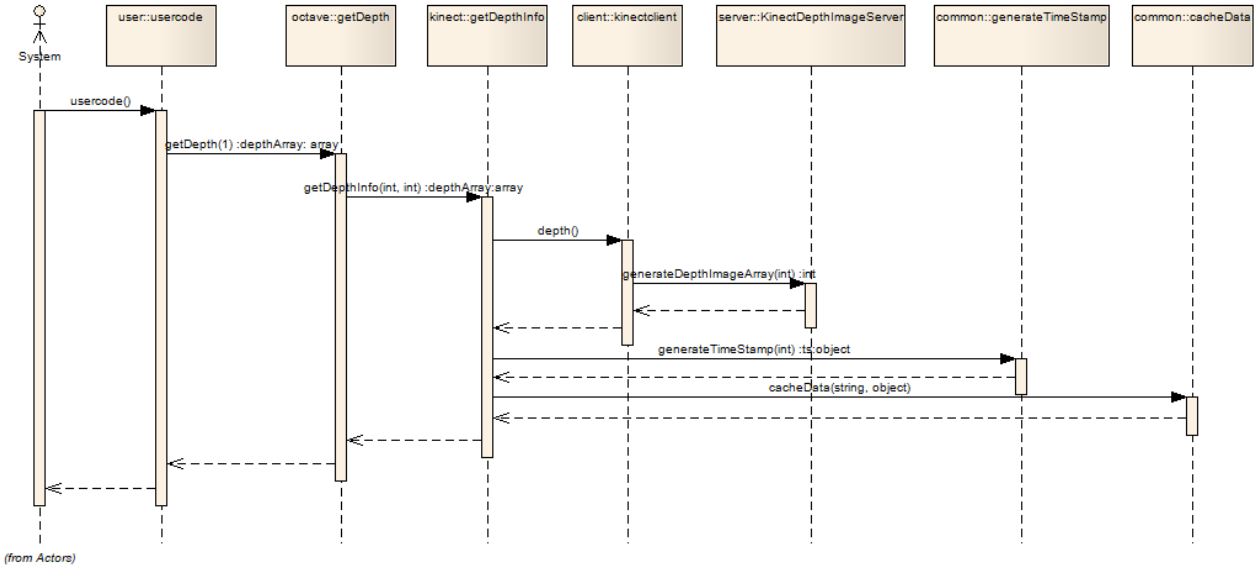


Figura 36 - Descripción del caso de uso 'Obtener imagen de profundidad'

2.5. Software

La presente sección trata de describir el software utilizado durante la elaboración del proyecto, así como la interconexión de los distintos componentes entre sí a la hora de implementar la solución.

El siguiente esquema muestra la dependencia y comunicación entre los distintos elementos software descritos a continuación:

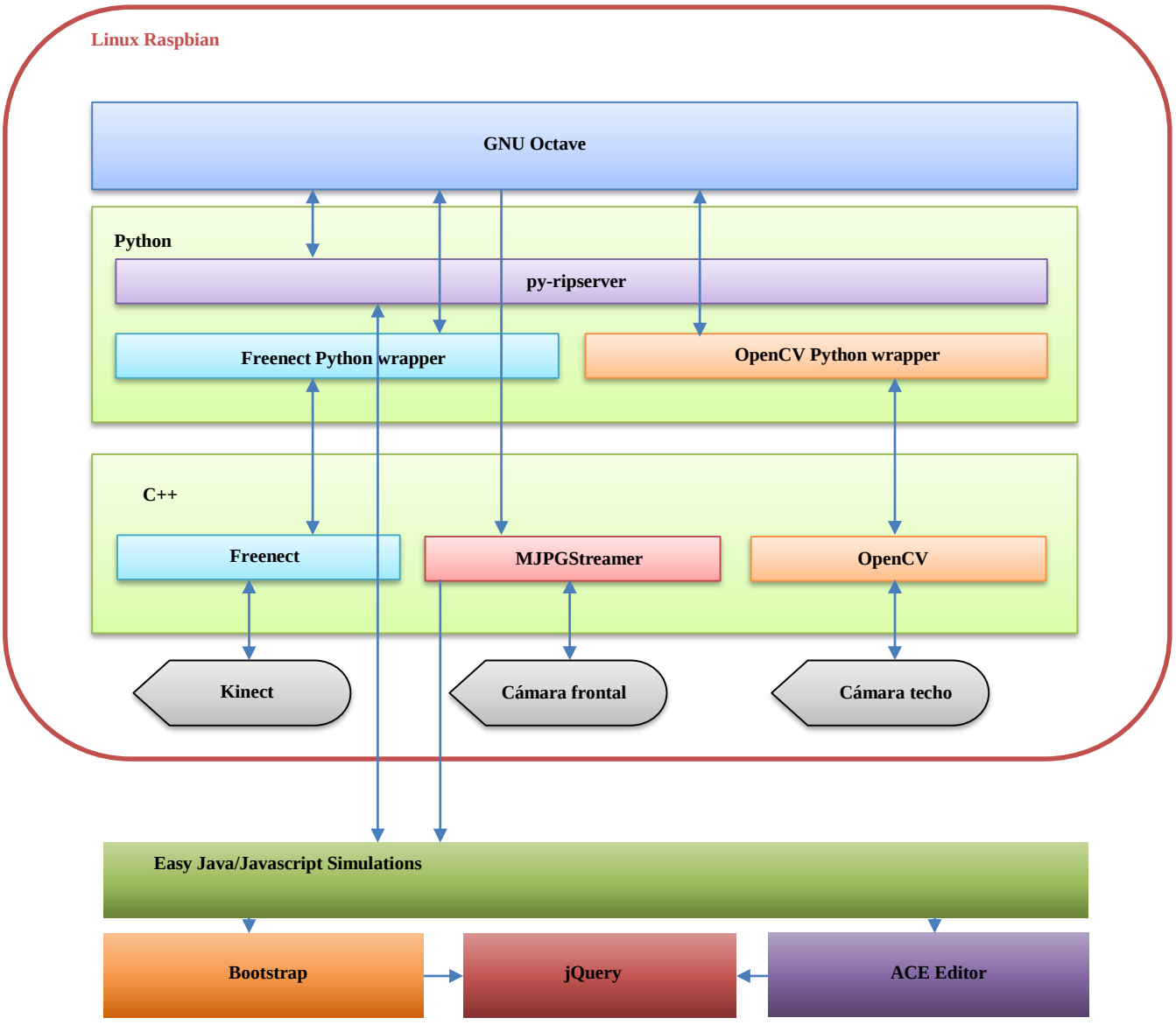


Figura 37 - Diagrama de interconexión software

2.5.1. Linux Raspbian

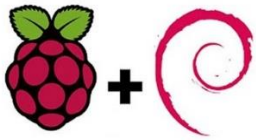


Figura 38 - Logo de Raspbian

Raspbian es una distribución Linux libre y gratuita basada en Debian preparada para ser instalada en todas las versiones de Raspberry Pi, distribuida bajo la licencia GPL, así como bajo otras licencias libres, lo cual la convierte en el sistema operativo ideal para un proyecto orientado a la docencia o a la investigación.

La versión instalada en el robot se corresponde con la versión 9.4 *Stretch* 2018-03-13, y puede ser descargada desde su página oficial [17]. Está preparada para ser instalada en una tarjeta microSD de al menos 16GB, para lo cual existen distintas herramientas, tanto para Linux como para Windows [18][19].

2.5.2. GNU Octave

Figura 39 - Logo de GNU Octave

GNU Octave es un entorno de programación de cálculo numérico basado en Matlab de tipo interpretado cuyo motor es utilizado para dar forma al núcleo de comunicación del robot.

Gracias a su facilidad de programación y a su extensibilidad, permite enlazar elementos heterogéneos como la escritura en un puerto serie, la obtención de datos a través de una cámara o el cálculo de una trayectoria.

La API proporcionada al alumno se ofrece en este lenguaje de programación, y permite ejecutar todas las posibles operaciones necesarias para operar sus actuadores y obtener información de sus sensores, así como utilizar estructuras de control de cualquier lenguaje de programación.

GNU Octave se extiende con el paquete *intrument-control* [20], necesario para lograr la comunicación con Arduino a través del puerto serie.

2.5.3. py-ripserver

Los laboratorios remotos de UNILabs hacen uso de una serie de protocolos estándar de comunicación entre el alumno y el dispositivo que se pretende controlar. *py-ripserver* [21] es un *middleware* desarrollado en Python, que ha sido migrado a la plataforma ARM64 y adaptado a las necesidades de la experiencia [22] que permite comunicar la interfaz gráfica desarrollada en EjsS con el robot de una forma homogénea, permitiendo el intercambio de datos y la actualización de la interfaz de forma automática.

Este servicio dispone de un entorno Python virtual, que es utilizado como intérprete por defecto para todas las tecnologías que hacen uso de este lenguaje, como OpenCV o Freenect.

2.5.4. Python 3.5

Figura 40 - Logo de Python

GNU Octave no es capaz por sí sólo de ejecutar todas las posibles tareas relacionadas con el control del robot, por lo que se han delegado algunas tareas en un lenguaje de programación de alto nivel. Más concretamente, en la versión 3.5 de Python.

La versión de Python se corresponde con la incluida en el entorno virtual desplegado con *py-ripserver*, de modo que se consigue minimizar el número de dependencias y aumentar la portabilidad de la aplicación.

Las principales tareas ejecutadas en este lenguaje de programación son las siguientes:

- Ejecución del servicio *py-ripserver* encargado de la comunicación entre EjsS y el robot.
- Servicio de captura de imágenes y matrices de profundidad en Kinect.
- Servicio de captura de imágenes y marcadores ArUco.
- Servicio de envío de matrices de resultados al usuario.

Python es uno de los lenguajes más versátiles y extensibles que existen, siendo también uno de los preferidos por la comunidad científica, para los que existen múltiples paquetes utilizados e implementados en el robot, como por ejemplo:

- **Oct2Py**: genera una sesión GNU Octave y la encapsula en un objeto, permitiendo la interacción directa entre Octave y Python.
- **NumPy**: añade soporte de vectores y matrices, así como compatibilidad nativa con el formato .mat que utiliza GNU Octave y Matlab. Por lo tanto, permite comunicar directamente Python y GNU Octave a través de ficheros, facilitando la interacción entre ambos lenguajes y permitiendo dividir las tareas a realizar por ambos elementos.
- **SciPy**: añade soporte para álgebra lineal, procesamiento de señales y procesamiento de imágenes.
- **MatPlotLib**: junto a *NumPy*, permite la generación de gráficas a partir de conjuntos de datos.
- **Freenect**: implementa un *wrapper* que permite acceder al dispositivo Kinect a través de este lenguaje.
- **OpenCV**: implementa un *wrapper* que permite acceder a la funcionalidad de la biblioteca de visión artificial OpenCV, y por lo tanto, gestionar marcadores ArUco.

2.5.5. Cython

Figura 41 - Logo de Cython

Cython [23] es un superconjunto de Python diseñado para realizar una traducción a C de lenguaje escrito en Python. Esto proporciona la ventaja principal de proporcionar la facilidad de programación de Python junto a la velocidad de proceso de C, aunando lo mejor de los dos lenguajes.

El uso principal de este lenguaje radica en la implementación de la biblioteca *Freenect*, necesaria para el acceso al dispositivo Kinect.

2.5.6. MJPGStreamer

MJPGStreamer [24] es un *middleware* capaz de trasladar frames en formato JPEG entre una fuente y un destino. Es utilizado para capturar *frames* de la cámara frontal del robot y enviarlas a un servicio web a partir del cual es posible realizar un *streaming* en vivo de lo que se encuentra frente al robot.

2.5.7. Freenect

Tal y como se expone en la sección *XBOX Kinect*, no existe un driver *oficial* del dispositivo Kinect para ser utilizado en otro hardware distinto a la videoconsola XBOX, por lo que la comunidad ha investigado la forma de poder hacer uso de las prestaciones de este elemento de visualización en sistemas operativos Windows y Linux.

La solución más extendida es a través de la biblioteca Freenect, perteneciente a su vez al proyecto OpenKinect [25].

Este driver está desarrollado en C/C++, pero se proporciona un *wrapper* compilado en *Cython* para poder utilizarse desde Python. Por lo tanto, se integra directamente en el entorno virtual del py-ripserver para permitir capturar imágenes y matrices de profundidad desde el dispositivo Kinect.

2.5.8. C++

Figura 42 - Logo de C++

Dada la cercanía de C++ [26] con el hardware, las aplicaciones desarrolladas en este lenguaje proporcionan una velocidad mucho mayor que las desarrolladas en otros lenguajes de alto nivel.

Debido a ello, C++ se utiliza para la compilación del proyecto de visión artificial *OpenCV*, del cual se generarán posteriormente *wrappers* para permitir utilizarlo desde Python, tal y como se hace con Freenect.

2.5.9. OpenCV

Figura 43 - Logo de OpenCV

OpenCV [27] es un conjunto de bibliotecas liberadas bajo licencia BSD, codificadas en C++ y desarrolladas inicialmente por Intel Corporation, que proporcionan distintas funcionalidades relacionadas con la visión por computador.

El motivo de incluir este software en el proyecto es la reciente inclusión de ArUco, un software de realidad aumentada desarrollado por la Universidad de Córdoba que proporciona la funcionalidad de detección de balizas mediante los denominados *marcadores* o *markers*.

El proyecto es de gran envergadura, por lo que su compilación e instalación es muy elevado en términos computacionales y temporales (en torno a unas ocho horas de compilación). Tal y como ocurría con Freenect, también se generan *wrappers* para Python, permitiendo a este lenguaje de programación obtener los datos necesarios y las capturas de cámara necesarias para la localización por balizas.

2.5.10. ArUco

ArUco [28] es una biblioteca liberada bajo licencia BSD utilizada para la estimación de posición y orientación de marcadores de geometría cuadrada, desarrollada en C++ por investigadores de la Universidad de Córdoba e incluida en la versión 4.0.0 de OpenCV.

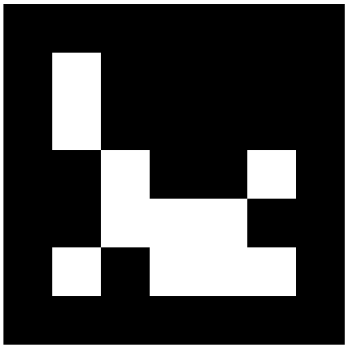


Figura 44 - Ejemplo de marcador ArUco

El funcionamiento de este software consiste en el análisis de un *frame* capturado de una cámara y realizar la detección de la presencia de estos marcadores en ella, extrayendo en caso positivo diversa información como, por ejemplo:

- Identificador único asociado al marcador.
- Orientación del marcador en los tres ejes espaciales.
- Traslación del marcador en los tres ejes respecto al centro de la cámara
- Localización de las coordenadas de las esquinas del marcador.

Estos marcadores pueden utilizarse a modo de baliza mediante su colocación en el techo de la estancia en la que se encuentra el robot, permitiendo al alumno realizar una localización del vehículo gracias a las lecturas obtenidas a partir de la cámara situada en el techo y la orientación y posición de estos marcadores.

2.5.11. EjsS

Easy Java/Javascript Simulations o EjsS es un software de modelado desarrollado por la Universidad de Murcia, utilizado para generar la interfaz gráfica del laboratorio remoto en formato XHTML, cuyo resultado será finalmente integrado en UNILabs.

Este software permite la creación de interfaces gráficas a través de un editor integrado, cuyo aspecto puede observarse en la siguiente figura:

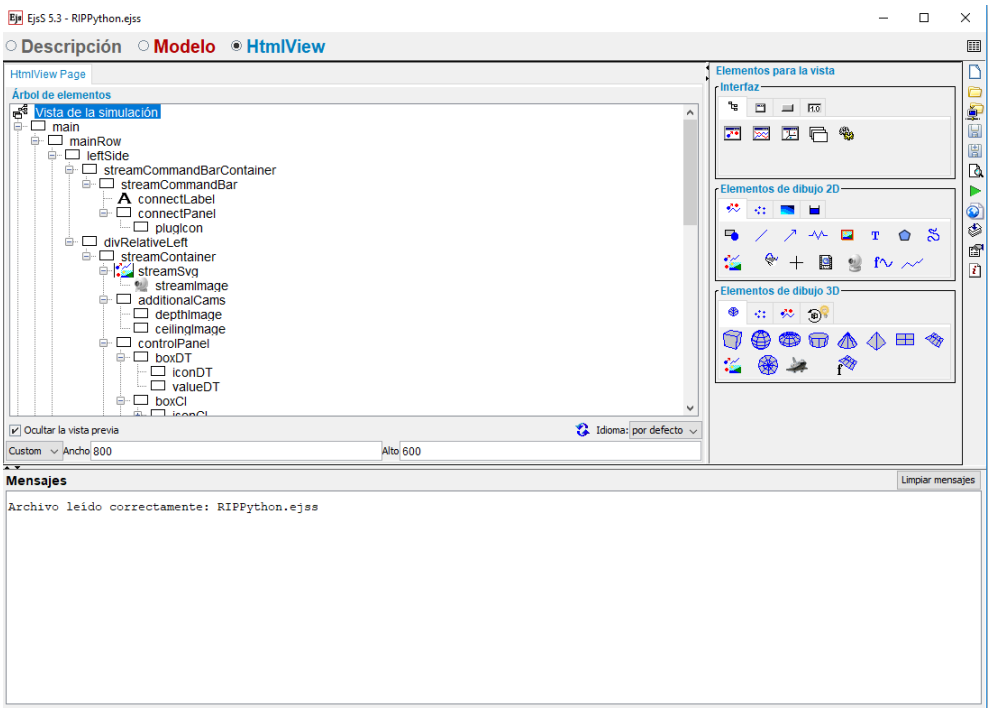


Figura 45 - Editor de interfaces de EjsS

Adicionalmente, EjsS permite la definición de código propio en formato Java o Javascript, así como elementos de modelado y conectores con elementos de terceros, como es el caso de *py-ripserver*, con el que es capaz de comunicarse de forma nativa.

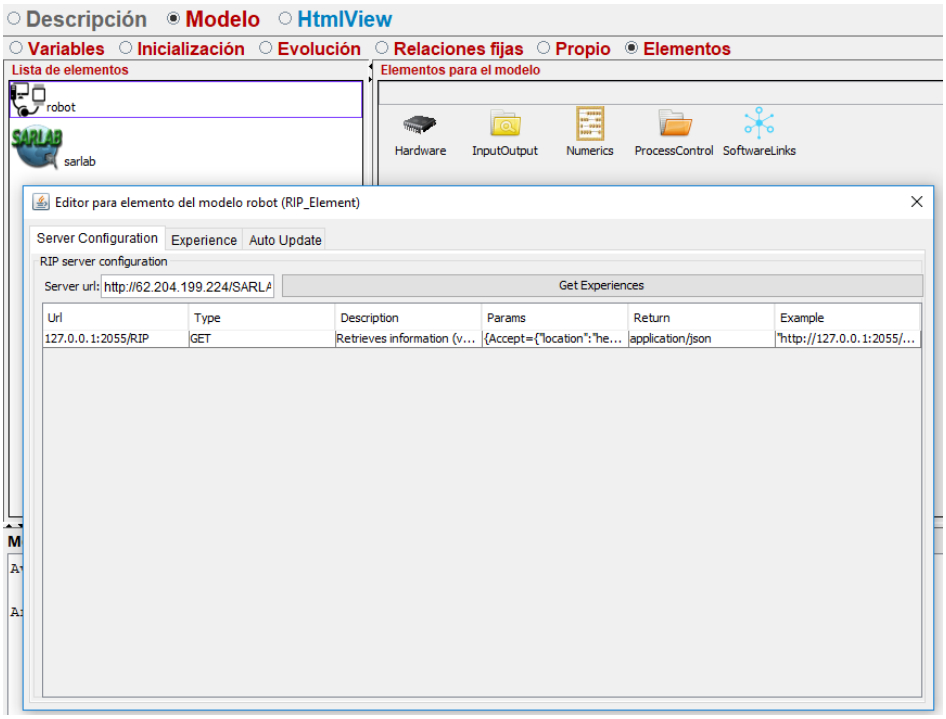


Figura 46 - Conexión con py-ripserver desde EjsS

No obstante, debido a que la herramienta de generación de interfaces es bastante limitada respecto al aspecto estético, se ha decidido intentar mejorar un poco el aspecto de la aplicación, por lo que se han añadido elementos de terceros que hagan la visualización más agradable, tales como *Bootstrap* o *Ajax.org Cloud9 Editor (ace)*.

2.5.12. Bootstrap

Figura 47 - Logo de Bootstrap

Bootstrap [11] es un framework *front-end* de código abierto que permite la inclusión de estilos, plantillas, iconos y otros componentes de interfaz de forma rápida y sencilla basándose únicamente en elementos HTML, CSS y Javascript.

Se inyecta en la interfaz EjsS a través de su CDN, por lo que permite utilizar sus estilos simplemente apuntando a un par de URLs concretas.

2.5.13. jQuery

Figura 48 - Logo de jQuery

jQuery [10] es una biblioteca de código abierto que simplifica el manejo del DOM añadiendo una API a Javascript. Esta biblioteca se incluye como dependencia de otros elementos de la interfaz, como Bootstrap o Ajax.org Cloud9 Editor.

2.5.14. Ajax.org Cloud9 Editor

El último elemento software digno de mención en cuanto a relevancia en el proyecto es el editor ACE [9] (Ajax.org Cloud9 Editor), editor de texto online de código abierto desarrollado por Ajax.org que permite la escritura de código fuente de diversos lenguajes como si un IDE de escritorio se tratara.

En el caso del robot, está integrado con EjsS para permitir la inclusión de código Octave, por lo que el código fuente se presentará y resaltará detectando los elementos del lenguaje de forma muy parecida a como lo haría el propio editor de texto de la aplicación.

2.6. Proceso de desarrollo

El proyecto comienza con el análisis de los requisitos del proyecto y con la investigación de las tecnologías involucradas. El equipo director del proyecto facilitó el acceso remoto al robot, localizado en el laboratorio de la UNED, así como diversa documentación relacionada con los proyectos anteriores en los que se basaba el proyecto actual.

Dada la dificultad derivada de obtener información directa de un dispositivo situado en un entorno remoto, se hace uso de la documentación [1] y [2] para diseñar y construir un robot *dummy* que permita simular el robot, para lo cual se adquiere material similar al descrito en [2] y se construye un prototipo en cuya placa Arduino se codifica un software que simule las entradas y salidas del robot original.

Figura 49 - Prototipo diseñado para el trabajo en local

En la Figura 49 puede observarse el prototipo utilizado para el trabajo en local, que se compone de algunos de los elementos descritos en el hardware original.

A continuación, se instala el software necesario para la codificación del robot:

- Octave
- py-ripserver
- freenect
- mjpgstreamer

2.6.1. Octave

Como primer problema, nos encontramos con que Octave no se instala correctamente mediante la orden *apt-get install octave*, ya que existe un *bug* con el JDK de Java que impide su instalación. Tras investigar el motivo, concluimos que es necesario realizar la desinstalación de los siguientes paquetes antes de efectuar la instalación:

- ca-certificates-java
- icedtea-netx-common
- libatk-wrapper-java
- libatk-wrapper-java-jni
- openjdk-8-jre-headless
- openjdk-8-jre
- oracle-java-jdk

Tras esto, se procede con la instalación de los siguientes paquetes:

- oracle-java8-jdk
- ca-certificates-java
- octave
- octave-dev

Con lo cual Octave quedaba ya perfectamente instalado.

Hecho esto, se procede a intentar conectar el hardware y acceder a los distintos elementos a través de Octave, es decir:

- Arduino
- Kinect
- Cámara frontal
- Cámara superior

La conexión a Arduino requería acceso al puerto serie, el cual se lograb mediante la instalación del paquete *instrument-control* de Octave. No obstante, nos encontramos con un nuevo problema, consistente en que la instalación del paquete mostraba un error durante la compilación.

La solución consistió en aumentar la cantidad de memoria swap disponible, creando un archivo de 1GB para realizar este proceso.

2.6.2. MJPGStreamer

Una vez conseguidas enviar órdenes a Arduino a través de Octave, se intentó realizar un streaming de la cámara frontal, lo cual fue relativamente sencillo a través de la biblioteca MJPGStreamer.

Se decide reducir la resolución a 320x240 para reducir el ancho de banda disponible.

2.6.3. Kinect, OpenKinect y Freenect

El tercer elemento a configurar era el dispositivo Kinect. Basándonos en el trabajo descrito en [2], hacemos uso de la biblioteca *Psychtoolbox* de Octave, para lo cual debemos descargar el paquete *libfreenect* para acceder al dispositivo.

No obstante, tras una primera implementación inicial, se observa que la conectividad con Kinect es muy inestable, produciéndose errores y desconexiones de forma constante, por lo que es necesario plantear otra vía de obtener las matrices de profundidad.

Finalmente, se opta por hacer uso del *wrapper* de *freenect* para Python y la creación de un esquema cliente-servidor en el que se codifica un servidor TCP (KinectDepthImageServer) que se mantiene a la espera de peticiones por parte de un cliente (kinectclient).

El *wrapper* de Python, pese a no ser perfecto y fallar ocasionalmente, cumple correctamente su función. Dado que la biblioteca *NumPy* de Python permite la lectura y escritura de ficheros *.mat*, se decide codificar la adquisición de imágenes y matrices de profundidad del dispositivo Kinect mediante el siguiente proceso:

1. Octave invoca el script Python "kinectclient" indicándole que almacene una matriz de profundidad, una imagen o ambos elementos en una ruta conocida tanto por KinectDepthImageServer como por Octave.
2. El servidor crea un fichero de bloqueo para indicar que está trabajando y que la matriz y/o imagen no pueden ser aún leídas por el cliente.
3. El servidor solicita a Kinect la imagen y/o matriz de profundidad, almacenándola en la ruta predefinida.
4. Una vez finalizado el proceso, el servidor elimina el fichero de bloqueo, indicando así que los archivos están listos para su consumo.
5. Octave espera a que el archivo de bloqueo no exista para realizar la lectura de los archivos solicitados.
6. En caso de que los archivos solicitados existan, comprueba que la fecha de modificación de los archivos es posterior a la fecha en la que se solicitaron.
7. Si el archivo o archivos solicitados cumplen los requisitos anteriores, Octave carga la matriz resultante y devuelve su contenido a la línea principal del programa, teniendo ya acceso a la información de profundidad.

Otro de los problemas a los que tuvimos que enfrentarnos fue la necesidad de *traducir* las imágenes de profundidad a distancias reales, para lo cual hizo falta investigar cómo hacerlo de forma sencilla y computacionalmente aceptable, obteniendo una respuesta satisfactoria en [5] y en [6], donde se usan los parámetros intrínsecos de la cámara para realizar el cálculo.

De este modo, se proporciona al usuario la matriz de datos en bruto para que tenga la oportunidad de diseñar su propio método, pero también se proporciona un método que realiza el cálculo traduciendo los valores a centímetros, pudiendo visualizar por pantalla el resultado final.

2.6.4. EjsS y RIPServer

Teniendo ya parte de la funcionalidad implementada, se comienza el desarrollo de la interfaz EjsS manteniendo comunicación constante con la dirección del proyecto, surgiendo varios problemas que fueron rápidamente solventados, como, por ejemplo:

- Incompatibilidad del entorno virtual
- Colisiones de estilos CSS entre el proyecto EjsS y UNILabs.
- Problemas de comunicación entre EjsS y py-ripserver.
- Incapacidad de conexión al servidor remoto desde direcciones externas al propio servidor.

Como hemos indicado, la comunicación con la dirección del proyecto facilitó y agilizó mucho la resolución de estos problemas, haciendo que la integración de cliente y servidor se realizara satisfactoriamente, tal y como se muestra en la siguiente figura:

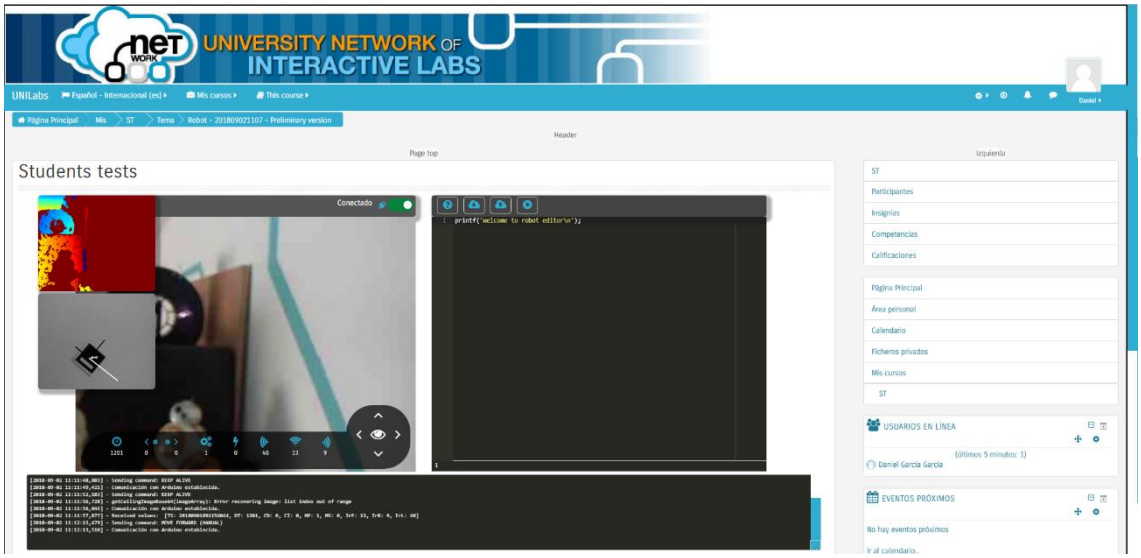


Figura 50 - Integración de EjsS en UNILabs con py-ripserver

Tras actualizar debidamente tanto el servidor py-ripserver como los plugins de EjsS, la conexión al servidor comenzó a funcionar tal y como se esperaba a través del proxy del laboratorio.

2.6.5. Arduino

El desarrollo se fue alternando entre el prototipo local y el robot remoto, hasta que se tuvo finalmente acceso físico al robot de producción, en el que se fueron refinando sucesivamente los métodos de conexión a Arduino.

Dado que el programa de aplicación embebido en la placa Arduino está planteado como un modelo petición-respuesta, los métodos de actuación consisten principalmente en el envío de una orden de escritura seguida de una orden de lectura. Esto es: el robot no genera lecturas por sí mismo, sino que devuelve las lecturas de sus sensores al recibir una orden de escritura. Estos valores se almacenan en un formato *parecido* a JSON, pero sin serlo, por lo que se procedió a la homogeneización de los formatos de los mensajes a JSON válido antes de ser enviados a los métodos que se encargaban de procesar la información. Además, debido al envío casi simultáneo de la orden de escritura y la de lectura, durante las pruebas con el robot final se detectó que existían errores importantes a la hora de enviar solicitudes al robot, en las que el puerto serie se veía saturado y producía comportamientos erráticos. Por ello se optó por añadir la posibilidad de realizar la lectura tras un retardo calculado en

función de la señal enviada, de realizar la lectura tras un tiempo proporcionado por el usuario o directamente por no realizar la lectura.

Finalmente, el comportamiento del robot comenzó a ser el esperado, en el que los envíos de las señales provocaban la acción codificada.

2.6.6. Persistencia

El siguiente reto al que nos enfrentamos fue la necesidad de evitar que un corte de luz o de la conexión hiciese que se perdiera la información recabada por el robot durante la experiencia. Sumando este reto a que se pueden llegar a manejar volúmenes elevados de datos (las matrices de profundidad son costosas en cuanto a la cantidad que ocupan en memoria), se optó por la siguiente solución:

- Cada dato obtenido por un medio sensorial solicitado por el código del usuario es *cacheado* en una variable distinta cuya nomenclatura sigue el siguiente formato:
    - [tipo-variable][Marca de tiempo en formato yyyymmddHHMMSSFFF]
- Estas variables se identifican por una letra inicial que precede al *timestamp* que da nombre a la variable, que son las siguientes:
    - aTIMESTAMP: lecturas de Arduino.
    - kTIMESTAMP: lecturas de Kinect.
    - xTIMESTAMP: localizaciones de las esquinas de los marcadores ArUco
    - rTIMESTAMP: vectores de rotación de los marcadores ArUco.
    - tTIMESTAMP: vectores de traslación de los marcadores ArUco.
- Una vez que se ha creado la variable, se añade a un fichero *.mat* en la caché del robot (/var/robot/cache), cuyo nombre vendrá dado por el siguiente formato:
    - cache_[ID deUNILabs]_[Marca de tiempo en formato yymmddHHMMSSFFF]
- Por lo tanto, se creará un nuevo fichero por cada usuario y marca de tiempo, haciendo que los ficheros de caché no se sobrescriban al crear una nueva sesión y permitiendo, en caso necesario, que el equipo docente pueda recuperar los datos obtenidos durante una sesión determinada.
    - Los datos de caché se borran automáticamente transcurridos siete días desde su creación para evitar agotar el espacio de almacenamiento del robot.

- Una vez que el usuario finaliza la sesión (voluntaria o involuntariamente), éste puede pulsar un botón de la interfaz que transformará el último fichero de caché con su ID de usuario en un fichero .mat que contendrá todos los valores anteriores aglutinados en una única matriz por cada tipo de dato. Más concretamente:
    - **robotData**: <nx9 double>: almacena las lecturas del robot. En esta matriz se insertarán aquellas variables con el formato ayyyymmddHHMMSSFFF. El significado de cada posición es la siguiente:
        - 1: TS - Marca de tiempo en formato yyyymmddHHMMSSFFF
        - 2: DT – Tiempo desde la última lectura
        - 3: CD – Número de cuentas del encoder derecho
        - 4: CI – Número de cuentas del encoder izquierdo
        - 5: MP – Estado del motor (1=encendido, 0=apagado)
        - 6: MS – Estado del láser (1=encendido, 0=apagado)
        - 7: IrF – Distancia en cm detectada por el sensor infrarrojo frontal
        - 8: IrR - Distancia en cm detectada por el sensor infrarrojo derecho
        - 9: IrL - Distancia en cm detectada por el sensor infrarrojo izquierdo
    - **robotDepthImages**: <nx480x640 double>: almacena las matrices de profundidad capturadas por Kinect.
    - **robotDepthImageTimestamps**: <nx1 double>: almacena las marcas de tiempo de las capturas de profundidad, que se correlacionan por el índice. Esto es, a la captura robotDepthImages(5,:,:) le corresponderá la marca de tiempo robotDepthImageTimestamps(5).
    - **cornerData**: <nx10 double>: almacena la información de las esquinas de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición y el resto de campos con el valor NaN. Los índices tienen el siguiente significado:
        - 1: Timestamp de la captura en formato yyyymmddHHMMSSFFF.
        - 2: Identificador del marcador ArUco.
        - 3,4: Coordenadas (x,y) de la esquina inferior izquierda del marcador.
        - 5,6: Coordenadas (x,y) de la esquina inferior derecha del marcador.
        - 7,8: Coordenadas (x,y) de la esquina superior derecha del marcador.
        - 9,10: Coordenadas (x,y) de la esquina superior izquierda del marcador.

- **rotationData**: <nx8 double>: cada fila contiene ocho elementos con la información referente a la rotación de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición y el resto de campos con el valor NaN. Los índices tienen el siguiente significado:
  - 1: Timestamp de la captura.
  - 2: Identificador del marcador ArUco.
  - 3-5: Vector de rotación en representación eje-ángulo.
  - 6: Ángulo de alabeo expresado en radianes.
  - 7: Ángulo de cabeceo expresado en radianes.
  - 8: Ángulo de guiñada expresado en radianes.
- **translationData**: <nx5>: cada fila contiene cinco elementos con la información referente a la traslación de los marcadores ArUco. En caso de que durante una solicitud de captura ArUco no se detecten marcadores, se creará un registro con la marca de tiempo de la petición y el resto de campos con el valor NaN. Los índices tienen el siguiente significado:
  - 1: Timestamp de la captura.
  - 2: Identificador del marcador ArUco.
  - 3: Coordenada X del vector de traslación.
  - 4: Coordenada Y del vector de traslación.
  - 5: Coordenada Z del vector de traslación.

- La información obtenida por ArUco puede correlacionarse mediante (timestamp, id), ya que una misma captura compartirá estos dos valores en las tres matrices proporcionadas por ArUco (cornerData, rotationData y translationData).
- Un servicio que escucha en el puerto 8082 se encarga de recibir peticiones del siguiente tipo:
  - http://ip-del-servidor:8082/ID_de_UNILabs
- Al recibir la petición, en servidor realizará la obtención del último fichero de caché, realizará la transformación al nuevo formato y será entregado a través de HTTP para su descarga por parte del estudiante.

Si el estudiante ya ha ejecutado alguna sesión previa y accede a la experiencia, podrá descargarse el fichero de la última sesión siempre y cuando no comience una sesión nueva. Para ello, deberá

pulsar un botón habilitado para ello en la esquina superior derecha del editor de texto, que no estará disponible durante el transcurso de la ejecución de la experiencia.

NOTA: el control manual NO genera datos de caché.

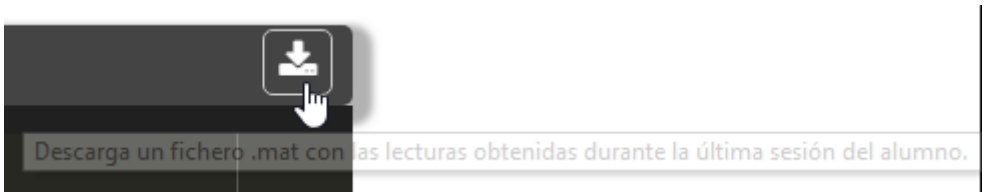


Figura 51 - Botón de descarga con los datos recabados por el alumno.

2.6.7. Robustez

Otro de los requisitos necesarios para el correcto funcionamiento del robot es que éste fuera robusto y que no provocase estados de bloqueo, por lo que se optó por dos decisiones de diseño que cubriesen este aspecto:

La primera de ellas consistió en establecer un *timeout* máximo en el que el alumno pudiese ejecutar una experiencia de usuario. Este tiempo es de tres minutos por defecto, por lo que, si un código escrito por el usuario tarda más de este tiempo en finalizar, la ejecución será abortada. Esto cubre posibles problemas como la entrada en bucles infinitos.

No obstante, esta decisión trae consigo la problemática de que el robot tarde más de tres minutos en ejecutar el programa, lo cual es bastante probable, ya que los procesos de obtención de imágenes de profundidad y de imágenes del techo no son inmediatas y ralentizan la adquisición de datos.

Por ello, se proporciona al alumno un método especial, *setExecutionTimeout*, que informa al RIPserver sobre el deseo del alumno de ampliar esta ventana de tiempo hasta un máximo de quince minutos (900 segundos).

Para que la operación funcione, el código deberá contener este código (y sólo este) en la primera línea del programa, pasando el resto del código a Octave. De este modo, si el usuario se ha asegurado de que el programa no incurre en bucles no deseados y necesita ampliar este tiempo, podrá hacerlo bajo su responsabilidad.

Otra de las posibles problemáticas radicaba en la posibilidad de que el servicio *RIPServer* o alguno de sus procesos asociados dejaran de responder. Para ello se ha implementado un *watchdog* en el *cron*

del sistema operativo que interroga al proceso cada cinco minutos, reiniciando todos los servicios en caso de que no responda. Adicionalmente, se controla mediante una confirmación que el usuario no abandone o recargue la página accidentalmente.

2.6.8. Feedback

Como último requisito de usabilidad se estableció la necesidad de que el usuario recibiese *feedback* por parte del robot, para lo cual se implementó una caja de texto que actúa a modo de *log* del sistema, mostrando por defecto los mensajes de importancia (*warnings* y *errors*), el envío de operaciones manuales y la recepción de datos por parte del robot.

En caso de que el usuario decida incluir sentencias *fprintf* en su código, el resultado de éstas se mostrará también por la consola, permitiendo obtener la información deseada del entorno del robot.

Adicionalmente es posible modificar el nivel de detalle de los mensajes de log mediante la orden *setDebugLevel*, estableciendo un valor entre 1 (menos detalle) y 5 (más detalle). No obstante, este método está pensado para labores de depuración y desarrollo, por lo que no se recomienda utilizarlo salvo que exista una buena razón para ello.

2.6.9. OpenCV y ArUco

El último eslabón del desarrollo se basaba en la idea aportada por Juan Ignacio Forcén en [1], consistente en añadir un sistema de localización basada en balizas mediante marcadores ArUco. Desde el momento en el que Juan Ignacio desarrolló su trabajo hasta el día de hoy, ArUco ha dejado de ser un trabajo de investigación basado en OpenCV por el personal investigador de la Universidad de Córdoba para pasar a formar parte de pleno derecho de OpenCV, por lo que ha bastado con instalar las bibliotecas para poder tener acceso a esta funcionalidad.

La instalación, no obstante, no ha sido sencilla, ya que ha sido necesario descargar el código fuente, compilarlo y generar a continuación los correspondientes *wrappers* en Python (Juan Ignacio hacía uso directamente de C++, lenguaje en el que está codificado OpenCV y ArUco, para esta tarea).

El proceso de compilación lleva en torno a unas ocho horas, ya que no existen binarios para instalar directamente en una distribución Raspbian ARM64. Además, los artefactos intermedios ocupan cerca de 5GB en disco, por lo que fue necesario habilitar espacio mediante la desinstalación del *wolfram-engine* que por defecto viene instalado en la distribución Raspbian.

Tras la instalación fue necesario generar los *wrappers* de Python para poder invocar la funcionalidad de ArUco desde este entorno, así como instalarlos en el entorno virtual del servidor RIP, desde el cual se ejecuta siempre Python.

Una vez que ArUco estaba instalado, fue necesario realizar una calibración de la cámara para obtener ciertos parámetros necesarios para la captura de la información de los marcadores, para lo cual fue necesario diseñar e imprimir un CharUco (Charboard + ArUco), colocarlo en una superficie rígida y realizar el proceso de obtención de los parámetros intrínsecos de la cámara de los parámetros de distorsión, que serán utilizados durante el proceso de obtención de los marcadores[34][35][36][37][38].

La Figura 52 muestra el resultado de la generación del *ChArUco* utilizado para obtener los datos intrínsecos de la cámara, correspondiente a los siguientes valores:

$$cameraMatrix = \begin{pmatrix} 127{,}07960793226029 & 0 & 187{,}9832593519401 \\ 0 & 192{,}65367457747448 & 453{,}3474663142604 \\ 0 & 1 & 0 \end{pmatrix}$$

$$distortionCoefficients = (0{,}08303926284668485 \quad 0{,}005748723261263953 \quad 0{,}11692000998930098 \quad 0{,}002869130570971433 \quad -0{,}0006282266980082555)$$

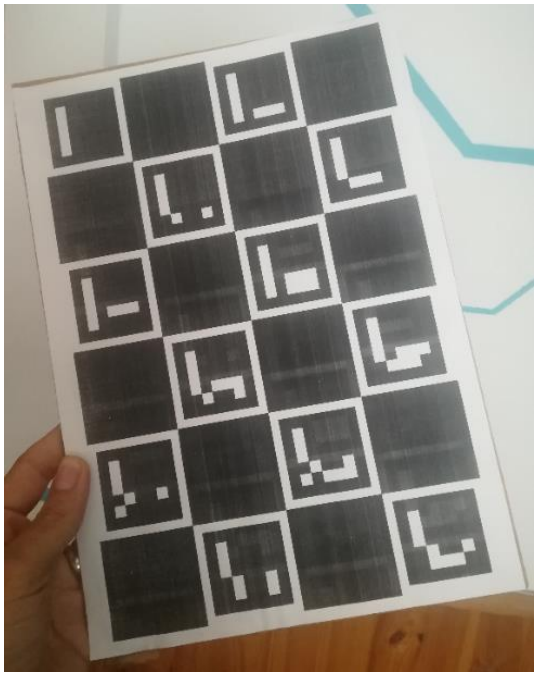


Figura 52 - Impresión del ChArUco de calibración de la cámara

Una vez calibrada la cámara, se generó un servicio siguiendo un esquema cliente-servidor similar al descrito en la captura de imágenes de profundidad Kinect. Este servicio sigue el siguiente esquema de funcionamiento:

1. Octave invoca el script Python "arucoclient" indicándole que almacene una colección de matrices con la información de los marcadores, una imagen o ambos elementos en una ruta conocida tanto por ArucoServer como por Octave.
2. El servidor crea un fichero de bloqueo para indicar que está trabajando y que la matriz y/o imagen no pueden ser aún leídas por el cliente.
3. El servidor solicita a la cámara la imagen y/o colección de matrices de los marcadores, almacenándola en la ruta predefinida.
4. Una vez finalizado el proceso, el servidor elimina el fichero de bloqueo, indicando así que los archivos están listos para su consumo.
5. Octave espera a que el archivo de bloqueo no exista para realizar la lectura de los archivos solicitados.
6. En caso de que los archivos solicitados existan, comprueba que la fecha de modificación de los archivos es posterior a la fecha en la que se solicitaron.
7. Si el archivo o archivos solicitados cumplen los requisitos anteriores, Octave carga las matrices resultante y devuelve su contenido a la línea principal del programa, teniendo ya acceso a la información de los datos de los marcadores.

2.7. Localización

El robot, tal y como está implementado, proporciona dos elementos principales para definir su localización: localización relativa a partir de los datos odométricos obtenidos a través de los encoders y localización absoluta a partir de los marcadores *ArUco* situados en el techo.

2.7.1. Modelo diferencial

Dado que el robot implementa un sistema cinemático diferencial, éste girará mediante la aplicación de un valor de corriente distinto a las ruedas de cada lado. Un valor similar en ambos motores hará

que la posición del robot se traslade en un movimiento rectilíneo sin modificar su orientación, pero un valor distinto hará que una rueda recorra más distancia que la otra, llevando inevitablemente a la ejecución de un cambio de dirección.

El cálculo de la posición mediante el modelo diferencial se realizará a partir de la siguiente información:

- Cuentas de cada encoder recorridas por cada rueda.
- Número total de cuentas en cada rueda.
- Radio de la rueda
- Distancia entre el punto de apoyo de ambas ruedas.

Conociendo estos datos, será posible realizar un cálculo de la posición a partir de las cuentas registradas por cada encoder.

Figura 53 - Relación entre el radio de la rueda y el arco recorrido en una vuelta

Sabiendo que cada rueda registra 6000 cuentas por revolución, con estos datos ya podremos calcular la distancia recorrida por cada una de las ruedas al detectar un número *n* de cuentas:

$$d = \frac{2\pi rc}{N} \qquad (6)$$

Siendo:

- d: distancia recorrida por la rueda
- r: radio de la rueda
- c: cuentas leídas por el encoder
- N: número total de cuentas del encoder.

Además de un movimiento rectilíneo, el robot puede realizar dos tipos de movimientos adicionales:

Giro sobre su propio eje: no realiza desplazamiento espacial, pero sí rotacional, por lo que únicamente cambiaría su ángulo de giro.

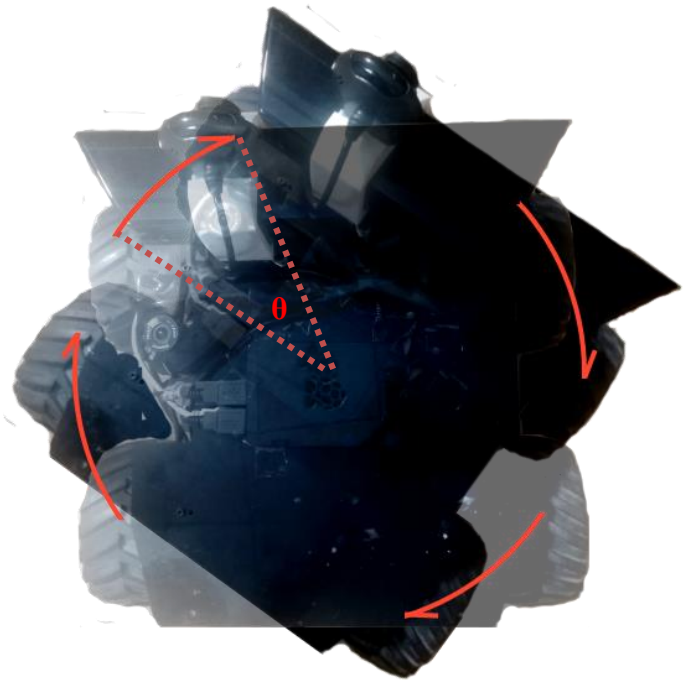


Figura 54 - Giro del robot sobre su propio eje

Cuando esto ocurre, es posible obtener la variación del ángulo a partir de la fórmula (7) que describe el cálculo de la longitud de un arco de circunferencia:

$$l_{arco} = \frac{\theta l_{circunferencia}}{2\pi} = \frac{\theta 2\pi r}{2\pi} = \theta r \qquad (7)$$

Dado que el radio es conocido y que es posible obtener l_{arco} a través de la ecuación (6), tendremos que el ángulo girado será:

$$\theta = \frac{l_{arco}}{r} \qquad (8)$$

Giro durante el avance: realiza desplazamiento rotacional y espacial. Se trata de una combinación de los dos casos anteriores, en los que el centro de giro del robot no coincide con el centro del mismo, sino que se traslada a un punto imaginario, en el que cada rueda traza una circunferencia de distinto radio, pero con centro común, tal y como puede observarse en la figura siguiente:

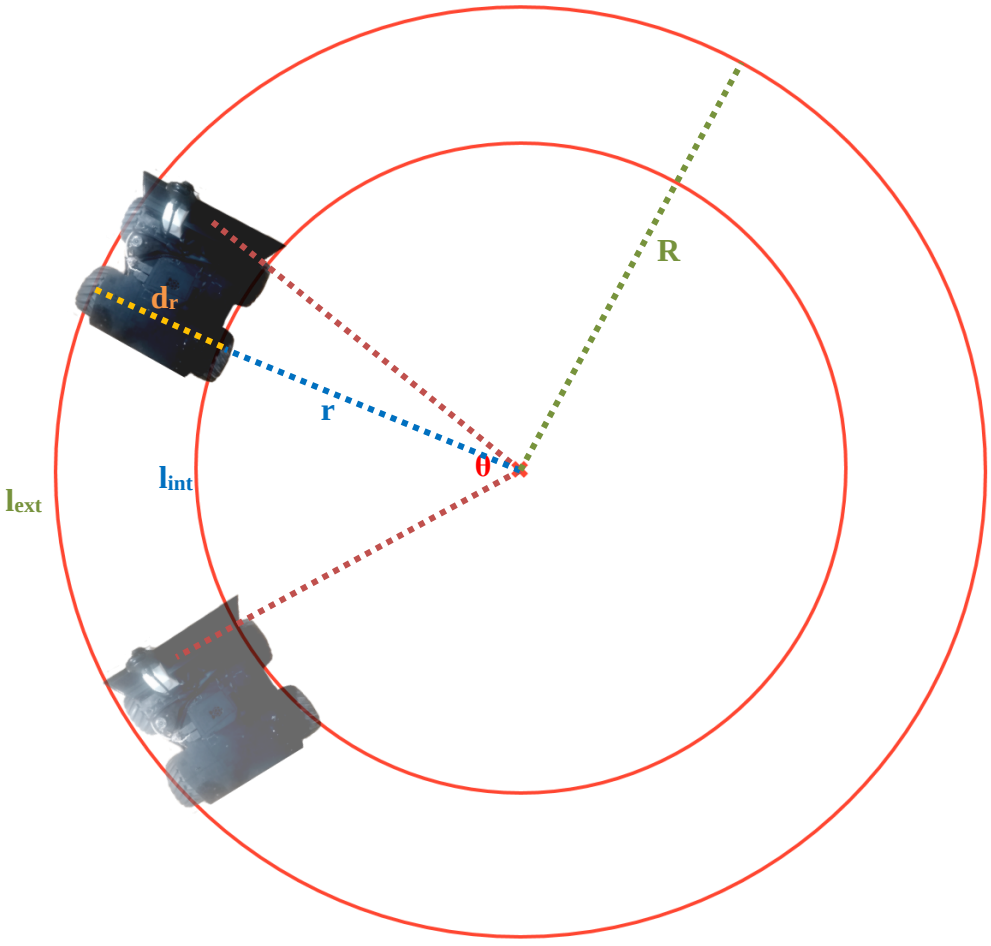


Figura 55 - Movimiento de rotación y traslación simultaneo

Siendo:

- d_r: distancia entre las ruedas (en el caso de este robot, 28 cm).
- l_{ext}: longitud del arco de circunferencia recorrido por la rueda exterior, correspondiente al valor obtenido añadiendo a (6) el número de cuentas leídas por el encoder.
- l_{int}: longitud del arco de circunferencia recorrido por la rueda interior, correspondiente al valor obtenido añadiendo a (6) el número de cuentas leídas por el encoder.

El ángulo descrito por ambas ruedas es el mismo, pero las ruedas situadas en el exterior del giro recorren más distancia que las ruedas en el interior, por lo que $l_{ext} > l_{int}$.

Sabiendo que la longitud del arco es igual al ángulo recorrido por el radio y que el ángulo recorrido en ambas circunferencias es el mismo, puede deducirse lo siguiente:

$$\theta_{giro} = \frac{l_{int}}{r} = \frac{l_{ext}}{R} \qquad (9)$$

De donde se obtiene:

$$\frac{l_{int}}{r} = \frac{l_{ext}}{r + d_r}; \quad r + d_r = \frac{r l_{ext}}{l_{int}} \; ; \quad l_{int} r + l_{int} d_r = r l_{ext} \qquad (10)$$

Despejando r, obtendremos el valor del radio de giro de la rueda interior:

$$r = \frac{d_r l_{int}}{l_{ext} - l_{int}} \qquad (11)$$

Una vez conocido el radio interior, podemos conocer el radio exterior sumándole la distancia entre ambas ruedas:

$$R = r + d_r \qquad (12)$$

Finalmente, la posición del robot vendrá determinada por el centro de ambos radios, es decir, por la suma de la mitad de la distancia entre ambas ruedas:

$$r_{robot} = r + \frac{d_r}{2} \qquad (13)$$

Por último, sustituyendo (11) en (9) obtendremos finalmente el ángulo de giro:

$$\theta_{giro} = \frac{l_{int}}{r} = \frac{l_{int}}{\frac{d_r l_{int}}{l_{ext} - l_{int}}} \qquad (14)$$

Conocido el ángulo y la posición anterior dada por las coordenadas (x_0, y_0), y su orientación dada por θ, es posible calcular la nueva posición, para lo cual deberemos en primer lugar calcular el ángulo que el eje de las ruedas forma con el origen de la circunferencia y la posición (x, y) de dicho origen.

Si el ángulo de giro es positivo (sentido antihorario), el ángulo que forma el eje de las ruedas será el resultado de restarle 90º (π/2 rad) a la orientación actual del robot, sumándosela en caso de que el giro sea negativo.

$$\theta_{eje_0} = \begin{cases} \theta_0 - \frac{\pi}{2}, & si\ c_d > c_i \\ \theta_0 + \frac{\pi}{2} & si\ c_i \leq c_d \end{cases} \qquad (15)$$

Siendo:

- c_d: cuentas leídas por el encoder derecho
- c_i: cuentas leídas por el encoder izquierdo

El nuevo ángulo del eje de las ruedas sería, por lo tanto:

$$\theta_{eje} = \begin{cases} \theta_{eje_0} + \theta_{giro}, & si\ c_d > c_i \\ \theta_{eje_0} - \theta_{giro}, & si\ c_i \leq c_d \end{cases} \qquad (16)$$

Mientras que la nueva orientación vendría dada por (17):

$$\theta = \begin{cases} \theta_0 + \theta_{giro}, & si\ c_d > c_i \\ \theta_0 - \theta_{giro}, & si\ c_i \leq c_d \end{cases} \qquad (17)$$

Conocido el ángulo inicial del eje de las ruedas θ_{eje0} y la distancia del centro de los ejes al centro de la circunferencia imaginaria, calculamos mediante trigonometría las coordenadas (x, y) de dicha circunferencia:

$$x_{circunferencia} = r_{robot} \cos(\theta_{eje_0}) \quad (18)$$

$$y_{circunferencia} = r_{robot} \operatorname{sen}(\theta_{eje_0}) \quad (19)$$

Con esta información, calculamos la distancia recorrida en los ejes X e Y:

$$d_x = r_{robot} \cos(\theta_{eje}) \quad (20)$$

$$d_y = r_{robot} \operatorname{sen}(\theta_{eje}) \quad (21)$$

Finalmente, calculamos la posición (x, y), que será el resultado de sumarle la distancia recorrida en cada eje a la posición actual restándole la distancia al centro del círculo:

$$x = x_0 + d_x - x_{circunferencia} \quad (22)$$

$$y = y_0 + d_y - y_{circunferencia} \quad (23)$$

Con estos datos, habríamos obtenido la información necesaria para calcular la posición y orientación del robot a partir de las cuentas de los encoders, el radio de las ruedas y la distancia entre ejes.

Este algoritmo se implementa en el método de Octave *calculateMovement*, perteneciente a la API del robot y puesto a disposición del alumno. No obstante, si se desea que el alumno desarrolle su propio algoritmo, hay múltiple literatura que puede ser utilizada para tal fin [48].

2.7.2. Localización absoluta mediante balizas

Junto a los encoders, la cámara que el robot tiene apuntando al techo puede proporcionar información relativa a los marcadures ArUco detectados sobre él. Cada marcador detectado proporcionará los siguientes datos:

- Un identificador (único por cada marcador)
- Un array con las posiciones (x, y) de sus esquinas, comenzando por la esquina inferior izquierda y finalizando por la esquina superior izquierda, recorriendo el marcador en sentido antihorario (positivo). El contenido del array será, por lo tanto:
    - [TIMESTAMP, ID, x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3]

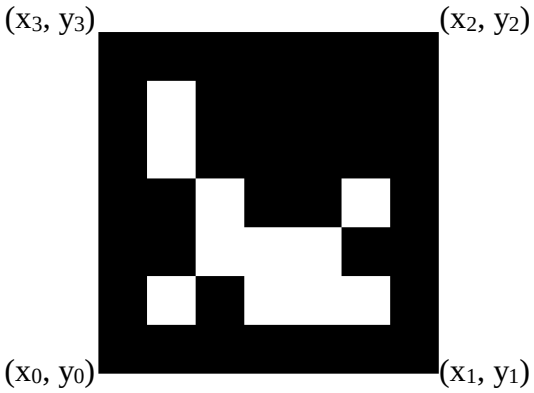


Figura 56 - Localización de las esquinas del marcador en el array de esquinas de ArUco.

- Un array con la información de rotación del marcador en dos formatos: eje-ángulo (tres primeros valores) y alabeo, cabeceo y guiñada, es decir:
    - [TIMESTAMP, ID, v_x, v_y, v_z, θ_{alabeo}, $\theta_{cabeceo}$, $\theta_{guiñada}$]
- Un array con el vector (x,y,z) de traslación:
    - [TIMESTAMP, ID, d_x, d_y, d_z]

ArUco proporciona la información de rotación en forma de vector eje-ángulo [49], consistente en representar una rotación a través de un vector unitario y un ángulo de revolución sobre dicho vector.

Cualquier rotación en tres dimensiones puede representarse de esta manera, pasando de una orientación a otra a través de un eje y un ángulo, tomando como positiva la dirección que cumpla la regla de la mano derecha.

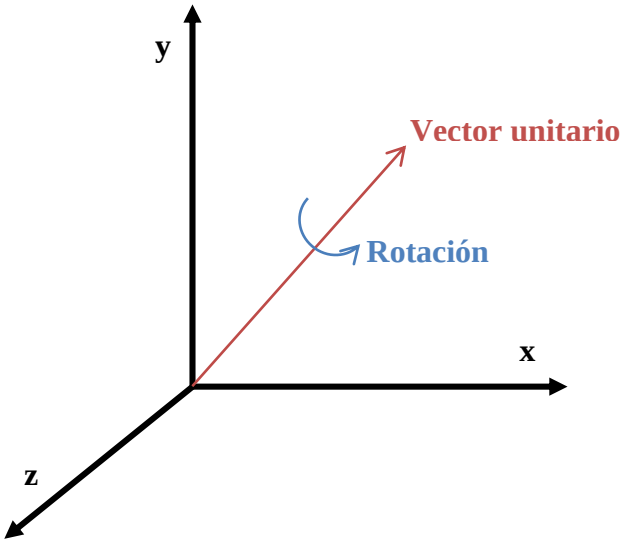


Figura 57 - Representación en eje-ángulo

Dado que obtener ángulos *puros* a partir de esta representación requiere cálculo matricial, se realiza el cálculo del alabeo, cabeceo y guiñada calculado a partir de los tres vectores anteriores, ya que debido a que el giro se producirá únicamente en uno de los ejes (no se plantea que el robot realice cambios de altitud, y se entiende que los marcadores ArUco estarán paralelos al plano del robot), únicamente necesitaremos el ángulo de guiñada, que nos proporcionará directamente el ángulo de rotación del marcador.

Los conceptos de alabeo, cabeceo y guiñada (*roll*, *pitch*, *yaw* en inglés) provienen de la aeronáutica, y establecen la inclinación de una aeronave respecto a los tres ejes, tal y como se muestra en la siguiente figura:

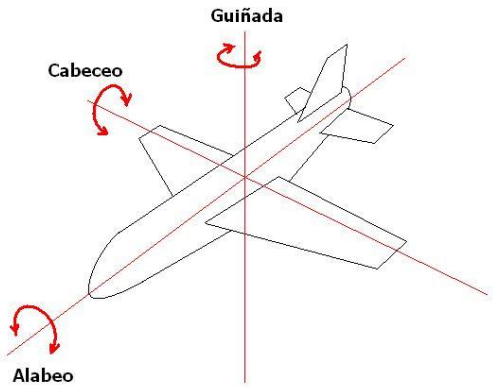


Figura 58 - Representación en alabeo, cabeceo, guiñada

2.8. Tratamiento de la información

2.8.1. Formato de los datos

Tras la finalización de la sesión se generará un fichero que el usuario podrá descargar, conteniendo los datos capturados en las siguientes matrices:

- robotData: información recabada desde Arduino referente a cuentas y sensores.
- robotDepthImages: imágenes de profundidad obtenidas desde el dispositivo Kinect.
- robotDepthImageTimestamps: marcas de tiempo de las imágenes almacenadas en robotDepthImages, coincidentes en su índice.
- cornerData: información de los marcadores ArUco correspondientes a las esquinas de los marcadores.
- rotationData: información de los marcadores ArUco correspondientes a los vectores de rotación eje-ángulo, más sus ángulos de alabeo, cabeceo y guiñada.
- translationData: información de los marcadores ArUco correspondientes a los vectores de desplazamiento.

Todas estas matrices tienen como primer campo un *timestamp* con el instante en el que se realiza la captura, en formato yyyymmddHHMMSSFFF, de modo que es sencillo conocer a qué instante pertenece cada dato, permitiendo su análisis posterior.

Para más información sobre el formato de los datos, su significado y cómo obtenerlos, se insta al lector a consultar la sección *Persistencia*.

2.8.2. Construcción de mapas

Un mapa se construye por una combinación de datos odométricos y sensoriales, de modo que pueda trasladarse la información referente a los obstáculos detectados a una posición determinada en el espacio.

Dado que el robot proporciona varios métodos de detección de obstáculos y de localización, será el usuario el que decida qué combinación utilizar.

A continuación, se ejemplificará cómo realizar un mapa tridimensional mediante localización mediante odometría calculada a partir del modelo diferencial y los datos de profundidad obtenidos a partir del dispositivo Kinect, aunque estos datos podrían complementarse con la información de los sensores de infrarrojos y la detección de los marcadores ArUco.

Se aconseja que los movimientos del robot no sean muy bruscos a la hora de realizar el muestreo, ya que la capacidad de proceso de Raspberry Pi es limitada y la velocidad de obtención de los datos no se caracteriza por ser especialmente rápida.

Las matrices de profundidad se almacenan en una matriz <nx480x640>, por lo que lo primero que deberíamos hacer es extraer una a una cada una de estas matrices y redimensionarlas para convertirlas en matrices <640x480>.

```
load('robot.mat');

m1 = reshape(robotDepthImages(1,:,:), 480, 640);
m2 = reshape(robotDepthImages(2,:,:), 480, 640);
m3 = reshape(robotDepthImages(3,:,:), 480, 640);
m4 = reshape(robotDepthImages(4,:,:), 480, 640);
m5 = reshape(robotDepthImages(5,:,:), 480, 640);
m6 = reshape(robotDepthImages(6,:,:), 480, 640);
m7 = reshape(robotDepthImages(7,:,:), 480, 640);
m8 = reshape(robotDepthImages(8,:,:), 480, 640);
m9 = reshape(robotDepthImages(9,:,:), 480, 640);
m10 = reshape(robotDepthImages(10,:,:), 480, 640);
```

Listado de código 1 - Redimensionado de matrices de profundidad

Hecho esto, transformaríamos la matriz de profundidad a distancias en centímetros, tal y como hace el método *depthMapToCm*, que hacía uso de los parámetros intrínsecos del dispositivo Kinect para obtener las medidas reales.

```
function cmArray = depthMapToCm(depthArray)
    cmArray = double(depthArray);
    if size(cmArray) > 0
        cmArray(cmArray>=2047) = -100000;
        cmArray = abs(round(100./(double(cmArray)*-0.0030711016+3.3309495151)));
    endif
endfunction
```

Listado de código 2 – Función para obtener distancias a partir de los datos de profundidad

Usaremos este método para transformar nuestras matrices de profundidad a centímetros.

```
cm1 = depthMapToCm(m1);
cm2 = depthMapToCm(m2);
cm3 = depthMapToCm(m3);
cm4 = depthMapToCm(m4);
cm5 = depthMapToCm(m5);
cm6 = depthMapToCm(m6);
cm7 = depthMapToCm(m7);
cm8 = depthMapToCm(m8);
cm9 = depthMapToCm(m9);
cm10 = depthMapToCm(m10)
```

Listado de código 3 - Transformación de arrays de profundidad a medidas reales

Ahora mismo tenemos una matriz en la que cada coordenada (x, y) tiene asociado un valor de profundidad z, que simboliza la distancia al dispositivo Kinect. No obstante, este mapa de profundidad no será suficiente para realizar un mapa, sino que deberemos realizar una nueva transformación, convirtiendo la imagen de profundidad en una **nube de puntos**.

La nube de puntos añade la información de las coordenadas (x, y) a la información de profundidad, es decir, que en lugar de tener una única matriz cuyas coordenadas simbolizan los valores "z" de cada píxel, se tendrán tres matrices, cada una de las cuales simbolizarán los valores "x", "y" y "z" de cada píxel.

Así, si el punto (120, 801) tiene el valor "30" en la primera matriz, "59" en la segunda matriz y "84" en la tercera matriz, significará que sus coordenadas (x, y, z) serán (30, 59, 84).

Por lo tanto, la matriz original se descompondrá en tres matrices, cada una de las cuales almacenará el valor de cada punto en una de las tres coordenadas, coincidiendo la descomposición en profundidad (Z) con la matriz original.

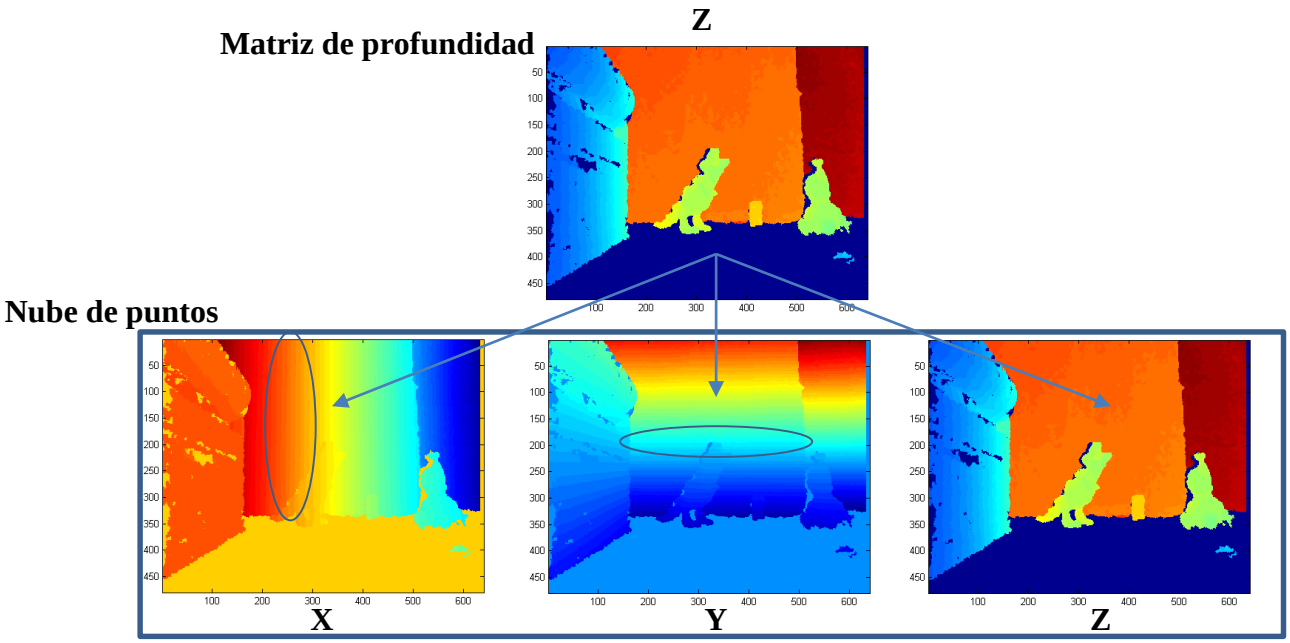


Figura 59 - Conversión de la imagen de profundidad en nube de puntos

Como podemos observar, el mapa de color de la matriz correspondiente al eje X muestra líneas verticales, ya que todos los valores de la columna tienen valores similares (misma coordenada X).

Del mismo modo, el mapa de color de la matriz Y muestra un efecto similar con las filas: muestran valores similares para la misma altura.

La matriz Z, por el contrario, mostrará el relieve de los objetos, puesto que se corresponde con la profundidad.

Por lo tanto, es importante tener en cuenta cómo Kinect gestiona las coordenadas: la dirección derecha-izquierda se corresponderá con el eje X. La dirección abajo-arriba se corresponderá con el eje Y, mientras que la dirección atrás-adelante se corresponderá con el eje Z.

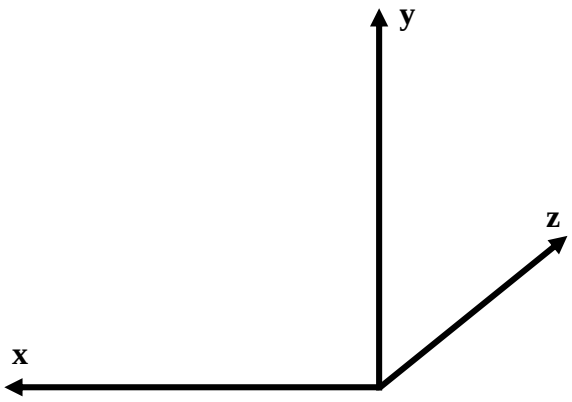


Figura 60 - Marco de referencia de coordenadas de Kinect

Una vez que tenemos definido el marco de referencia en el que trabajaremos, haremos la transformación a una nube de puntos a través del siguiente método que hace uso de los parámetros intrínsecos de Kinect, adaptado de [50]:

```
function pointCloud = depthArrayToPointCloud(depthArray)
    pointCloud = zeros([size(depthArray) 3]);
    height = size(depthArray, 1);
    width = size(depthArray, 2);

    f = 5.453;

    for xCoord = 1:width
      for yCoord = 1:height
        % Los valores de z ya los tenemos
        zValue = depthArray(yCoord,xCoord);
        pointCloud(yCoord, xCoord,3) = zValue;

        % Calculamos las distancias x
        pointCloud(yCoord, xCoord,1) = -(zValue/f)*((xCoord-width/2)*0.0093+0.063);

        % Calculamos las distancias y
        pointCloud(yCoord, xCoord,2) = -(zValue/f)*((yCoord-height/2)*0.0093+0.039);
      end
    end

    x = pointCloud(:,:,1);
    y = pointCloud(:,:,2);
    z = pointCloud(:,:,3);
    x(x==0)=NaN;
    y(y==0)=NaN;
    z(z==0)=NaN;

    pointCloud(:,:,1) = x;
    pointCloud(:,:,2) = y;
    pointCloud(:,:,3) = z;

    pointCloud = int16(pointCloud);
  end;
```

Listado de código 4 - Transformación de array de profundidad en nube de puntos

Obtendremos, por lo tanto, las nubes de puntos correspondientes a las distintas matrices de profundidad:

```
pc1 = depthArrayToPointCloud(cm1);
pc2 = depthArrayToPointCloud(cm2);
pc3 = depthArrayToPointCloud(cm3);
pc4 = depthArrayToPointCloud(cm4);
pc5 = depthArrayToPointCloud(cm5);
pc6 = depthArrayToPointCloud(cm6);
pc7 = depthArrayToPointCloud(cm7);
pc8 = depthArrayToPointCloud(cm8);
pc9 = depthArrayToPointCloud(cm9);
pc10 = depthArrayToPointCloud(cm10);
```

Listado de código 5 - Conversión de matrices en nubes de puntos

Con esto habríamos completado la extracción de la información proporcionada por Kinect. No obstante, para construir un mapa deberemos hacer uso de la odometría y poder trasladar cada punto a su correspondiente lugar, tanto en posición como en orientación.

Hay que tener en cuenta que las medidas proporcionadas por Kinect son absolutas hacia el observador, por lo que deberemos contextualizarlas rotando y trasladando dichos puntos a los lugares que le corresponden.

Para girar un punto haremos uso de una matriz de rotación bidimensional, ya que el giro se realizará sobre el eje y, permaneciendo el ángulo sobre dicho eje invariante.

$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\text{sen}(\theta) \\ \text{sen}(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_0 \\ z_0 \end{bmatrix} \qquad (24)$$

Con lo que las coordenadas de los puntos x, y serían las siguientes tras la rotación:

$$x = xcos(\theta) - zsen(\theta) \qquad (25)$$

$$z = xsen(\theta) + zcos(\theta) \qquad (26)$$

En caso de que el punto de giro no se corresponda con el punto (0, 0), habría que realizar una traslación temporal del punto de giro antes de realizar el producto, devolviendo el punto al origen tras realizar esta operación, es decir:

$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\text{sen}(\theta) \\ \text{sen}(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_0 - x_{giro} \\ z_0 - z_{giro} \end{bmatrix} + \begin{bmatrix} x_{giro} \\ z_{giro} \end{bmatrix} \qquad (27)$$

En el caso que nos ocupa, el punto de giro siempre será el (0, 0), por lo que no deberemos preocuparnos por este problema.

El siguiente código realiza una rotación de una nube de puntos sobre el eje Y:

```
function rotatedPointCloud = rotatePointCloud(pointCloud, theta, xCenter, zCenter)
    rotatedPointCloud = pointCloud;

    % Extraemos los componentes individuales
    x = pointCloud(:,:,1);
    y = pointCloud(:,:,2);
    z = pointCloud(:,:,3);

    % Linealizamos la matriz para operar con un array unidimensional 1x(640x480)
    x_lineal = double(reshape(x,1,size(x,1)*size(x,2)));
    z_lineal = double(reshape(z,1,size(z,1)*size(z,2)));

    % Creamos un array de dos filas, en la que el primer índice hará referencia a x o a y,
    % y el segundo índice contendrá el valor correspondiente.
    v = [x_lineal;z_lineal];

    center =  repmat([xCenter;zCenter],1,length(x_lineal));

    % Matriz de rotación
    R = [cos(theta) -sin(theta) ; sin(theta) cos(theta)];
    s = v - center;

    % Rotamos
    so = R*s;

    %Devolvemos al origen
    vo = so + center;

    xRotated = vo(1,:);
    zRotated = vo(2,:);

    Xr = int16(reshape(vo(1,:), size(x,1), size(x,2)));
    Zr = int16(reshape(vo(2,:), size(z,1), size(z,2)));

    rotatedPointCloud(:,:,1) = Xr;
    rotatedPointCloud(:,:,3) = Zr;
end
```

Listado de código 6 - Rotación de los valores X,Z de una nube de puntos

Como último paso realizaríamos la traslación de los puntos a su verdadera posición, para lo cual nos limitaríamos a sumar la actual posición x, z:

```
function transformedPointCloud = transformPointCloud(pointCloud, xCenter, zCenter, theta)
    % Rotamos la nube de puntos
    transformedPointCloud = rotatePointCloud(pointCloud, theta, 0, 0);

    % Desplazamos la nube de puntos
    transformedPointCloud(:,:,1) = transformedPointCloud(:,:,1) + xCenter;
    transformedPointCloud(:,:,3) = transformedPointCloud(:,:,3) + zCenter;
end
```

Listado de código 7 – Transformación completa de la nube de puntos

Con estas herramientas ya tendríamos todo lo necesario, junto con los métodos descritos en el *Modelo diferencial*, para generar un mapa tridimensional del entorno.

El siguiente apartado, *Resultados*, describe los resultados obtenidos al realizar una prueba realizada mediante este método.

3. Resultados

3.1. Experiencia de usuario

Es importante destacar que el objetivo de este proyecto no es realizar un análisis exhaustivo de técnicas de localización o generación de mapas, sino de adaptar y mejorar aquellas herramientas ya existentes para que puedan ser utilizadas de forma sencilla por un alumno que quiera conectarse a un laboratorio remoto en el que se encuentre el robot.

El alumno que haga uso de la aplicación deberá acceder a UNILabs para, posteriormente, acceder a la experiencia correspondiente. Una vez dentro, se visualizará el siguiente entorno:

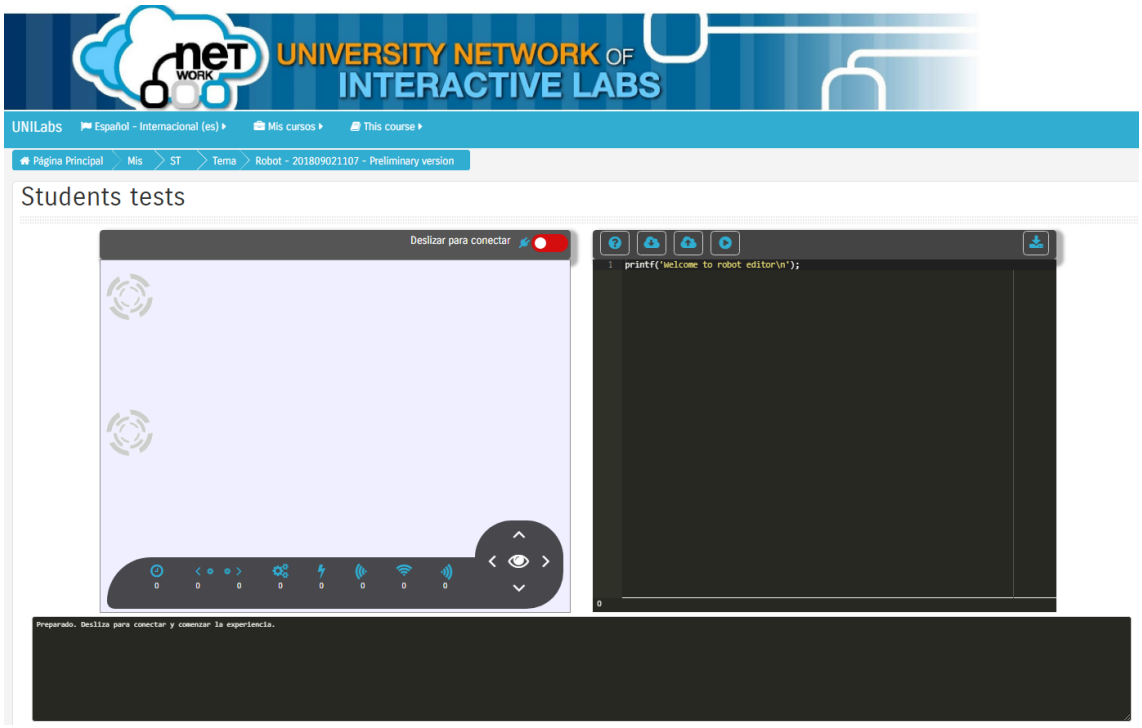


Figura 61 - Integración de la experiencia en UNILabs

El espacio de trabajo está dividido en tres secciones:

- Sección operativa (izquierda)
- Sección declarativa (derecha)
- Sección informativa (inferior)

A continuación, se describen los elementos de cada sección:

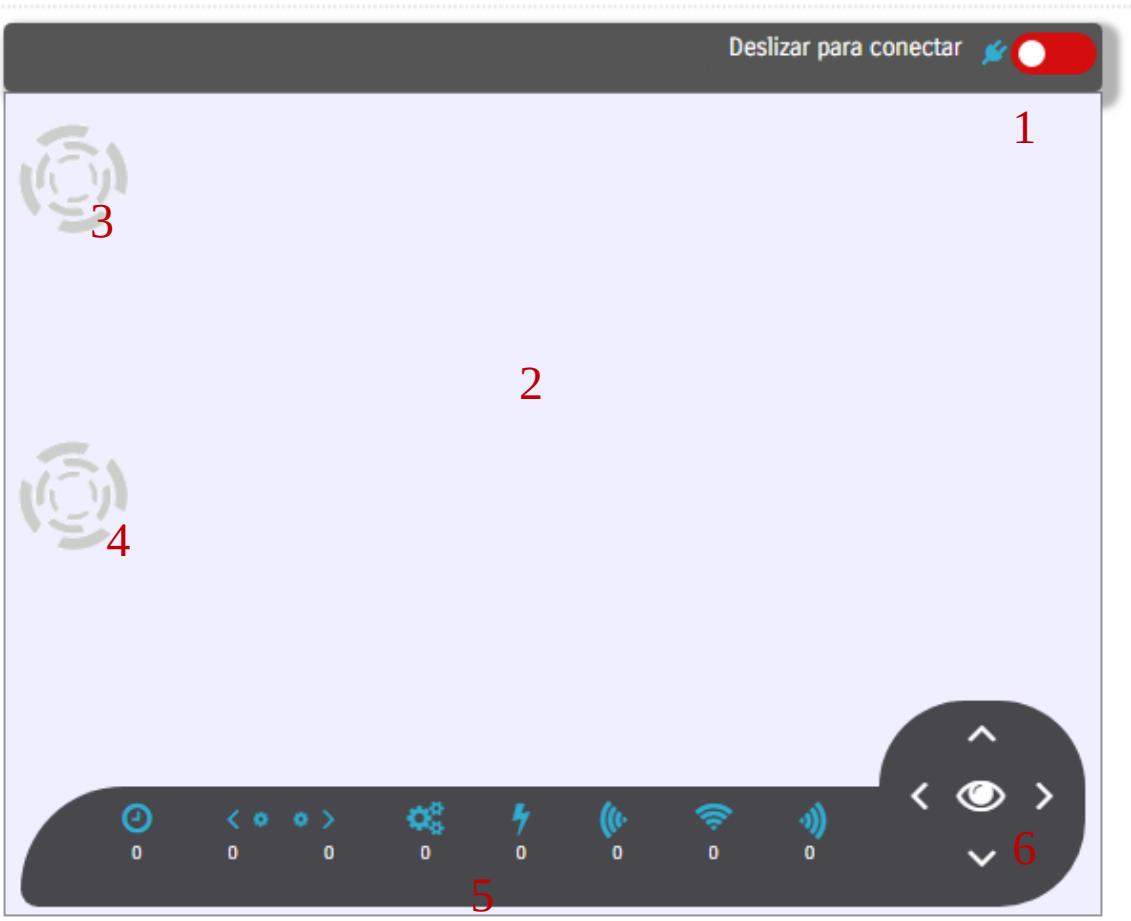


Figura 62 - Sección operativa

La sección operativa permite visualizar y teleoperar el robot a través de los controles de acción situados en la esquina inferior derecha.

Los elementos que componen la vista se detallan a continuación:

- **1**: Botón de conexión y desconexión de sesión. Inicia la sesión con el robot, cargando los servicios necesarios para ello. Es el primer paso a realizar para poder hacer uso del robot.
- **2**: Ventana de streaming del robot: visualizará en tiempo real la emisión de la cámara situada en la parte frontal del robot. La emisión comenzará tan pronto como se realice la conexión, y finalizará al cierre de ésta.
- **3**: Ventana de visualización de imágenes de profundidad. Si el usuario solicita la información actual del robot mediante la pulsación del icono del ojo situado en *6*, la imagen de profundidad obtenida se mostrará en esta localización.
- **4**: Ventana de visualización de marcadores ArUco. Si el usuario solicita la información actual del robot mediante la pulsación del icono del ojo situado en *6*, la imagen del techo

aumentada con la información sobre los marcadores ArUco obtenida se mostrará en esta localización.

- 5: Estado del robot. Muestra el contenido del último mensaje recibido desde el robot correspondiente a las lecturas de todos sus sensores, que son los siguientes:
    - Diferencia de tiempo desde la última lectura.
    - Cuentas del encoder izquierdo desde la última lectura.
    - Cuentas del encoder derecho desde la última lectura.
    - Estado del motor (1=encendido, 0=apagado).
    - Estado del láser (1=encendido, 0=apagado). El láser, no obstante, no se utiliza en esta experiencia.
    - Distancia en centímetros detectada por el sensor de infrarrojos izquierdo.
    - Distancia en centímetros detectada por el sensor de infrarrojos frontal.
    - Distancia en centímetros detectada por el sensor de infrarrojos derecho.
- 6: Panel de control. Permite mover el robot, así como solicitar una lectura de los datos mediante el envío de la señal *Keep Alive*.

Para comenzar la sesión deberemos pulsar el *slider* rotulado con *Deslizar para conectar*, que hará que se muestre el mensaje "Conectando".

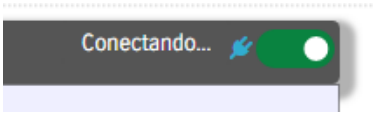


Figura 63 - Inicio de la conexión

Una vez que se haya establecido la conexión, se realizará una primera actualización de los sensores del robot y se mostrará el resultado en el log del sistema.

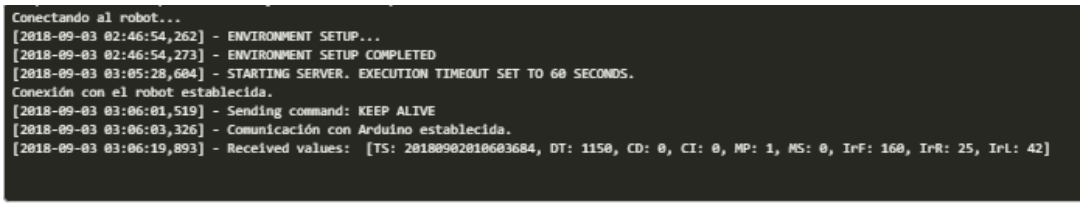


Figura 64 - Log del sistema

Tras realizar la conexión, se mostrará la etiqueta *Conectado* en la esquina superior derecha y se solicitarán las correspondientes imágenes a Kinect y a ArUco, mostrándose en sus correspondientes posiciones.

Igualmente, la información de los sensores se actualizará con la información disponible en el momento de realizar la operación.

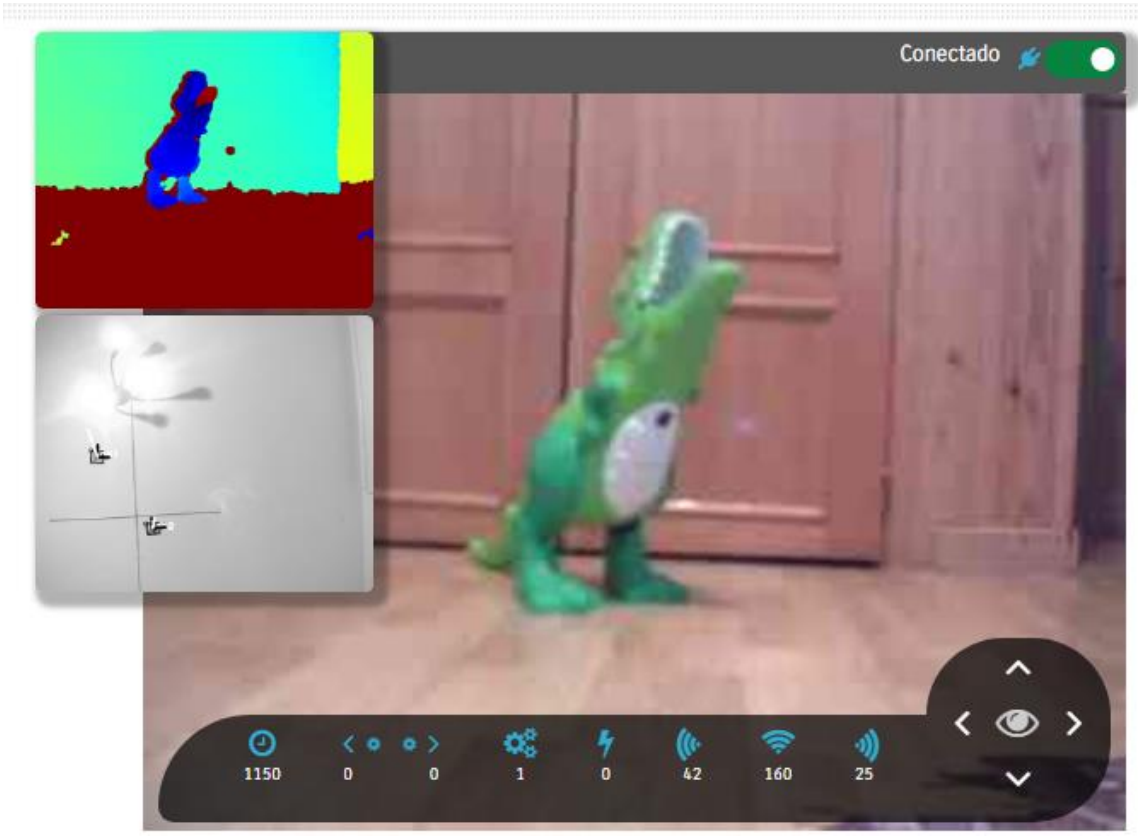


Figura 65 - Sección operativa del entorno

En cuanto a la sección declarativa, permitirá al alumno declarar y enviar al robot su código fuente, haciendo que se ejecuten aquellas ordenes declaradas en Octave.

Adicionalmente, permite efectuar operaciones de almacenamiento y recuperación de ficheros en el espacio de trabajo del alumno, así como acceder a la ayuda de la API del robot.

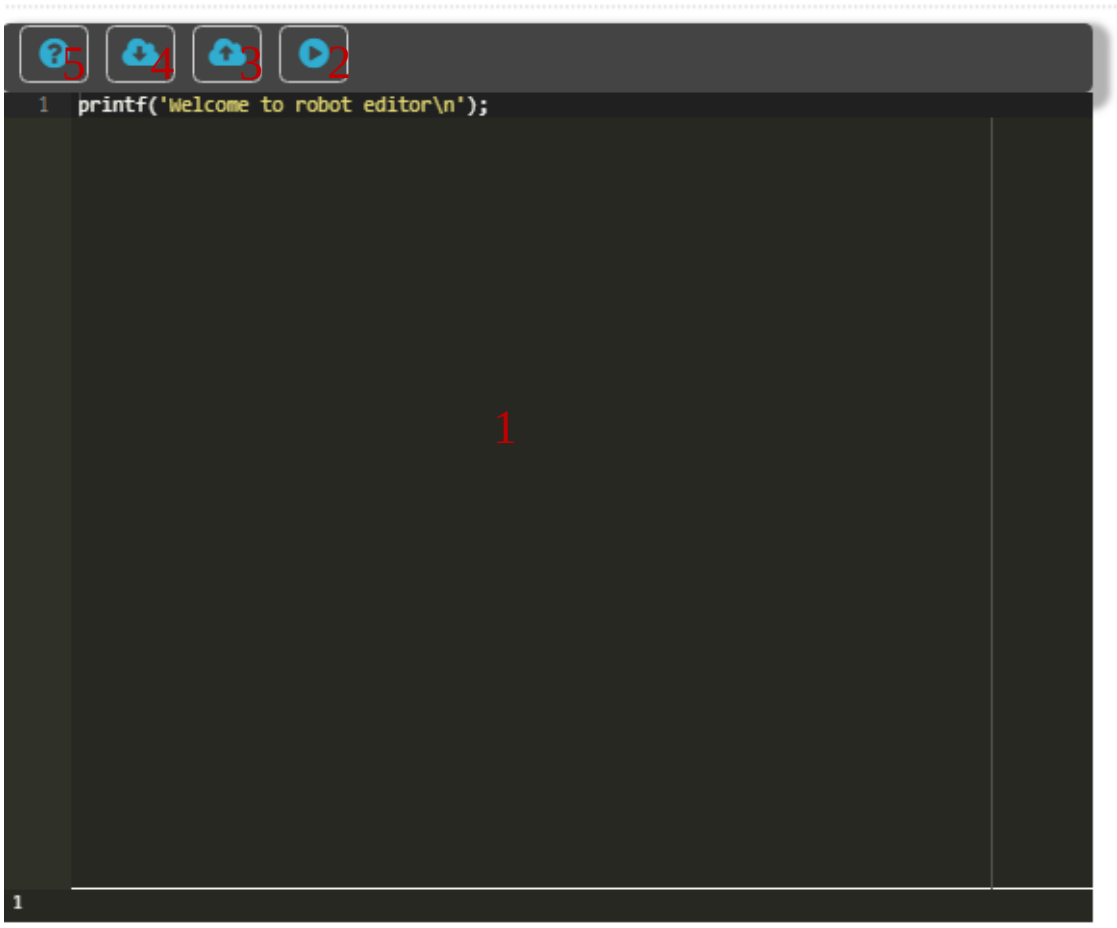


Figura 66 - Sección declarativa del entorno.

- 1: Editor de texto. Permite escribir el código Octave que será enviado al robot.
- 2: Enviar y ejecutar. Envía el código al robot, haciendo que se ejecute de forma instantánea, almacenando los resultados en la caché del alumno.
- 3: Guardar fichero en el espacio de trabajo. Permite guardar el contenido actual en un fichero .txt.

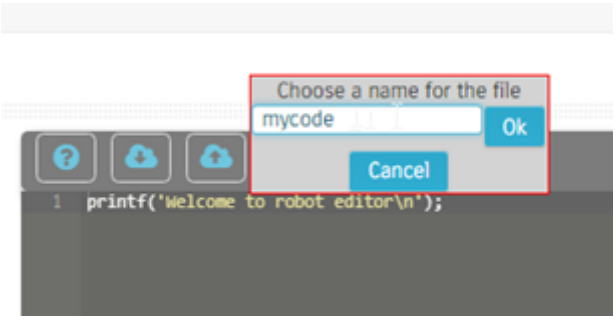


Figura 67 - Guardado de un archivo en el espacio de trabajo del alumno

- 4: Cargar fichero desde el espacio de trabajo. Permite recuperar el contenido de un fichero del espacio de trabajo y cargarlo en el editor de texto.
- 5: Ayuda de la API. Muestra la ayuda con la información de los métodos del robot que el alumno puede utilizar.

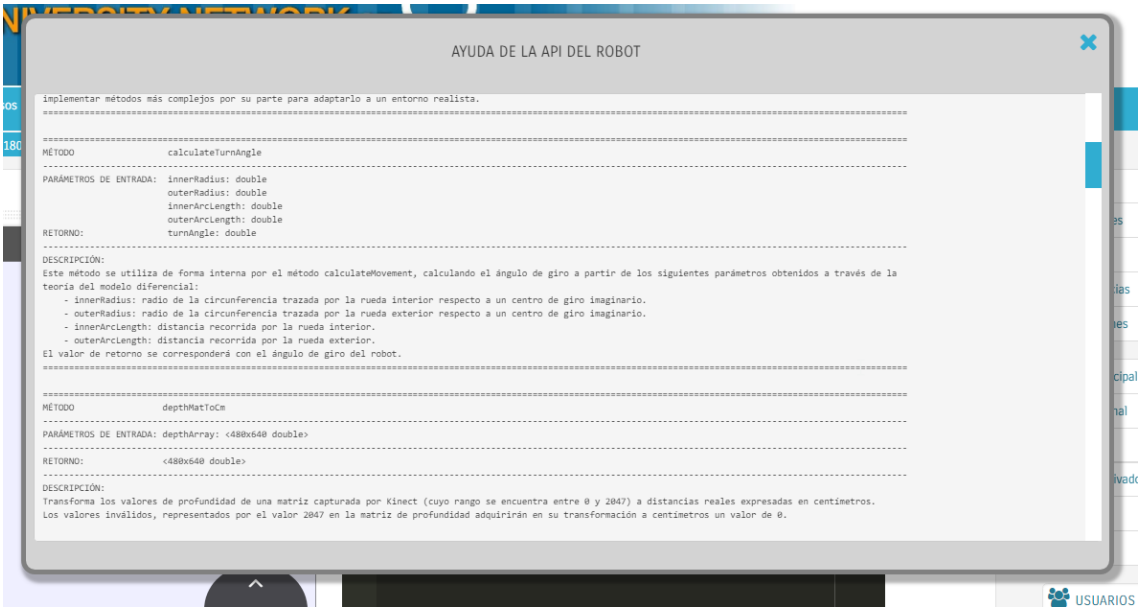


Figura 68 - Ayuda de la API del robot

Una vez que se haya finalizado la sesión se esperará unos instantes a que se visualice el botón de descarga del fichero .mat, que no estará disponible durante la conexión.

Figura 69 - Editor de texto, sin botón de descarga

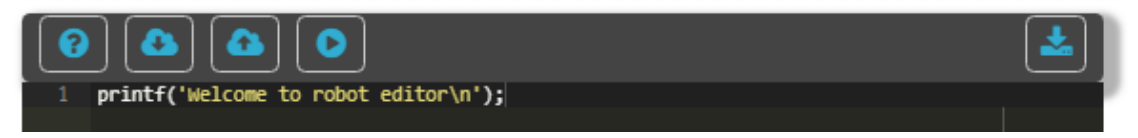


Figura 70 - Editor de texto, con botón de descarga

Una vez pulsado el botón, se procederá con la descarga del fichero *robot.mat*, que contendrá toda la información de la sesión preparada para su tratamiento *offline*, tal y como se explica en la sección *Persistencia*.

3.2. Generación de mapas

Tal y como se recoge en la sección *Construcción de mapas*, se ha planteado un ejemplo de construcción de mapas utilizando las técnicas descritas en dicha sección.

Para ello se ha preparado el siguiente entorno:

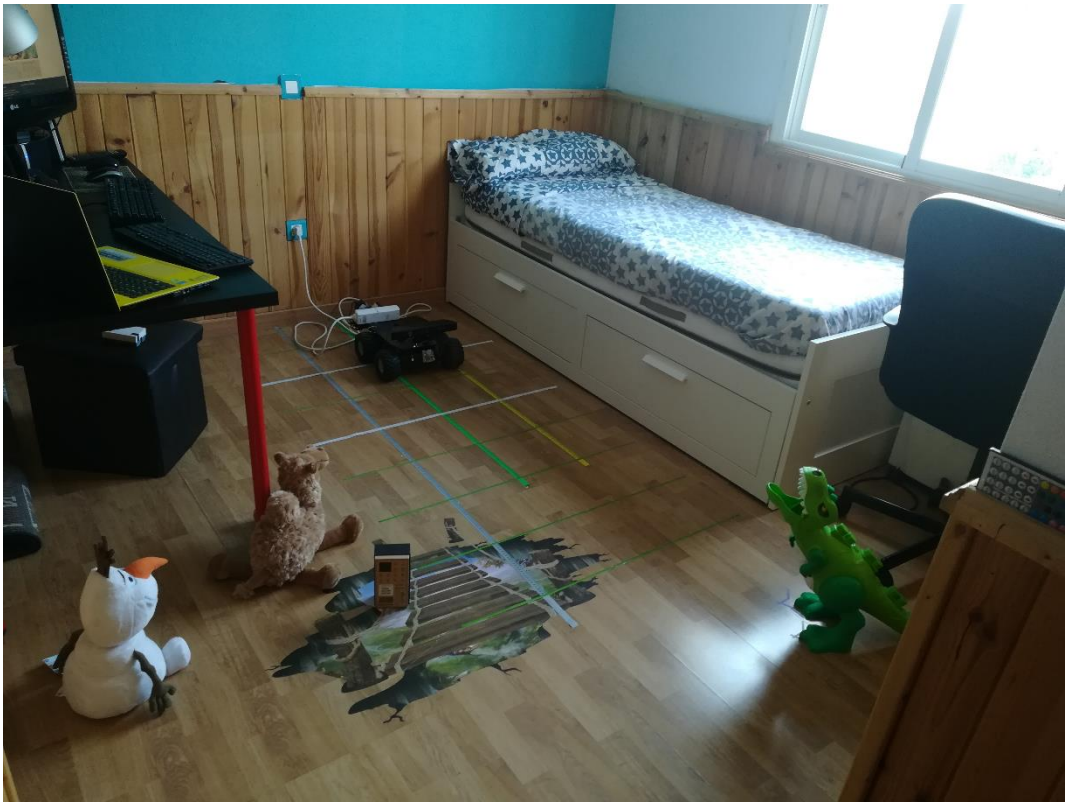


Figura 71 - Entorno de pruebas para la generación de mapas

Se han añadido además marcadores ArUco en el techo para comprobar que se detectaban correctamente:

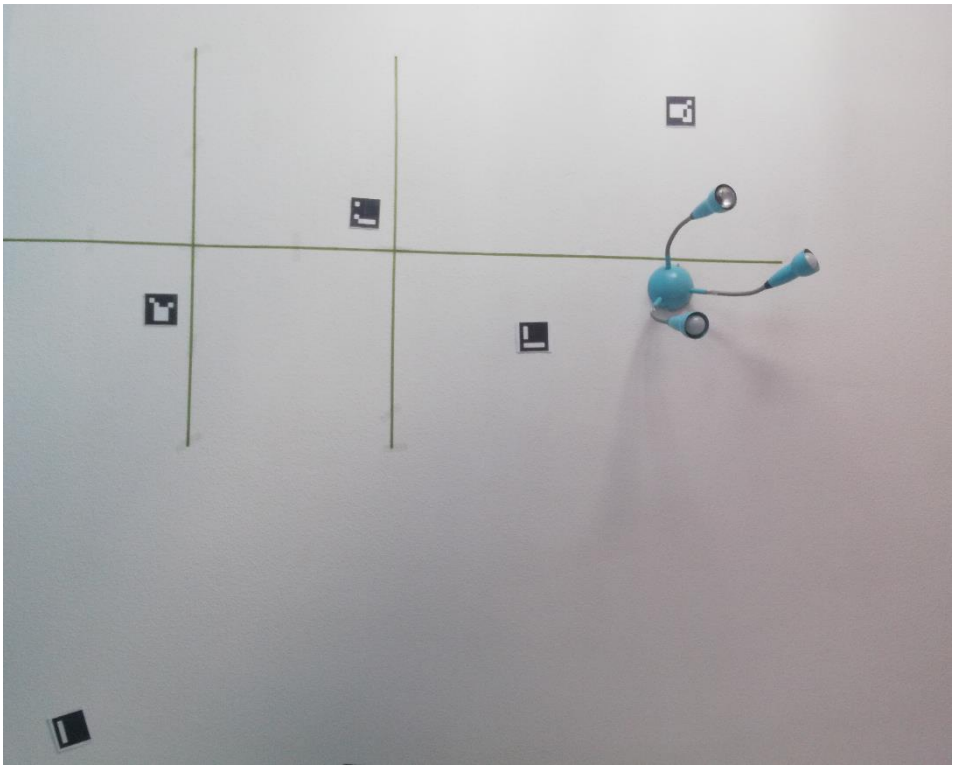


Figura 72 - Marcadores ArUco del entorno de pruebas

El siguiente esquema define de forma aproximada la estancia en la que se realizan las pruebas:

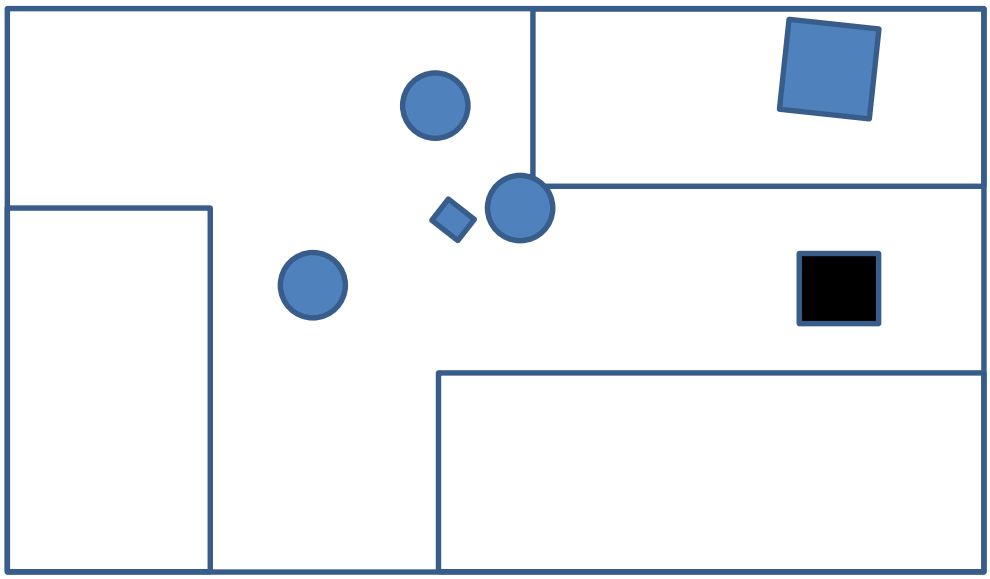


Figura 73 - Esquema de la estancia en la que se realizan las pruebas

Comenzamos colocando el robot en la posición (180, 75) y realizando una lectura inicial del entorno:

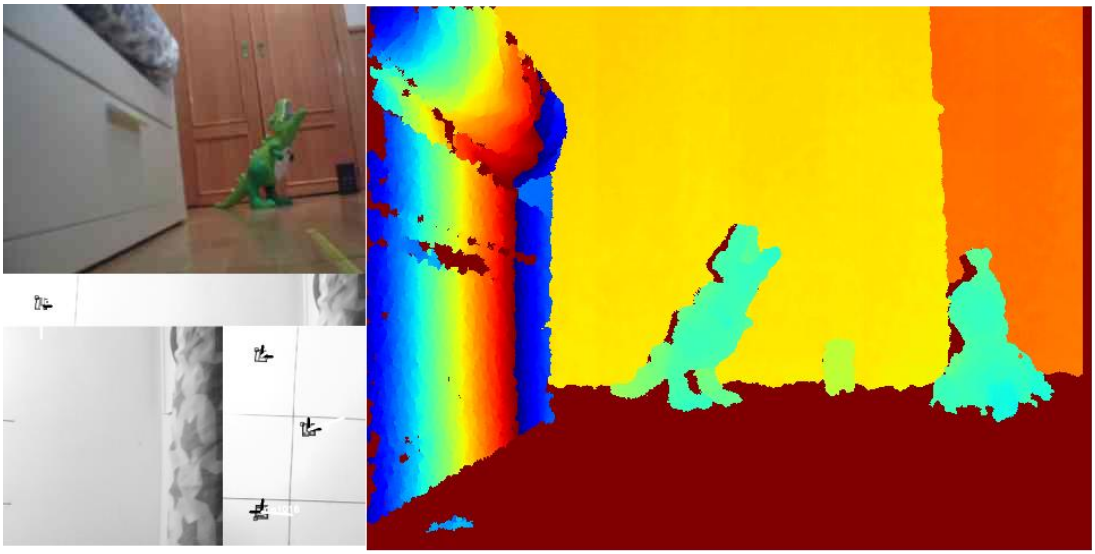


Figura 74 - Lectura inicial

Se detectan tres marcadores ArUco en el mismo instante, cuyos identificadores son 2, 1 y 1016, y cuyas esquinas se encuentran en las siguientes posiciones:

140	173	162	172	163	193	140	194
59	42	79	44	77	64	56	62
56	338	58	316	80	318	79	340

La visualización del array de profundidad produce la siguiente gráfica:

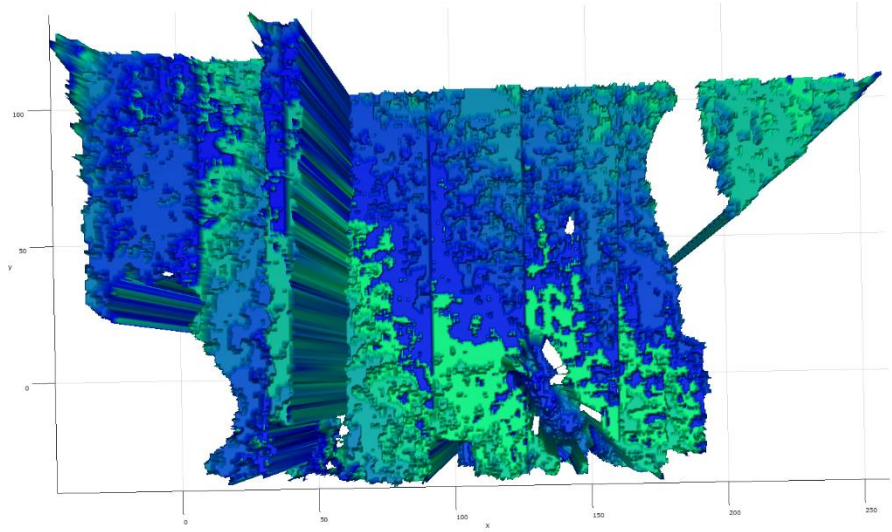


Figura 75 - Array de profundidad inicial

Realizando el proceso indicado en Construcción de mapas se genera la siguiente nube de puntos, trasladada al punto de origen del robot:

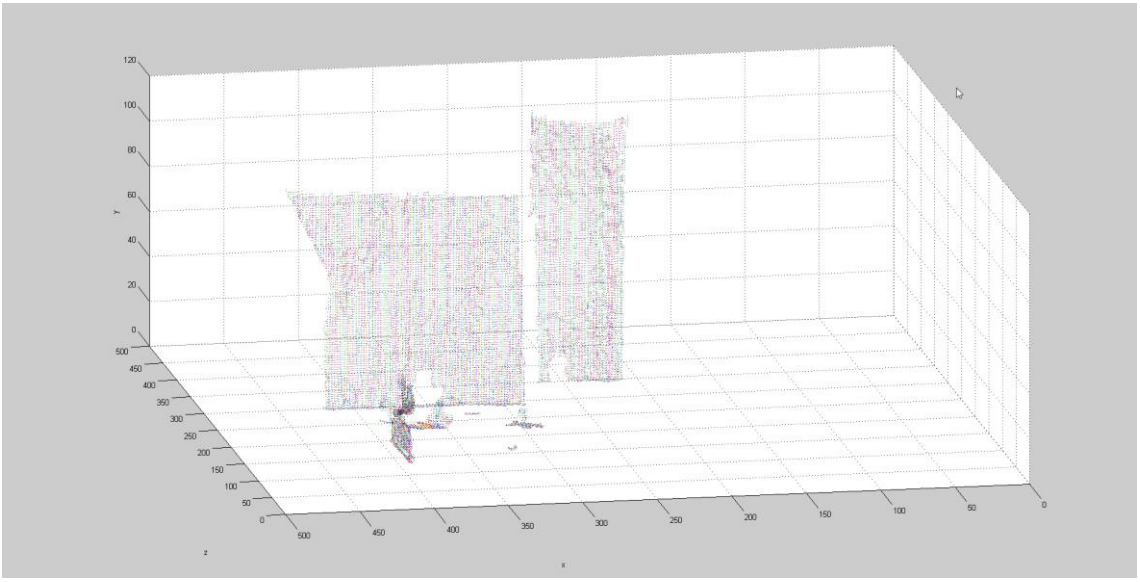


Figura 76 - Nube de puntos inicial

Se observa la localización de la cama, el armario del fondo y la figura del dinosaurio verde. Además, la localización de los puntos se aproxima bastante a su posición real en el espacio.

Hacemos avanzar al robot, obteniendo lo siguiente:

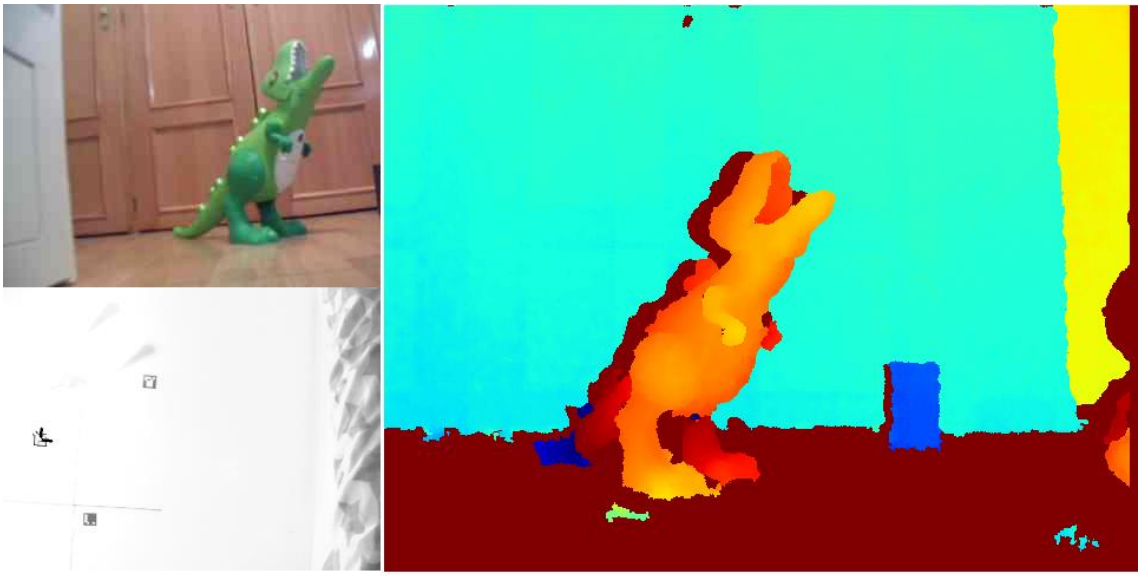


Figura 77 - Generación de mapas, visualización del paso 2

La odometría nos proporciona la siguiente información:

(181.288258, 151.098502 / 0.033854)

La nube de puntos de la captura de profundidad tras la transformación nos muestra la imagen siguiente:

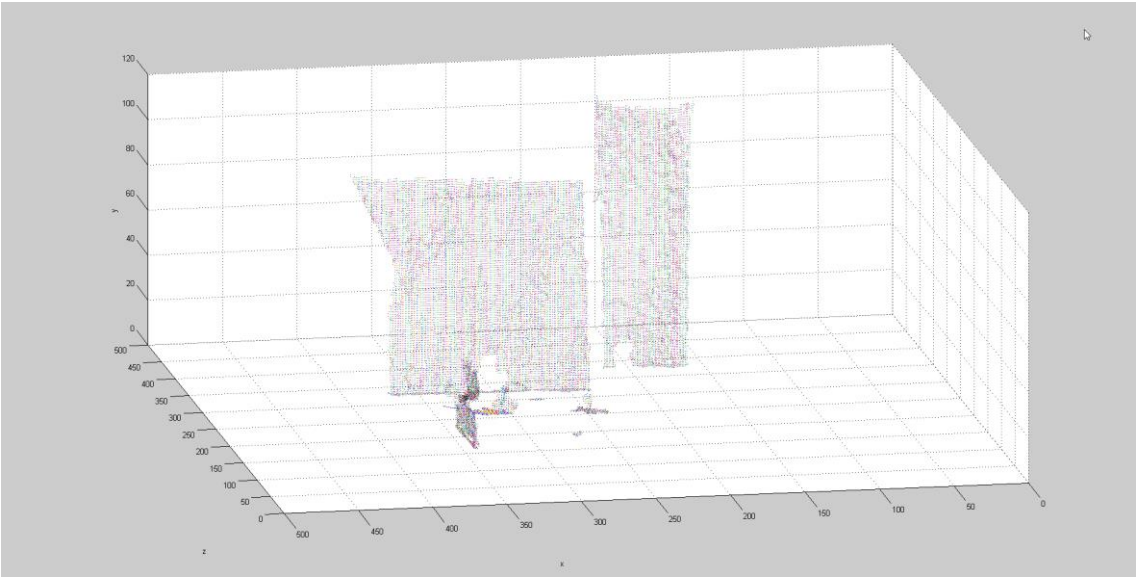


Figura 78 - Nube de puntos, paso 2

Tal y como podemos observar, el resultado es muy similar, ya que se ha recorrido poca distancia y en línea recta.

Realizaremos ahora un pequeño giro a la derecha:

Figura 79 - Generación de mapas, visualización del paso 3

Ahora la odometría nos dice que se ha producido el siguiente desplazamiento y la siguiente rotación:

(180.978189, 150.257007 / 0.672219)

La nube de puntos generada tras la transformación de la nube de puntos muestra ahora un aspecto distinto a las anteriores, puesto que la pequeña rotación ha cambiado la perspectiva y la localización de los puntos:

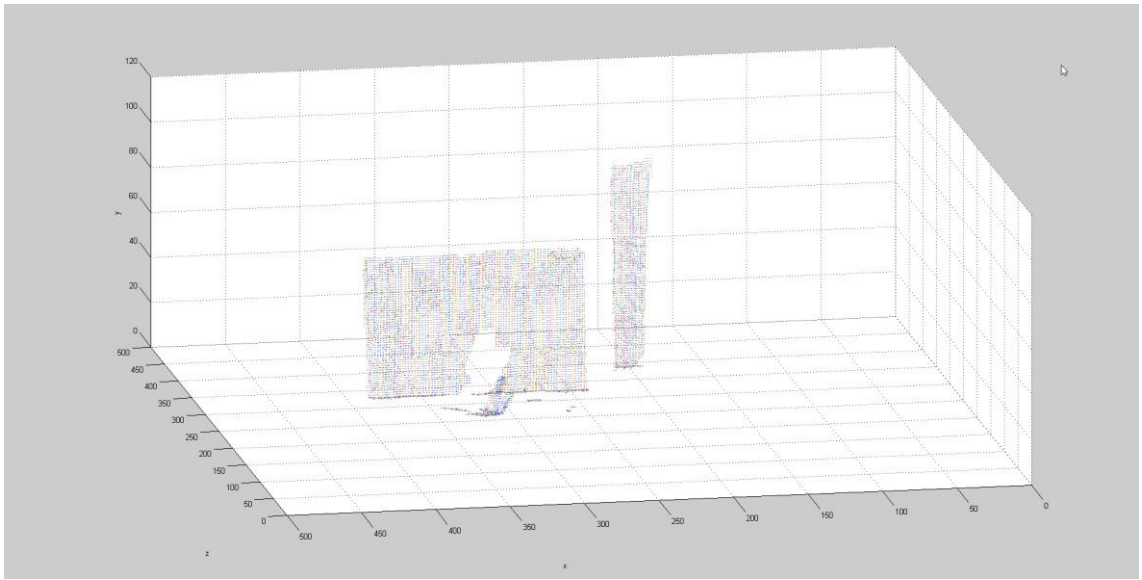


Figura 80 - Nube de puntos, paso 3

Seguimos rotando para intentar hacer un plano tridimensional del entorno. En el siguiente giro, obtenemos lo siguiente:

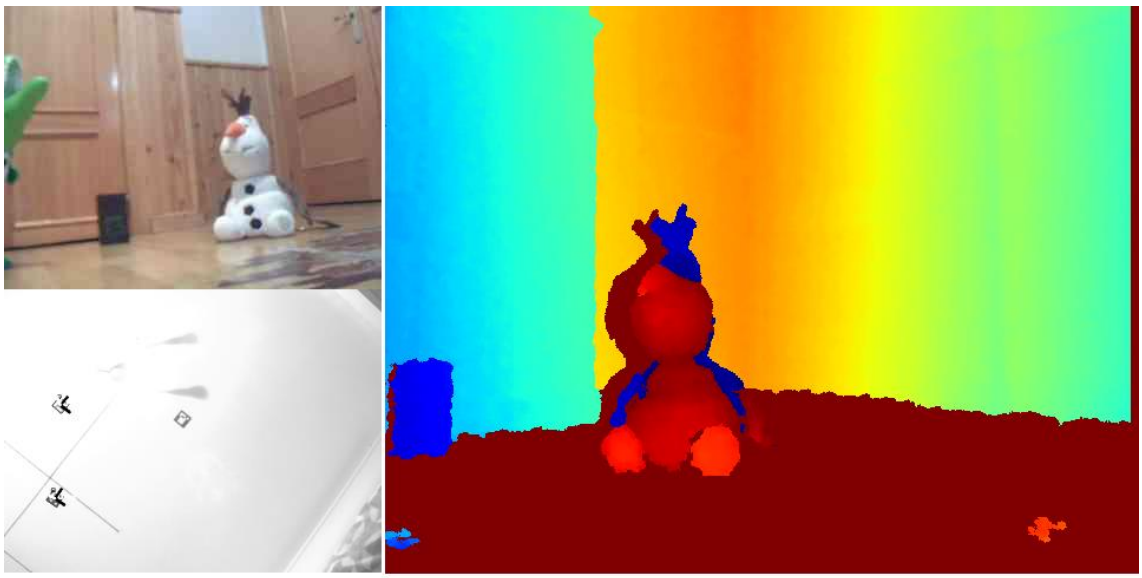


Figura 81 - Generación de mapas, visualización del paso 4

Los datos de odometría informan de la siguiente información:

(183.084781, 152.273333 / 0.942358)

La nube de puntos se muestra de la siguiente forma:

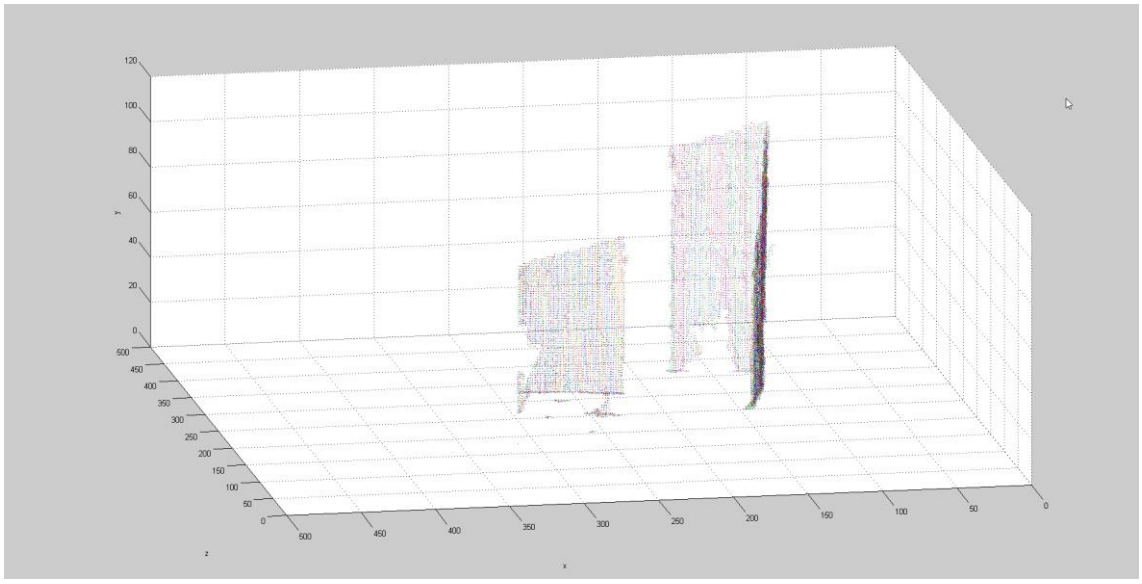


Figura 82 - Nube de puntos, paso 4

Como podemos observar, la esquina del fondo a la derecha se visualiza correctamente en la nube de puntos.
Realizamos una nueva medida tras continuar el giro.

Figura 83 – Generación de mapas, visualización del paso 5

Los datos de odometría son los siguientes:

(183.635146, 152.583245 / 1.173542)

La nube de puntos resultantes extiende la pared que se muestra a la derecha:

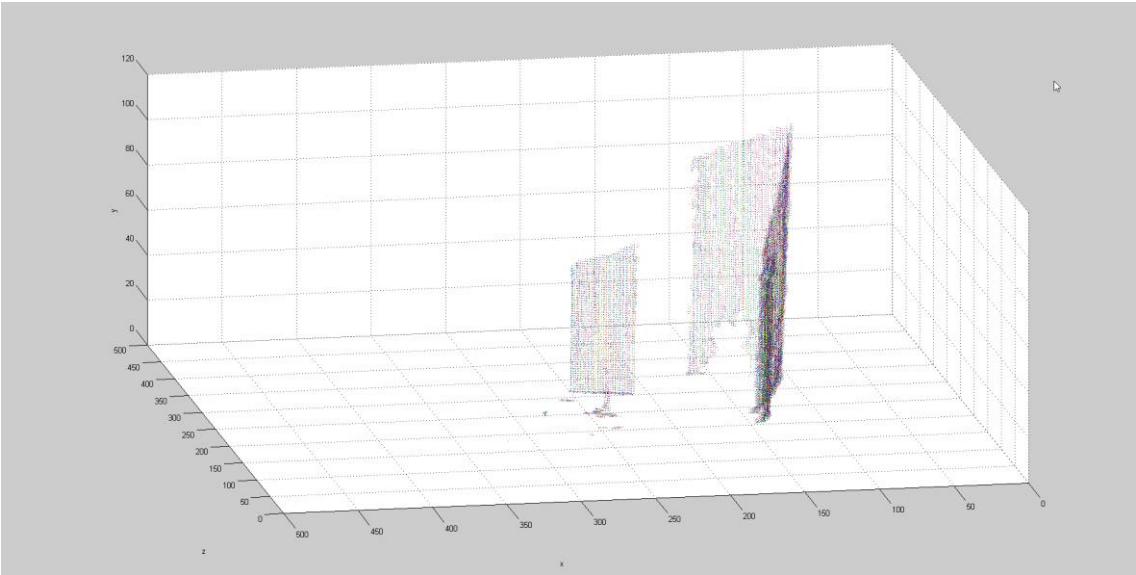


Figura 84 - Nube de puntos, paso 5

Siguiente giro:

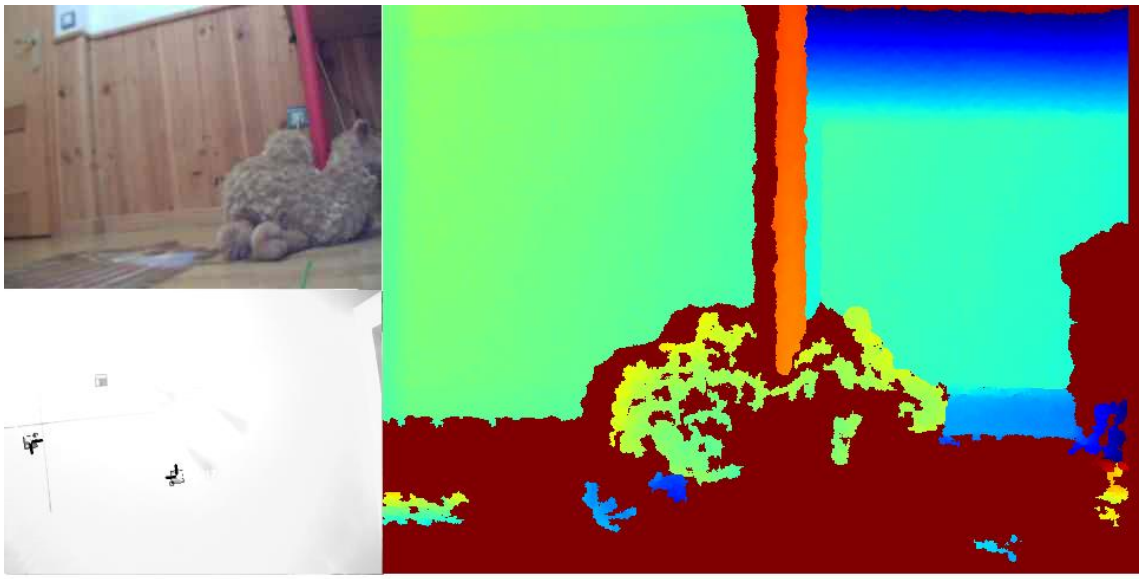


Figura 85 - Generación de mapas, visualización del paso 6

Odometría: (184.857377, 152.367406 / 2.317635)

La nube de puntos nos muestra claramente la rotación realizada hasta el momento.

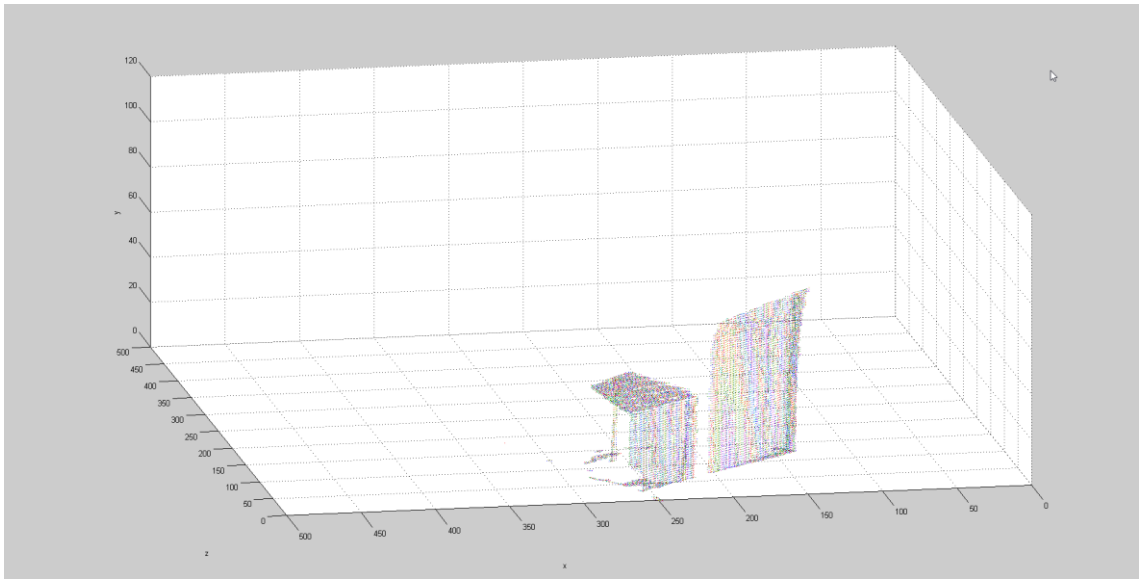


Figura 86 - Nube de puntos, paso 6

Al realizar la siguiente medida, podemos observar que la mayor parte de los píxeles adquieren valor inválido al estar demasiado cerca del sensor.

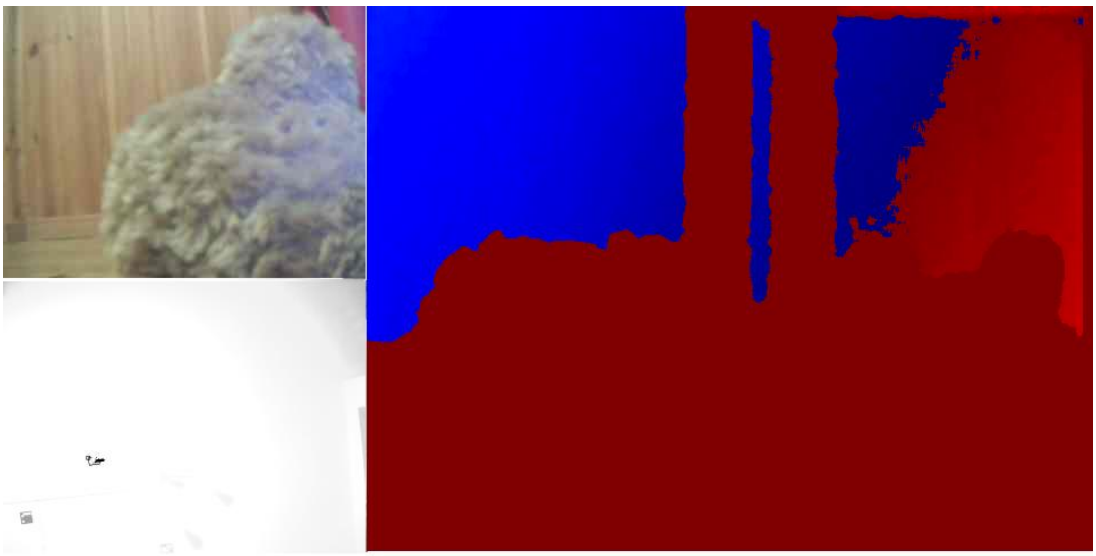


Figura 87 - Generación de mapas, visualización del paso 7

Esto como podemos imaginar, afecta a la generación de la nube de puntos, que se genera con muy pocos elementos.

Odometría: (227.075181, 112.515535 / 2.337113)

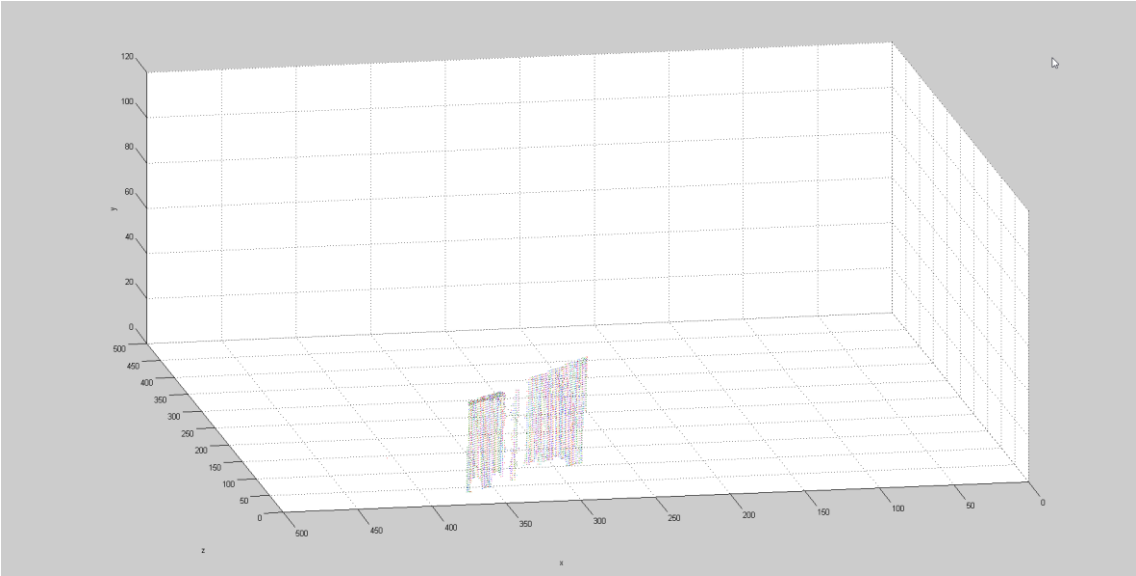


Figura 88 - Nube de puntos, paso 7

El nuevo giro sí que proporciona bastante información tridimensional, salvando aquello en primer plano, que se muestra con valor indeterminado.

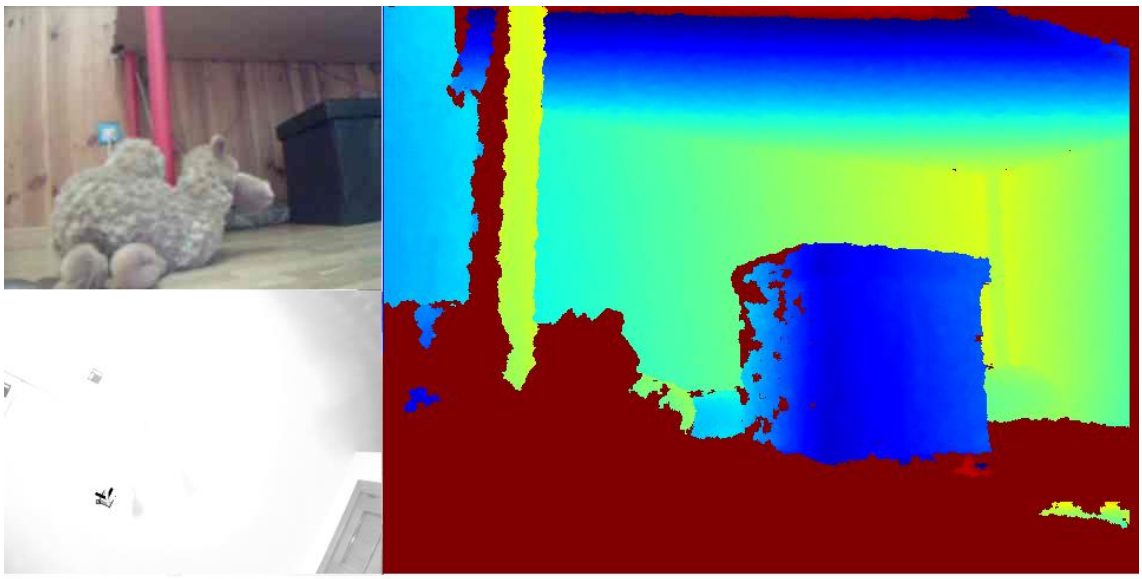


Figura 89 - Generación de mapas, visualización del paso 8

Odometría: (210.682508, 162.865740 / 3.316572)

Vemos cómo la mesa y la caja se modelan correctamente en la esquina de la habitación:

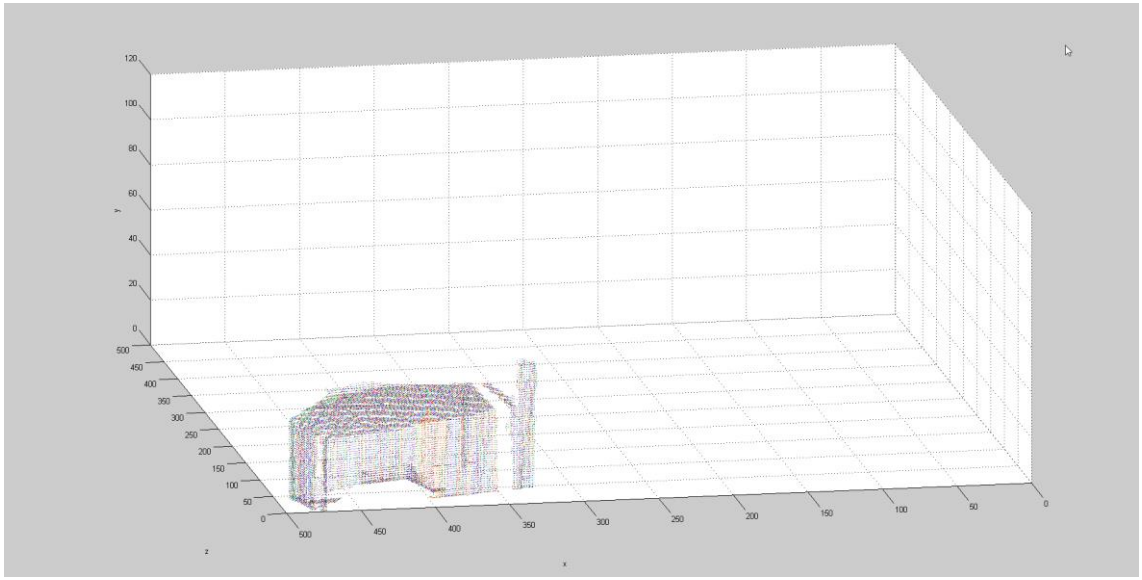


Figura 90 - Nube de puntos, paso 8

Seguimos girando y obteniendo datos:

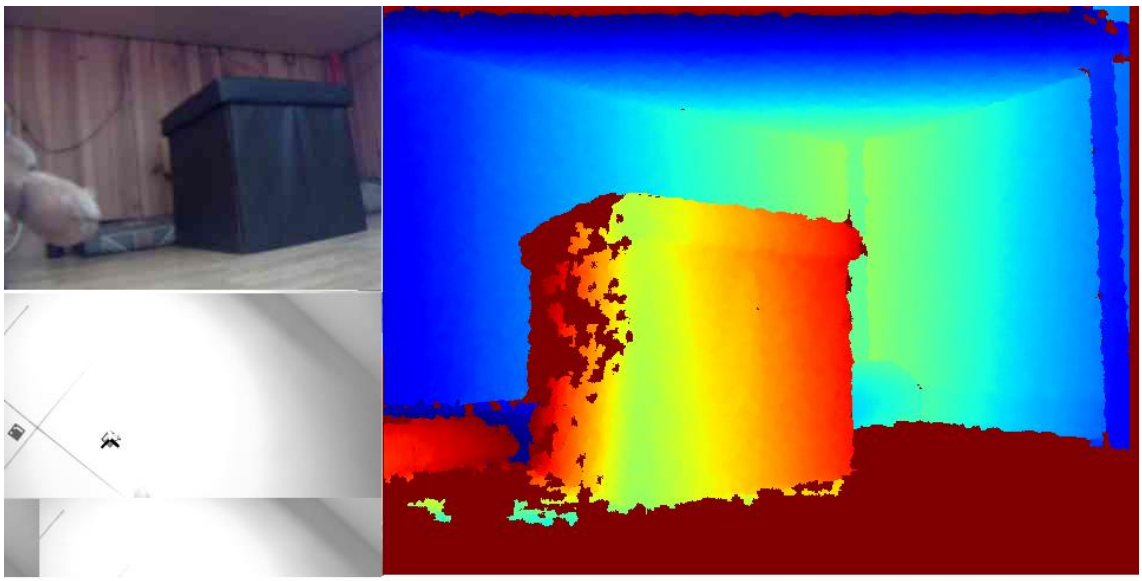


Figura 91 - Generación de mapas, visualización del paso 9

Odometría: (199.031271, 127.885970 / 3.609668)

Nuevamente, se añade la información al mapa tridimensional:

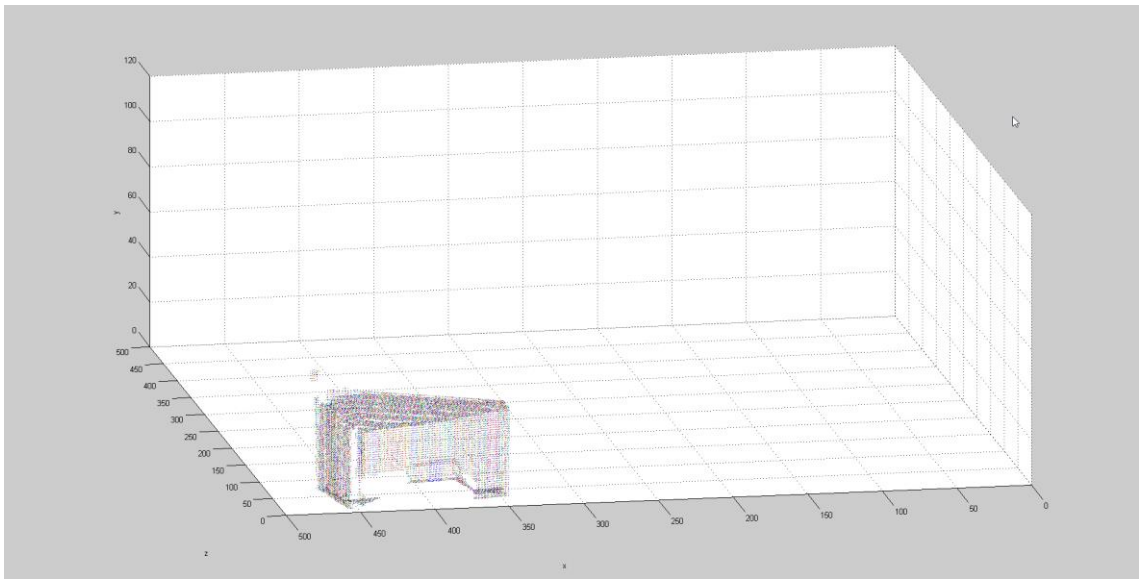


Figura 92 - Nube de puntos, paso 9

Un último giro hará que el robot se quede mirando en la dirección contraria a la que partió:

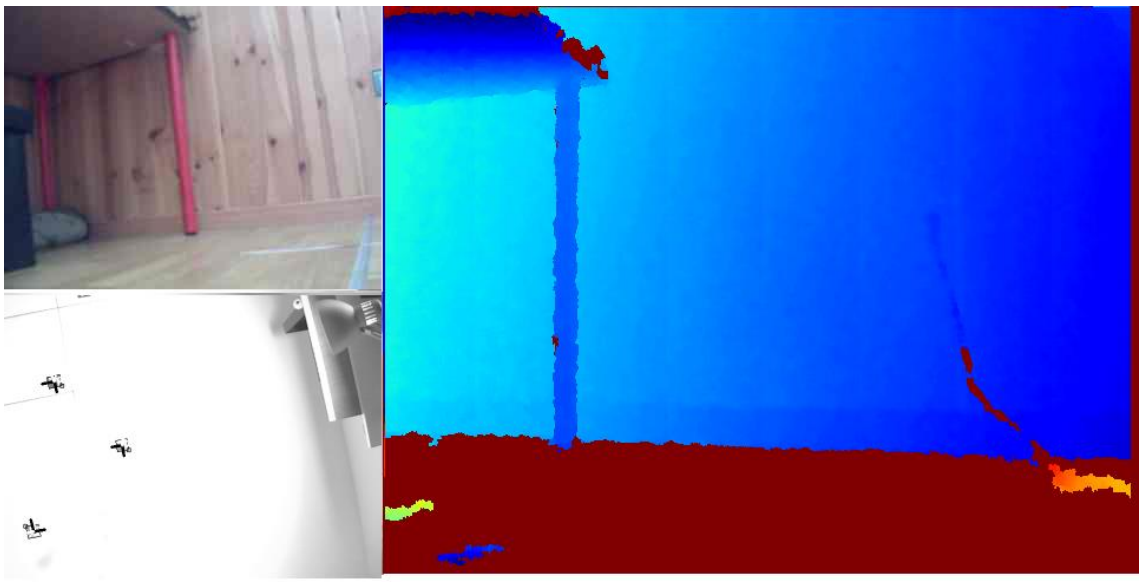


Figura 93 - Generación de mapas, visualización del paso 10

Odometría: (197.892028, 127.126372 / 4.638980)

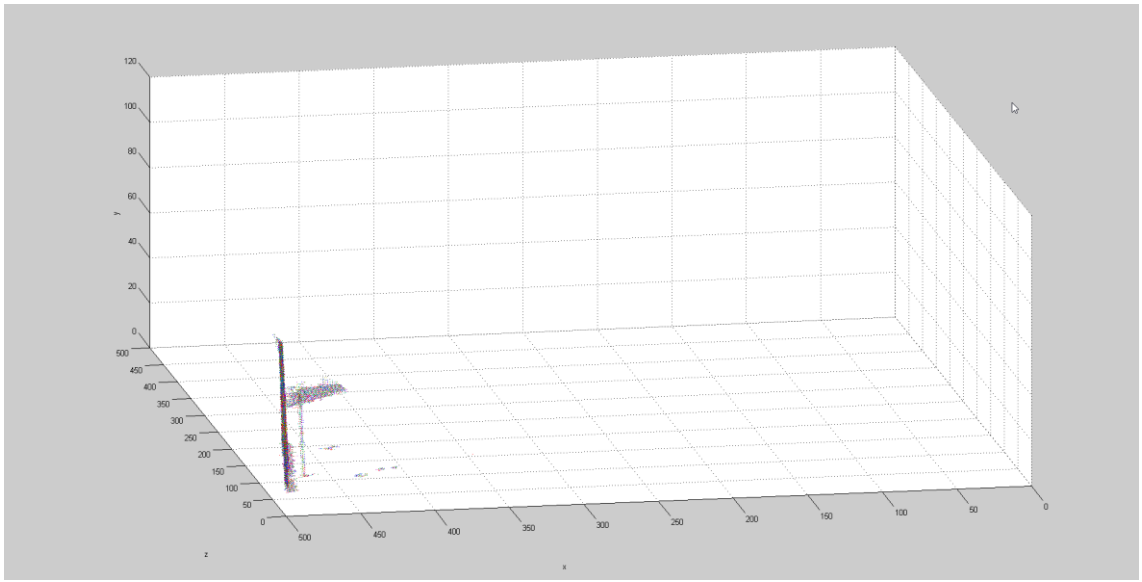


Figura 94 - Nube de puntos, paso 10

Una vez que todas las capturas se han realizado de forma independiente y se han generado sus correspondientes rotaciones y traslaciones, se juntan todas las medidas y se comprueba el resultado:

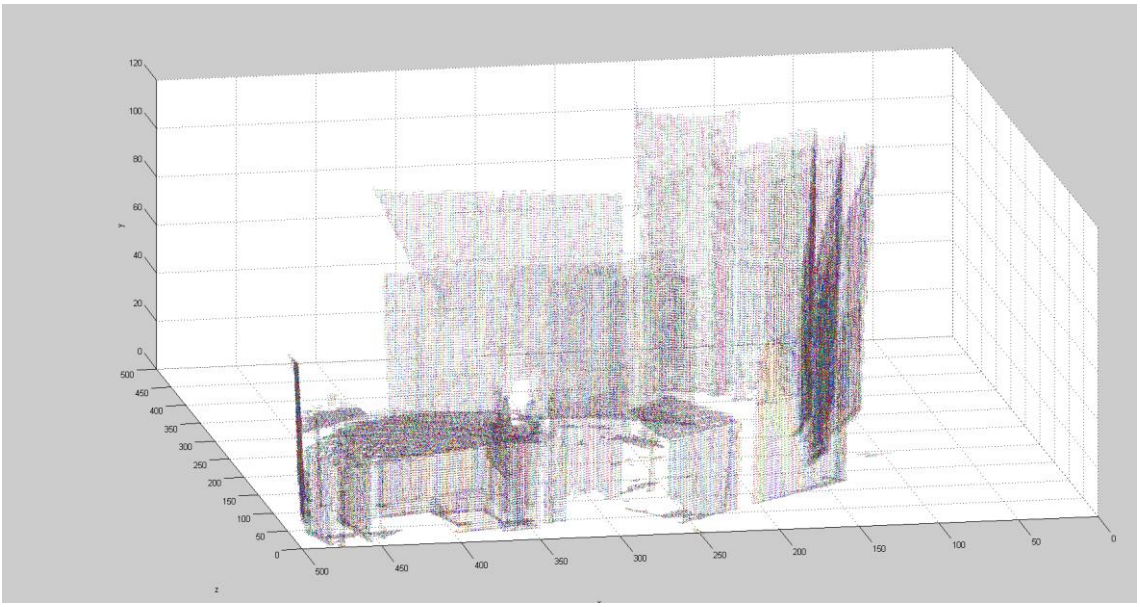


Figura 95 - Nube de puntos, fusión de todos los pasos intermedios

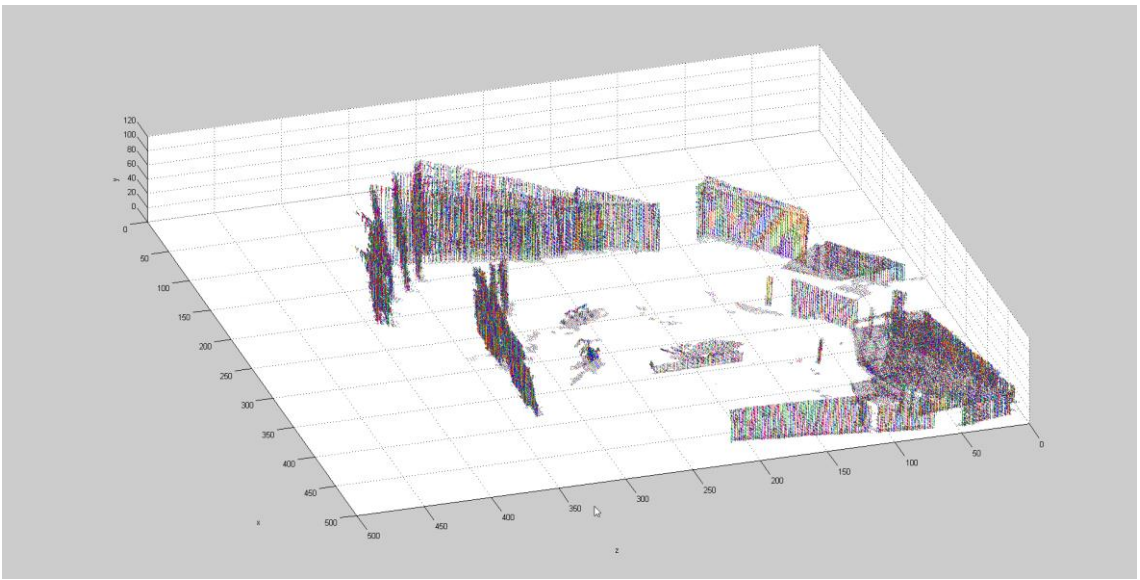


Figura 96 – Visualización de los errores en la odometría

Tal y como podemos observar, algunas medidas aparecen desplazadas. Esto se debe al error en la localización del robot, ya que los puntos que teóricamente son coincidentes aparecen ligeramente desplazados, aumentando la densidad en algunas localizaciones.

Para aumentar la eficacia de la localización pueden utilizarse diversas estrategias, como complementar el modelo diferencial con la localización absoluta mediante los marcadores ArUco o la utilización de algoritmos que simplifiquen la fusión de nubes de puntos, como por ejemplo Iterative Closest Point (ICP) [51].

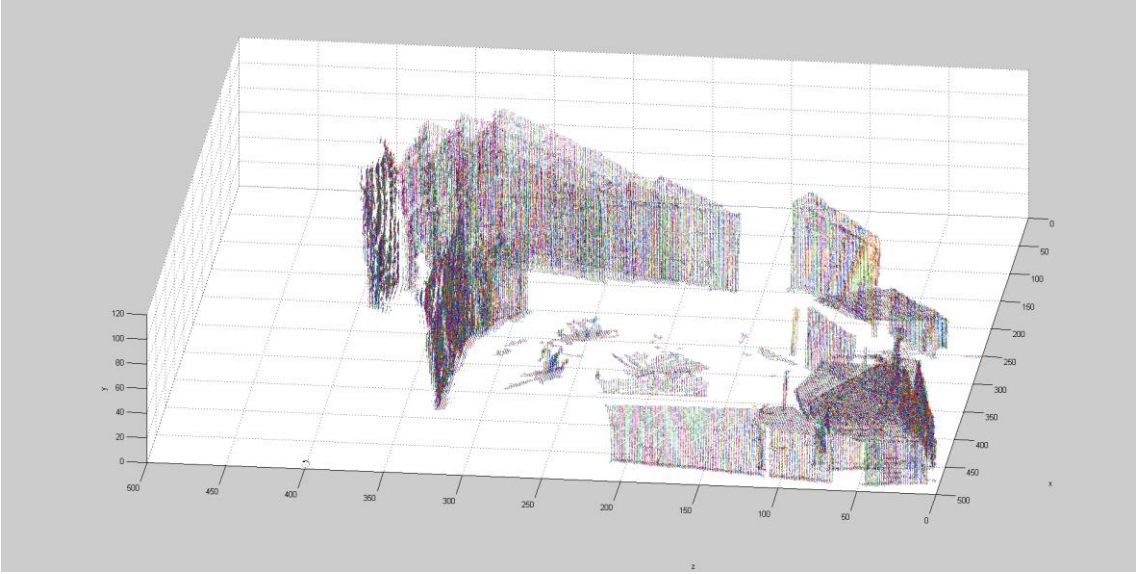


Figura 97 – Visión general de la habitación

4. Conclusiones

Con el paso de los años y la mejora de la tecnología, la robótica ha pasado de ser un elemento de ciencia-ficción a construirse por unos pocos euros, gracias a dispositivos de gran versatilidad, pero bajo precio, como Raspberry Pi, Arduino o Kinect.

Un elemento pensado como control de una videoconsola se ha convertido con el tiempo en un dispositivo versátil con gran potencial, hasta el punto de que es un estándar de facto en el entorno científico a la hora de realizar capturas en tres dimensiones, y todo ello pese a no contar con el soporte directo ni el apoyo de sus desarrolladores.

La conjunción de estos elementos ha dado lugar a un proyecto que permite la actuación y la obtención de datos de diversos sensores, abriendo la puerta a permitir aplicar la teoría de la robótica y del control en un dispositivo de altas prestaciones y bajo precio de forma remota.

No obstante, el reto de este proyecto no ha sido reinventar la rueda siguiendo los mismos pasos que otros compañeros, sino aprender al máximo de su trabajo para optimizarlo y mejorarlo con el objetivo de crear una experiencia de usuario satisfactoria para que otros alumnos puedan aplicar sus conocimientos de forma telemática.

En cualquier caso, si bien la construcción del robot ha quedado fuera del ámbito del proyecto, el diseño del prototipo local ha servido para familiarizarse con los componentes en los que se basa, facilitando enormemente la tarea de codificación. Gracias al ajustado precio de sus componentes, cualquiera puede construir un dispositivo de estas características, por lo que el acceso a este tipo de conocimientos y su aplicación a robots reales está al alcance de cualquiera, especialmente cuando la tecnología subyacente (OpenCV, Freenect, Octave, …) es libre y gratuita.

Como conclusión final, puede afirmarse que se han cubierto correctamente las especificaciones del proyecto, para lo cual ha sido necesario estudiar a fondo los trabajos relacionados tanto con el hardware como con las tecnologías necesarias para configurarlas e integrarlas en un único dispositivo. Como resultado se ha obtenido un robot de fácil configuración e instalación plenamente integrado en el entorno UNILabs, que permite ser teleoperado y programado por el alumno a través de una interfaz visualmente atractiva e intuitiva, así como conservar todos aquellos datos capturados por el robot durante la experiencia del usuario. Finalmente, se ha procurado aprovechar al máximo el potencial del robot, procurando proporcionar al estudiante del robot múltiples vías que le permitan generar un mapa de su entorno.

4.1. Líneas futuras

Debido a los limitados recursos de Raspberry Pi y a la cantidad de elementos que tiene que gestionar, en ocasiones ha sido complicado obtener soluciones factibles a los problemas que se han ido encontrando a lo largo del desarrollo.

Adicionalmente, se ha tenido acceso físico al robot durante un tiempo muy limitado, por lo que es probable que, una vez el robot se integre en el laboratorio remoto, la utilización por parte de los alumnos dé pie a que se produzcan nuevos errores difíciles de detectar en el entorno de desarrollo.

Funcionalmente, el robot está abierto a numerosas mejoras, como la interpretación de los marcadores ArUco, cuya funcionalidad en este proyecto se limita a la mera detección, almacenamiento y renderizado de sus datos asociados.

Del mismo modo, podría estudiarse la posibilidad de sustituir la versión v1 de Kinect por su modelo avanzado, el modelo v2. No obstante, la línea de investigación abierta al respecto ha llegado a punto muerto al ser necesario un gran ancho de banda para la gestión de los datos que sólo puede ser satisfecho por un puerto USB3. Dado que la Raspberry Pi 3 Model B únicamente dispone de puertos USB2, es muy probable que existan importantes dificultades a la hora de integrarlo en el robot de forma satisfactoria, máxime cuando la propia versión v1 provoca ya importantes problemas de estabilidad y compatibilidad.

5. Referencias y bibliografía

[1] Forcén Carvalho, J.I & Chaos García, D. (2015). *Navegación de un robot autónomo basado en balizas y luz estructurada*. Universidad Nacional de Educación a Distancia y Universidad Complutense de Madrid.

[2] Rivière Visier, L., Chaos García, D. & De La Torre Cubillo, L. (2017). *Creación de mapas tridimensionales mediante robot teleoperado y dispositivo Kinect*. Universidad Nacional de Educación a Distancia y Universidad Complutense de Madrid.

[3] Abdulmajeed, W. & Mansoor, R. (2014). *Implementing Kinect Sensor for Building 3D Maps of Indoor Environments*. International Journal of Computer Applications. 86. 18-22. 10.5120/15005-3182.

[4] Sperl, P. (2018, 5 de Agosto). *Microsoft Kinect Depth Sensing*. Recuperado de https://www.anyline.com/wp-content/uploads/2017/10/peter_kinect_praesentation.pdf

[5] Burrus, N. (2018, 5 de agosto). *Kinect Calibration*. Recuperado de http://nicolas.burrus.name/index.php/Research/KinectCalibration

[6] Konolige, K. & Mihelich, P. (2018, 5 de agosto) *Technical description of Kinect Calibration*. Obtenido de http://wiki.ros.org/kinect_calibration/technical

[7] Raspberry Pi Foundation (2018, 7 de agosto). *Raspberry Pi 2 Specification*. Recuperado de https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[8] Raspberry Pi Foundation (2018, 7 de agosto). *Raspberry Pi 3 Specification*. Recuperado de https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[9] Ace Editor Group (2018, 7 de mayo). *Ajax.org Cloud9 Editor (Ace)*. Recuperado de https://ace.c9.io/

[10] JS Foundation (2018, 7 de mayo). *jQuery, write less, do more*. Recuperado de https://jquery.com/

[11] Otto, M. Thornton, J. (2018, 7 de mayo). *Bootstrap*. Recuperado de https://getbootstrap.com/

[12] Fonticons, Inc. (2018, 7 de mayo). Font Awesome. Recuperado de https://fontawesome.com/

[13] Decima, S. (2018, 15 de julio). Detect Element Resize. Recuperado de https://github.com/sdecima/javascript-detect-element-resize/

[14] Brannon, M et al. (2002). *ROTATION: A review of useful theorems involving proper orthogonal matrices referenced to three- dimensional physical space*. University of New Mexico.

[15] Tomasi, C. (2013). *Vector Representation of Rotations*. Duke University.

[16] Swanson, K. (2018, 20 de julio). *Aruco Marker World Coordinates*. Recuperado de https://stackoverflow.com/questions/51270649/aruco-marker-world-coordinates

[17] Raspberry Pi Foundation (2018, 28 de julio). *Raspbian Downloads*. Recuperado de https://www.raspberrypi.org/downloads/

[18] Davis, T. & Davis, J. (2018, 28 de julio). *Win32 Disk Imager*. Recuperado de https://sourceforge.net/projects/win32diskimager/

[19] Resin.io (2018, 28 de julio). *Etcher*. Obtenido de https://etcher.io/

[20] Sutas, A. et al (2018, 8 de julio). *GNU Octave instrument-control package*. Recuperado de https://octave.sourceforge.io/instrument-control/

[21] Chacón Sombría, J. (2018, 27 de julio). *py-ripserver*. Recuperado de https://github.com/jcsombria/py-ripserver

[22] García García, D. (2018, 1 de septiembre). *py-ripserver*. Recuperado de https://github.com/danielggarcia/py-ripserver

[23] Behnel S. et al. (2018, 15 de julio). *Cython*. Recuperado de https://github.com/cython/cython/wiki

[24] Jackson, L. (2018, 12 de julio). *mjpg-streamer*. Recuperado de https://github.com/jacksonliam/mjpg-streamer

[25] Blake, J. (2018, 20 de julio). *OpenKinect Project*. Obtenido de https://openkinect.org

[26] Standard C++ Foundation. (2018, 20 de julio). *Standard C++*. Recuperado de https://isocpp.org/

[27] OpenCV Team. (2018, 20 de julio). *Open Source Computer Vision Library*. Recuperado de https://opencv.org/

[28] Muñoz-Salinas, R. & Garrido-Jurado, S. (2018, 12 de agosto). *ArUco: a minimal library for Augmented Reality applications based on OpenCV*. Recuperado de https://www.uco.es/investiga/grupos/ava/node/26

[29] Esquembre, F. (2018, 8 de marzo). *Easy Java/Javascript Simulations*. Recuperado de http://www.um.es/fem/EjsWiki/?userlang=es

[30] Grinberg, M. (2018, 20 de julio). *How to build and run MJPG-Streamer on the Raspberry Pi*. Recuperado de https://blog.miguelgrinberg.com/post/how-to-build-and-run-mjpg-streamer-on-the-raspberry-pi

[31] Tanzilli, S. (2018, 20 de julio). *Using mjpeg streamer to stream videos over HTTP*. Recuperado de https://www.acmesystems.it/video_streaming

[32] Zucker, P. (2018, 18 de agosto). *Aruco in OpenCV*. Recuperado de http://www.philipzucker.com/aruco-in-opencv/

[33] Freeman, E. (2018, 18 de agosto). *OpenKinect Python and OpenCV*. Recuperado de http://euanfreeman.co.uk/openkinect-python-and-opencv/

[34] Hauptmech (2018, 16 de agosto). *Camera calibration using ChArUco and Python*. Recuperado de http://answers.opencv.org/question/98447/camera-calibration-using-charuco-and-python/

[35] Nullboundary (2018, 17 de agosto). *Charuco calibration*. Recuperado de https://github.com/nullboundary/CharucoCalibration

[36] Rahman, A. (2018, 17 de agosto). *Camera calibration.* Recuperado de https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html

[37] Charleux, L.(2018, 17 de agosto) *Camera calibration using CHARUCO*. Recuperado de https://mecaruco2.readthedocs.io/en/latest/notebooks_rst/Aruco/sandbox/ludovic/aruco_calibration_rotation.html

[38] Garrido, S. (2018, 18 de agosto). *Detection of CharUco Corners*. Recuperado de https://github.com/opencv/opencv_contrib/blob/3.1.0/modules/aruco/tutorials/charuco_detection/charuco_detection.markdown

[39] Rosebrock, A. (2018, 16 de agosto). *Ubuntu 16.04: How to install OpenCV*. Recuperado de https://www.pyimagesearch.com/2016/10/24/ubuntu-16-04-how-to-install-opencv/

[40] Khoshelham, K. & Elberink, S. (2018, 22 de julio). *Accuracy and Resolution of Kinect Depth Data for Indoor Mapping Applications* .Recuperado de https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3304120/

[41] Burrus, N. (2018, 20 de agosto). *Calibrating the depth and color camera*. Recuperado de http://nicolas.burrus.name/index.php/Research/KinectCalibration

[42] Roberson, W. (2018, 21 de agosto). *How do I make plot a 3D matrix as a scatter plot with color based on value?* Recuperado de https://es.mathworks.com/matlabcentral/answers/312615-how-do-i-make-plot-a-3d-matrix-as-a-scatter-plot-with-color-based-on-value

[43] Lai, K. et al. (2018, 21 de agosto). *RGB-D Object Dataset*. Recuperado de https://rgbd-dataset.cs.washington.edu/software.html

[44] Glira, P. (2018, 21 de agosto). *Point cloud tools for Matlab*. Recuperado de http://www.geo.tuwien.ac.at/downloads/pg/pctools/pctools.html

[45] Ryeng, R. (2018, 22 de agosto). *How do axis-angle rotation vectors work and how do they compare to rotation matrices?* Recuperado de https://stackoverflow.com/questions/32485772/how-do-axis-angle-rotation-vectors-work-and-how-do-they-compare-to-rotation-matr

[46] Kangalow (2018, 25 de agosto). *Install Kinect V2 using libfreenect2 on NVIDIA Jetson TK1*. Recuperado de https://www.jetsonhacks.com/2015/02/26/install-kinect-v2-using-libfreenect2-on-nvidia-jetson-tk1/

[47] Kalachev, O. (2018, 20 de agosto). *Aruco markers generator!*. Recuperado de http://chev.me/arucogen/

[48] Valencia, J., Montoya, A & Hernando, L. (2009). *Modelo cinemático de un robot móvil tipo diferencial y navegación a partir de la estimación odométrica*. Universidad Tecnológica de Pereira.

[49] Baker, M.J. (2018, 25 de agosto). *Maths – Axis angle*. Recuperado de https://www.euclideanspace.com/maths/geometry/rotations/axisAngle/index.htm

[50] 19Per9 (2018, 21 de agosto). *How to generate a 3D point cloud from depth image and color image acquired from Matlab*. Recuperado de https://stackoverflow.com/questions/36890562/how-to-generate-a-3d-point-cloud-from-depth-image-and-color-image-acquired-from

[51] Pomerleau, F. Colas, F. Siegwart, R. *A Review of Point Cloud Registration Algorithms for Mobile Robotics*. Foundations and Trends in Robotics, Now Publishers, 2015, 4 (1), pp.1–104. <10.1561/2300000035>. <hal-01178661>

6. Listado de siglas, definiciones y acrónimos

API: *Application Programming Interface*. Conjunto de métodos y funciones puestos a disposición de un programador para hacer uso de una biblioteca software.

ARM: *Advanced RISC Machine*. Arquitectura de procesadores de conjunto reducido de instrucciones (RISC) diseñado para minimizar el tamaño de sus componentes y orientado a dispositivos embebidos.

ArUco: *Augmented Reality library from the University of Córdoba*. Software de realidad aumentada basado la localización y orientación espacial de marcadores similares a códigos QR.

ASCII: *American Standard Code for Information Interchange*. Código basado en el alfabeto latino que hace uso de 7 bits para la codificación de caracteres.

CPU: *Central Process Unit*. Hardware encargado de interpretar las instrucciones de un programa informático mediante operaciones aritméticas, lógicas y de entrada/salida.

EjsS: *Easy Java/Javascript Simulations*. Software desarrollado por la Universidad de Murcia para diseñar simulaciones por ordenador.

GB: *GigaByte*. Unidad de almacenamiento equivalente a un millón de bytes.

HTML: *HyperText Markup Language*. Lenguaje de marcado estándar para el desarrollo de páginas y aplicaciones web.

ICP: *Iterative Closes Point*. Algoritmo utilizado para minimizar la diferencia entre dos nubes de puntos.

IDE: *Integrated Development Environment*. Programa de ordenador que proporciona un conjunto de herramientas que simplifican la codificación de otros programas mediante la inclusión de editores de código fuente, compiladores y comprobadores de formato.

JSON: *JavaScript Object Notation*. Formato de archivo utilizado para transmitir información en forma de listas y pares clave-valor.

Kinect: dispositivo de captura de imágenes RGB y de profundidad diseñado y comercializado por Microsoft.

LIDAR: *Laser Imaging Detection And Ranging*. Dispositivo capaz de determinar la distancia física a un obstáculo mediante el uso de un haz láser pulsado.

Middleware: software encargado de adaptar información obtenida desde un software y transmitirla a otro software en un formato distinto al enviado por el software emisor.

MP: *MegaPíxel*. Equivale a un millón de píxeles. Su valor es utilizado para determinar la resolución de un dispositivo de captura de imágenes como resultado del producto del número de filas por el número de columnas. Así, cuando se dice que una cámara tiene una resolución de 2.1 MP, significa que es capaz de alcanzar resoluciones de 1920x1080 = 2.073.600 píxeles = 2.1 MP.

PNG: *Portable Network Graphics*. Formato gráfico que soporta transparencia y compresión de datos sin perdida.

RAM: *Random Access Memory*. Memoria de acceso volátil empleada como memoria de trabajo en computadoras y otros dispositivos.

RIP: *Routing Information Protocol*. Forma de *middleware* encargada de tratar y reenviar información entre dos elementos software

ROS: *Robot Operating System*. Framework de desarrollo de software para robots consistente en un conjunto de bibliotecas que proporcionan soporte para las funciones más comunes presentes en dispositivos robóticos.

UNILabs: *University Network of Interactive Labs*. Plataforma interuniversitaria *online* encargada de proporcionar cursos y laboratorios remotos.

USB: *Universal Serial Port*. Bus de comunicaciones serie estándar, empleado para el intercambio de información entre dispositivos electrónicos.

Watchdog: mecanismo de seguridad encargado de monitorizar el estado de un elemento hardware o software y que tiene la capacidad de realizar un reinicio del mismo en caso de bloqueo.

Wi-Fi: tecnología de comunicación inalámbrica entre dispositivos electrónicos que cumple con los estándares 802.11 de redes inalámbricas de área local.

A. ANEXO: Instalación y configuración

Como paso previo a la instalación del sistema, será necesario instalar *Raspbian* en una tarjeta microSD de al menos 16GB.

Una vez instalada, se ejecutará *raspi-config* para configurar el sistema operativo, la red y otros elementos propios de cualquier Raspberry Pi.

El sistema está pensado para ser instalado y configurado de forma sencilla a través de un script de instalación (setup.sh), por lo que no requiere mayores esfuerzos ni complicaciones. Basta lanzarlo desde una Shell en una Raspbian con conexión a internet y esperar a que se realicen las siguientes acciones:

- Generación de la estructura de directorios.
- Creación del fichero swap adicional
- Actualización del sistema
- Reinstalación de Java
- Instalación de dependencias
- Descarga del py-ripserver
- Instalación de octave
- Compilación de instrument-control
- Descarga, compilación e instalación de OpenCV
- Instalación del código del robot
- Reinicio del dispositivo.

Algunas de las operaciones llevadas a cabo por el instalador pueden llevar horas, por lo que se aconseja ser paciente a la hora de realizar la configuración. Como alternativa, se puede grabar directamente el fichero robot.img.zip en una tarjeta microSD con aplicaciones como Etcher[19], dejando ya el sistema preparado para su ejecución.