



**Universidad de Valladolid**



**ESCUELA DE INGENIERÍAS  
INDUSTRIALES**

**UNIVERSIDAD DE VALLADOLID**

**ESCUELA DE INGENIERIAS INDUSTRIALES**

**Grado en Ingeniería en Electrónica Industrial y Automática**

**CONTROL DE UN ROBOT CARTESIANO  
USANDO EL SISTEMA OPERATIVO ROBÓTICO  
ROS2**

**Autor:**

**Vázquez Ramos, Álvaro**

**Tutor:**

**Herreros López, Alberto  
Departamento Ingeniería de  
Sistemas y Automática.**

**Valladolid, Agosto 2020.**

**Resumen:**

En este proyecto se describe el desarrollo de programas para hacer uso del controlador industrial AMC4030 en el entorno ROS2 para operar un robot cartesiano.

Se distingue el funcionamiento manual para el que se operará desde una interfaz gráfica que será necesario desarrollar empleada para tareas de comprobación de rutas y mantenimiento. En modo automático recibirá las coordenadas de un nodo de ROS2 externo, que determinará la posición deseada.

Finalmente se mostrará un sencillo ejemplo en un sistema de visión artificial, de forma que se la cámara la que determine la posición deseada.

**Palabras clave:**

ROS2, C++, Windows, control, paquete, nodo.

**Abstract:**

In this project is described the development of programs for operating the AMC4030 industrial controller in the ROS2 environment.

For the manual operation there will be developed a graphical interface, that will allow maintenance and checking routes. For the automatic mode, the coordinates will be received from an external ROS2 node, that will resolve the desired position.

Due to the flexibility of ROS2, the developed programs and packages will allow the robot to operate in any kind of systems and workspaces.

**Key words:**

ROS2, C++, Windows, control, package, node.

# INDICE

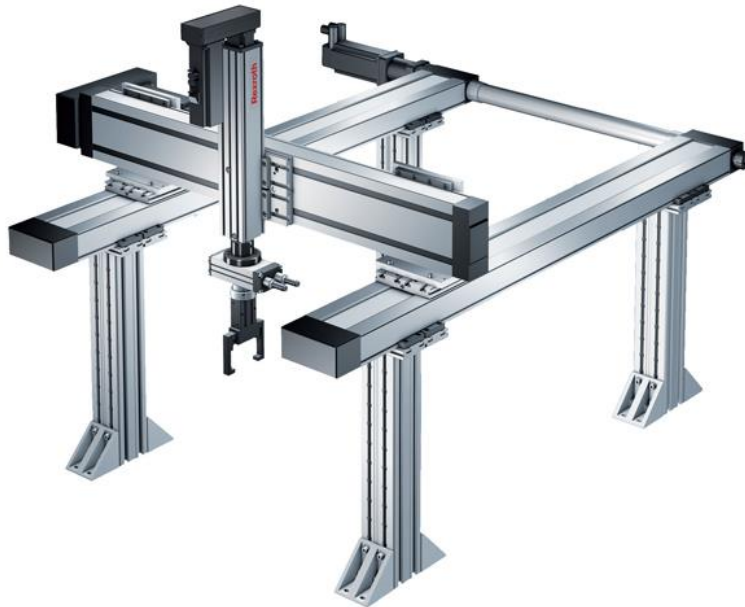
<b>1. Introducción</b>	<b>5</b>
<b>2. Descripción de herramientas y dispositivo.</b>	<b>7</b>
2.1. <i>Descripción hardware.</i>	7
2.1.1. FSL 120	7
2.1.2. Driver	8
2.1.3. AMC4030-3 axis controller	9
2.1.4. Finales de carrera.	10
2.1.5. Circuito de conexión.	11
2.1.6. Intel realsense SR300	11
2.2. <i>Configuración de parámetros del robot.</i>	12
2.3. <i>Herramientas software utilizadas.</i>	14
2.3.1. ROS2	14
2.3.2. C++/CLI	14
2.3.3. AMC4030 Software	15
2.3.4. Driver CH341SER	15
2.3.5. DLL Export Viewer	15
2.3.6. OpenCV	16
<b>3. ROS2 Introducción básica.</b>	<b>17</b>
3.1. <i>Origen de ROS y necesidad de ROS2.</i>	17
3.2. <i>Instalación de ROS2</i>	18
3.2.1. Requisitos y preparación previa.	19
3.2.2. Instalación de Visual Studio 2019	19
3.2.3. Instalación Chocolatey.	19
3.2.4. Instalación de Git	20
3.2.5. Instalación de ROS2 con Chocolatey	20
3.2.6. Creación de acceso directo (opcional).	20
3.3. <i>Configuración del entorno de ROS2</i>	21
3.4. <i>Conceptos básicos ROS2</i>	21
3.4.1. Nodo	21
3.4.2. Temas	22
3.4.3. Servicios.	23
3.4.4. Parámetros.	25
3.4.5. Acciones.	25
3.4.6. Ficheros de lanzamiento (launch files)	27
3.4.7. Creación de espacio de trabajo	27
3.4.8. Paquetes	27
3.4.9. Interfaces	29
3.4.10. ros2doctor	32
<b>4. Visión artificial</b>	<b>33</b>
4.1. <i>Imagen</i>	33
4.2. <i>Filtros y convolución</i>	34
4.3. <i>Transformada de Hough aplicada a circunferencias</i>	35
4.4. <i>Mapa de profundidad</i>	35

4.5.	<i>El modelo estenopeico.</i>	36
4.6.	<i>Distorsiones</i>	37
4.6.1.	<i>Distorsión radial.</i>	38
4.6.2.	<i>Distorsión tangencial</i>	38
<b>5.</b>	<b>Desarrollo del proyecto</b>	<b>40</b>
5.1.	<i>Descripción del problema</i>	40
5.2.	<i>Clase COM_API</i>	41
5.2.1.	<i>Carga dinámica de funciones.</i>	41
5.2.2.	<i>Descripción de la clase COM_API</i>	43
5.3.	<i>Interfaz gráfica.</i>	44
5.3.1.	<i>Introducción a la herramienta.</i>	44
5.3.2.	<i>Vista general de la interfaz gráfica.</i>	46
5.3.3.	<i>Detalles de implementación.</i>	49
5.4.	<i>Programa de control.</i>	52
5.5.	<i>Paquete de ROS2</i>	54
5.5.1.	<i>Paquete de interfaz</i>	54
5.5.2.	<i>Paquete cliente servidor.</i>	56
5.6.	<i>Sistema de visión</i>	61
5.6.1.	<i>Obtención del mapa de profundidad</i>	62
5.6.2.	<i>Obtención de un punto en coordenadas de la cámara</i>	64
5.6.3.	<i>Obtención de un punto en coordenadas del robot</i>	66
5.6.4.	<i>Detección de la bola</i>	66
5.6.5.	<i>Ejecución del programa de movimiento.</i>	67
<b>6.</b>	<b>Prueba de funcionamiento</b>	<b>68</b>
6.1.	<i>Lanzamiento del nodo cliente.</i>	68
6.2.	<i>Lanzamiento del nodo servidor.</i>	68
6.3.	<i>Aplicación de comandos de ROS2.</i>	69
6.4.	<i>Prueba del sistema de visión artificial</i>	71
<b>7.</b>	<b>Conclusión y proyectos futuros</b>	<b>73</b>
7.1.	<i>Conclusión</i>	73
7.2.	<i>Proyectos futuros</i>	73
<b>8.</b>	<b>Bibliografía:</b>	<b>75</b>
<b>9.</b>	<b>Anexos</b>	<b>76</b>
9.1.	<i>Instalación</i>	76
9.2.	<i>Índice de figuras</i>	78
9.3.	<i>Código de funciones auxiliares en ejemplo de visión.</i>	81

## 1. Introducción

En este TFG se describe el proceso de desarrollo del software de control para hacer uso del controlador industrial AMC4030 aplicado a un robot cartesiano de articulaciones prismáticas en el sistema operativo robótico Ros2.

Un robot cartesiano consta de una serie de ejes que se desplazan cada uno de ellos de forma lineal y entre los diferentes ejes forman ángulos rectos. Es el tipo de robot más simple, gracias a lo cual existe una gran cantidad de fabricantes y tamaños (capaces de operar con cargas desde gramos a cientos de kilogramos).



*Figura 1. Robot cartesiano de 3 grados de libertad. Fuente: <https://base.imgix.net/>*

Las aplicaciones industriales de los robots cartesianos son inmensas como por ejemplo almacenaje, clasificación de productos, soldadura, incluyendo el emergente campo de impresión 3D.

A lo largo del proyecto se desarrollarán los programas de control que permitan operar un robot cartesiano de forma manual o automática por medio del controlador industrial AMC4030, que será quien envíe instrucciones a los actuadores.

En modo manual será un operario el que, por medio de una interfaz gráfica, introducirá la posición deseada. Ésta se desarrollará como un formulario de Windows en lenguaje C++/CLI en el entorno .NET.

En modo automático se hará uso del entorno ROS2 para realizar la comunicación. De esta forma el código realizado será fácilmente reutilizable para todo tipo de aplicaciones. Por ello en primer lugar se realizará una breve descripción de ROS2, que permitirá conocer los conceptos principales y comprender de una forma rápida las bases del entorno. Se hará especial énfasis en los comandos principales y de mayor utilidad, al finalizar la descripción el lector se familiarizará con términos y características de ROS2.

Posteriormente se describirá de forma pormenorizada tanto el sistema físico como cada una de las herramientas de desarrollo. Se hará especial énfasis en el tipo de robot utilizado.

Finalmente se realizará una prueba mediante comandos del sistema de ROS2 y un sencillo ejemplo de visión artificial que consistirá en la localización tridimensional de cuerpos esféricos. Para ello se empleará el SR300 de Intel. Consta de una cámara RGB y un sensor de profundidad basado en luz estructurada. El capítulo de visión artificial expone los conceptos principales para la comprensión del ejemplo.

La siguiente figura ilustra el sistema que se desarrollará, así como la interacción entre cada una de las partes:

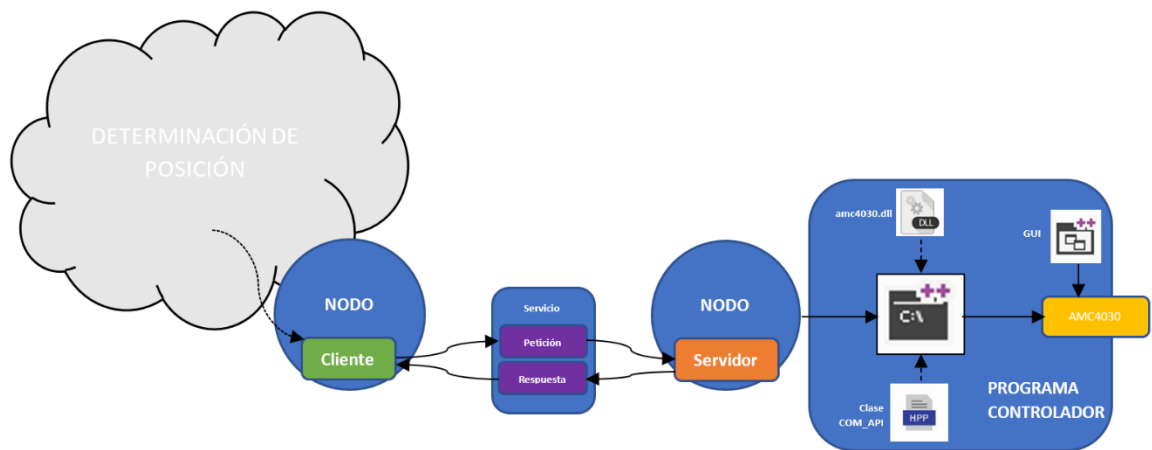


Figura 2. Diagrama de relación entre las partes del proyecto.

Como se observa en la figura un nodo servidor atenderá las peticiones de movimiento de un nodo cliente, llamado por un sistema externo (en los ejemplos utilizados en primer lugar será por medio de comandos, finalmente será la cámara la que determine la posición). A continuación el nodo servidor enviará las instrucciones al programa controlador. La interfaz gráfica envía instrucciones al controlador de forma directa.

Al finalizar se habrá logrado la puesta en marcha de un robot cartesiano para su inmediata integración en un sistema de ROS2, permitiendo al robot alcanzar la posición requerida e indicada por el nodo cliente, así como un control manual haciendo uso de la interfaz gráfica.

## 2. Descripción de herramientas y dispositivo.

En este apartado se realiza una breve descripción del dispositivo físico y herramientas empleadas.

### 2.1. Descripción hardware.

Se comentarán los diferentes componentes que componen el sistema físico.

#### 2.1.1. FSL 120

Guía lineal que emplea un husillo de bolas para el posicionamiento. En la imagen inferior se puede observar el mecanismo y la propia guía, el módulo es del fabricante Fuyu Technology.

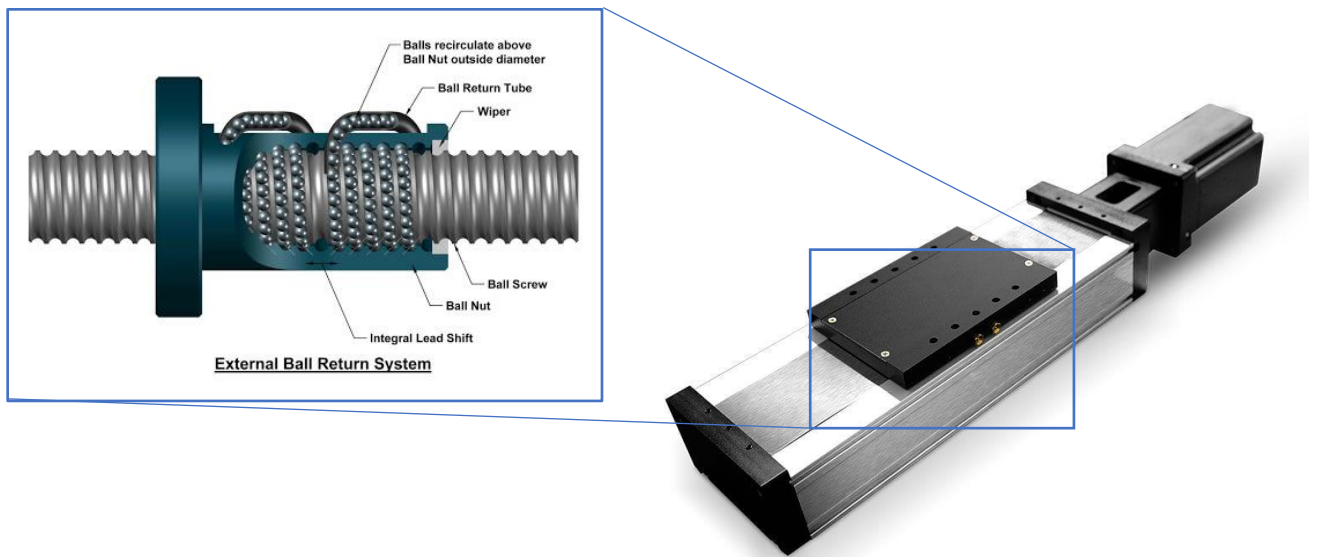


Figura 3. Mecanismo interno FSL 120. Fuentes: [fuyumotion.com, barnesballscrew.com]

Como se observa en la figura superior un husillo de bolas se encarga de la transformación del movimiento rotatorio de los motores en lineal, desplazándose así la plataforma.

Soporta una carga máxima horizontal de 100 kg con una precisión de posicionamiento de 0.03 mm. Para más información consultar la página web del fabricante [5].

Su diseño facilita el montaje de sistemas robotizados de articulaciones prismáticas de varios ejes. A continuación, se muestran las seis posibles configuraciones que recomiendan. Para la configuración se opta por la de la sexta imagen.

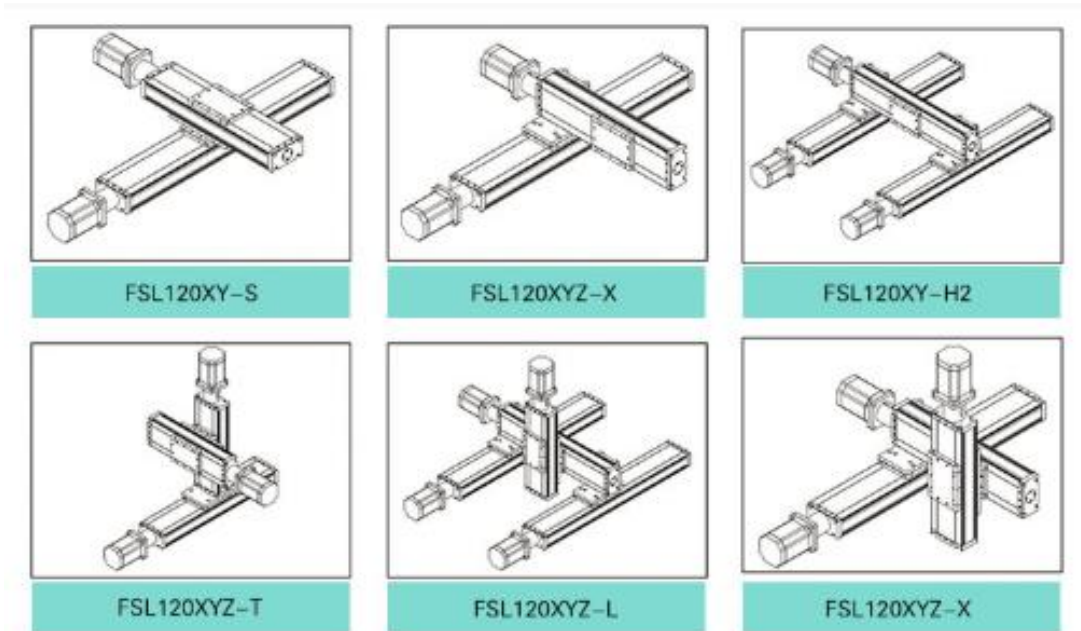


Figura 4. Configuraciones FSL 120. Fuente: [fuyumotion.com]

La configuración de las guías lineales FSL120XYZ-X ilustra nuestro robot cartesiano. Consta de por tanto de tres grados de libertad cuyo espacio de trabajo es un cubo de dimensiones iguales a los carriles de las guías lineales. No permite orientación en el punto alcanzado.

Los actuadores consisten en un motor paso a paso Nema 34H2A6840. Se trata de un motor bipolar, el esquema se ilustra en la figura inferior.

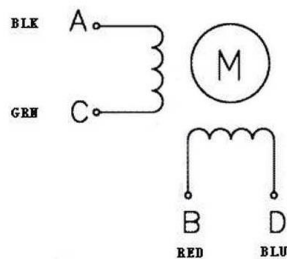


Figura 5. Esquema de motor bipolar.

El controlador aplica una secuencia de pulsos a los devanados de forma que por inducción hagan girar el rotor. Dado que la corriente nominal es de 4 A, es muy elevada para el controlador AMC4030, éste último operará por medio de drivers.

La ventaja del empleo de motores bipolares se traduce en un mayor par que los motores unipolares del mismo tamaño, no obstante, requieren de circuitos de control más complejos.

### 2.1.2. Driver

El modelo del driver es FMDT220A48NOM. Admite modificar la corriente desde un mínimo de 2.5 A hasta 6.2 A. La subdivisión del driver puede variarse desde 200 hasta 40000 pulsos/revolución. La subdivisión es el número de pulsos necesarios para completar una vuelta



completa del rotor. Los interruptores del driver D1-D4 permiten seleccionar la corriente y de D5-D8 la subdivisión de acuerdo a la tabla inferior.

D1	D2	D3	D4	$I_m(A)$	D5	D6	D7	D8	s/r
OFF	OFF	OFF	OFF	2.5	ON	ON	ON	ON	200
OFF	OFF	OFF	ON	2.7	ON	ON	ON	OFF	400
OFF	OFF	ON	OFF	2.9	ON	ON	OFF	ON	800
OFF	OFF	ON	ON	3	ON	ON	OFF	OFF	1000
OFF	ON	OFF	OFF	3.5	ON	OFF	ON	ON	1600
OFF	ON	OFF	ON	3.7	ON	OFF	ON	OFF	2000
OFF	ON	ON	OFF	3.9	ON	OFF	OFF	ON	3200
OFF	ON	ON	ON	4.2	ON	OFF	OFF	OFF	4000
ON	OFF	OFF	OFF	4.4	OFF	ON	ON	ON	5000
ON	OFF	OFF	ON	4.6	OFF	ON	ON	OFF	6800
ON	OFF	ON	OFF	4.8	OFF	ON	OFF	ON	8000
ON	OFF	ON	ON	5	OFF	ON	OFF	OFF	10000
ON	ON	OFF	OFF	5.3	OFF	OFF	ON	ON	12800
ON	ON	OFF	ON	5.7	OFF	OFF	ON	OFF	16000
ON	ON	ON	OFF	6	OFF	OFF	OFF	ON	20000
ON	ON	ON	ON	6.2	OFF	OFF	OFF	OFF	40000

La configuración requerida es la que se muestra en la tabla, logrando una corriente de 3.9 A y una subdivisión de 5000 s/r.

### 2.1.3. AMC4030-3 axis controller

Es un controlador de propósito general que permite el control de hasta 3 motores (servo motor o paso a paso).

En la imagen siguiente puede verse la parte frontal del controlador, que permite asociar los puertos con mayor facilidad.

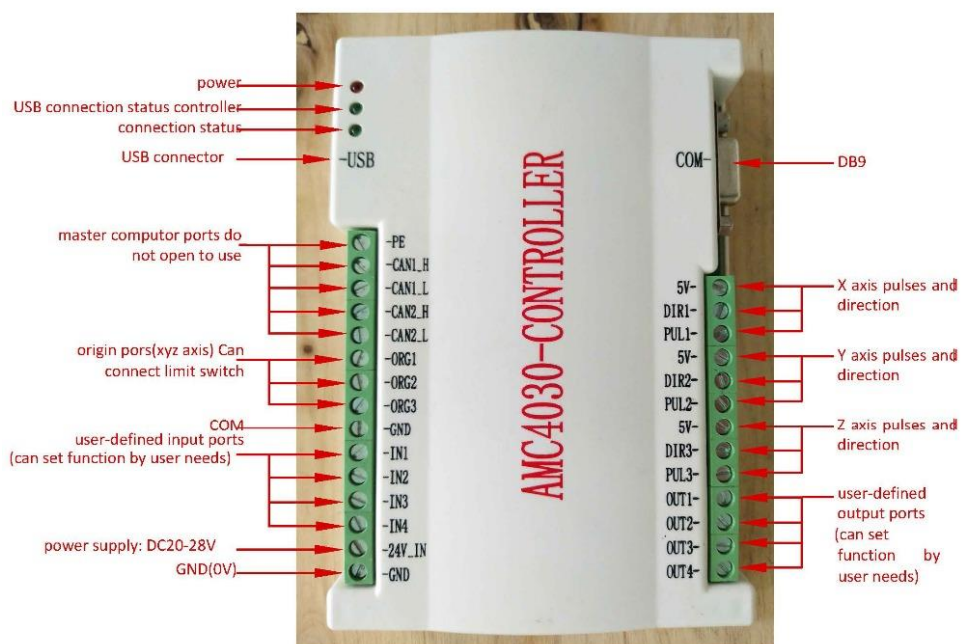


Figura 6. Frontal del controlador AMC4030. Fuente [Fuyumotion.com]

En la siguiente tabla se muestra la descripción de cada puerto:

Signal	Descriptions	Signal	Descriptions
PE	CAN Signal Ground	5V	Power output DC5V
CAN1_H	CAN1 High-level signal	DIR1	X-axis direction
CAN1_L	CAN1 Low-level signal	PUL1	X-axis pulse
CAN2_H	CAN2 High-level signal	5V	Power output DC5V
CAN2_L	CAN2 Low-level signal	DIR2	Y-axis direction
ORG1	X Original position signal	PUL2	Y-axis pulse
ORG2	Y-axis Original position signal	5V	Power output DC5V
ORG3	Z-axis Original position signal	DIR3	Z-axis direction
GND	Original point signal ground	PUL3	Z-axis pulse
IN1	General input port 1	OUT1	General output port 1
IN2	General input port 2	OUT2	General output port 2
IN3	General input port 3	OUT3	General output port 3
IN4	General input port 4	OUT4	General output port 4
24V_IN	24V Power input	USB	MINI USB, connect PC
GND	24V Power ground	COM	COM, connect HMI

La interacción con los drivers se hará a través de dos señales. Una señal de pulsos que se enviará al motor (“pulse”) y otra de dirección (“dir”) que determinará el sentido de la rotación.

Se requerirá de tres finales de carrera que determinen el origen de cada eje. Pueden usarse los puertos de entrada genéricos para definir funciones de usuario, no se usarán en la conexión. Los puertos de salida pueden ser usados para notificar situaciones de la máquina (ej: robot en parada de emergencia, robot en modo manual, etc). El controlador admite conexión USB para construir un sistema jerárquico de control o por otro lado emplear una HMI para la puesta en marcha de un sistema de control aislado.

#### 2.1.4. Finales de carrera.

Empleados para la detección del origen de cada eje. Serán sensores de proximidad F3N-18TN05-N R2M. La tensión de alimentación de 10 – 30 VDC con una corriente de salida de 200mA. Son de tipo NPN luego su conexión con el controlador es la siguiente:

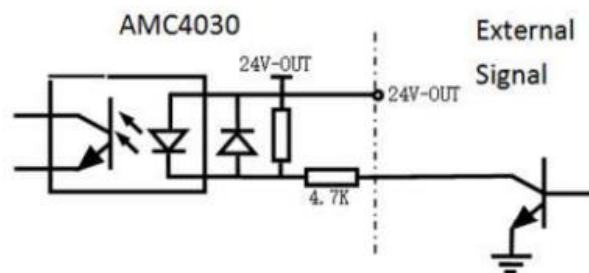


Figura 7. Conexión de final de carrera al controlador. Fuente [Fuyumotion.com]

### 2.1.5. Circuito de conexión.

En la figura inferior se muestra el circuito de conexión empleando 3 motores paso a paso bipolares (actuadores de cada eje) con sus finales de carrera correspondientes.

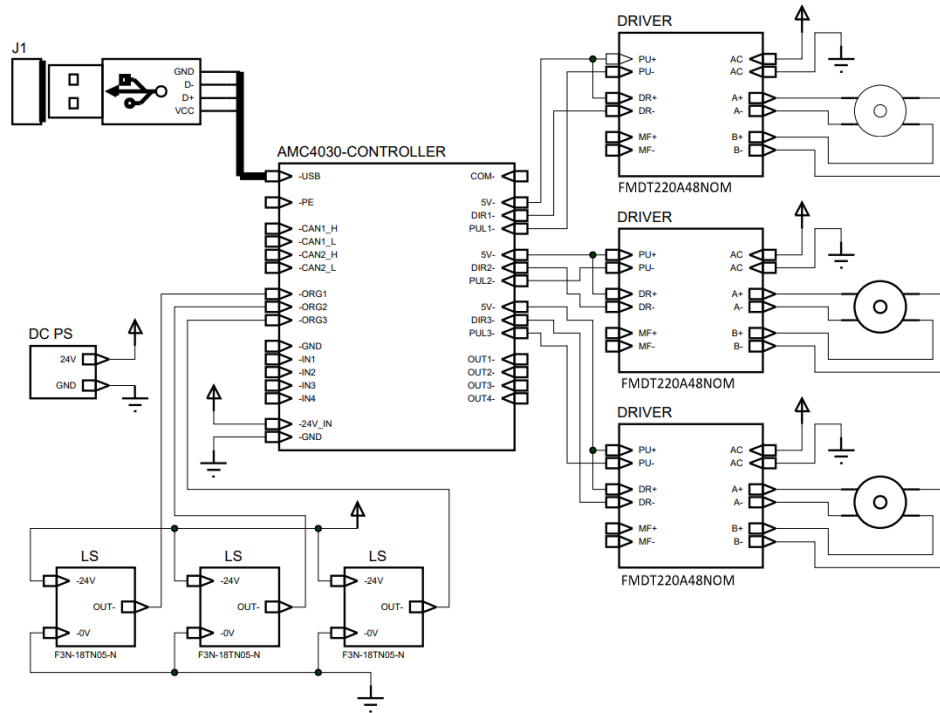


Figura 8. Esquema de conexión.

### 2.1.6. Intel realsense SR300

Cámara con sensor de profundidad de Intel. Para la detección de la profundidad emplea luz estructurada.

La luz estructurada consiste en la emisión de patrones de luz (véase como líneas paralelas negras y blancas de anchura variable) para luego observar la interacción del patrón con el objeto del cual se quiere conocer la distancia.

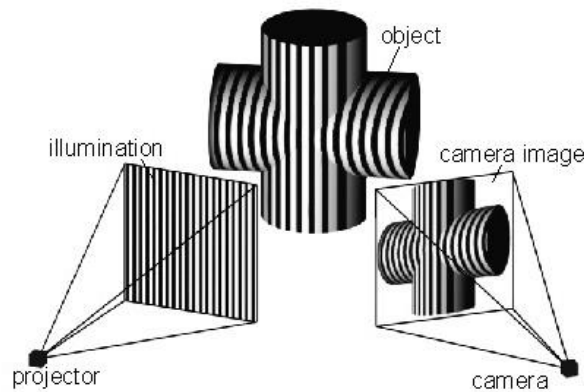


Figura 9. Esquema luz estructurada. Fuente [15]

La luz estructurada tiene una gran precisión, pero posee un alcance reducido con lo que puede no resultar muy conveniente su uso en ambientes industriales. Existen otras alternativas interesantes como las imagen estéreo.

La figura inferior muestra los componentes internos del dispositivo, en ella se pueden observar tanto el proyector como la cámara de infrarrojos, así como la cámara de color.

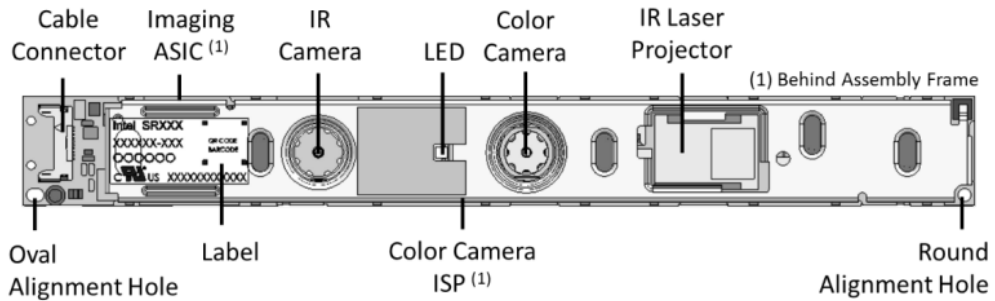


Figura 10. Componentes del dispositivo Intel real sense SR300. Fuente: [\[https://www.mouser.com/pdfdocs/intel\\_realsense\\_camera\\_sr300.pdf\]](https://www.mouser.com/pdfdocs/intel_realsense_camera_sr300.pdf)

Intel proporciona una librería que permite un rápido desarrollo de aplicaciones que hagan uso de sus sensores de profundidad, llamada realsense.

La librería distingue dispositivos, sensores y secuencias.

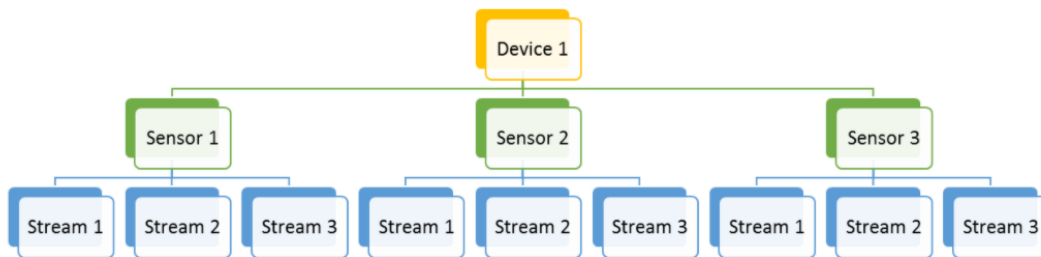


Figura 11. Arquitectura de la librería librealsense. Fuente [13]

El dispositivo hace referencia a la propia cámara. Una cámara puede tener diversos sensores, uno RGB, uno de luz estructurada para detectar profundidad (como en el caso del SR300), estéreo, inercial, etc. Finalmente cada sensor puede tener diversas secuencias de fotogramas que determinan aspectos como la resolución, FPS y el formato, se identifican de forma única por un número de perfil. Así un sensor puede tener cientos de secuencias distintas.

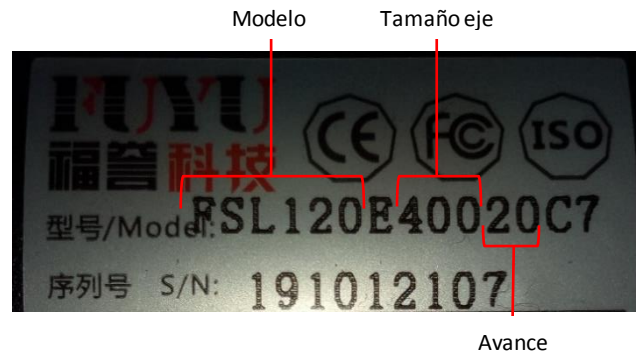
## 2.2. Configuración de parámetros del robot.

La configuración de los parámetros del robot incluye:

- Distancia de cada movimiento: depende de la geometría del tornillo. Puede observarse en el número del modelo de la guía lineal. Se especifica en mm.

- Subdivisión del driver: número de pulsos necesarios para completar una vuelta completa del motor. Es un parámetro crítico para el control. El fabricante recomienda un valor de 5000 para el controlador empleado.
- Velocidad de retorno a origen: velocidad empleada cuando el robot recibe la instrucción "Home". Una velocidad lenta hará el proceso tedioso, pero garantizará el reconocimiento de los sensores. Se especifica en mm/s.
- Dirección de retorno a origen: dependiendo de la colocación de los sensores (al principio o al final del carril), la dirección de búsqueda de los sensores será antihoraria (N) o horaria (P).
- Carrera de la guía lineal: posición máxima alcanzable. Esta información puede obtenerse también del número del modelo. Se especifica en mm.
- Offset de origen: desplazamiento positivo respecto al punto en el que se encuentran los finales de carrera. El offset garantiza que al realizar dos instrucciones "home" consecutivas el robot alcance posiciones no deseadas que puedan comprometer su correcto funcionamiento.

En la imagen inferior se muestra la designación de uno de los motores de los ejes.



*Figura 12. Designación de uno de los motores.*

Considerando una velocidad de retorno a origen de 20 mm/s con sentido negativo, un offset de 5 mm y los tamaños de los demás ejes la configuración de los parámetros del robot son los que se muestran a continuación:

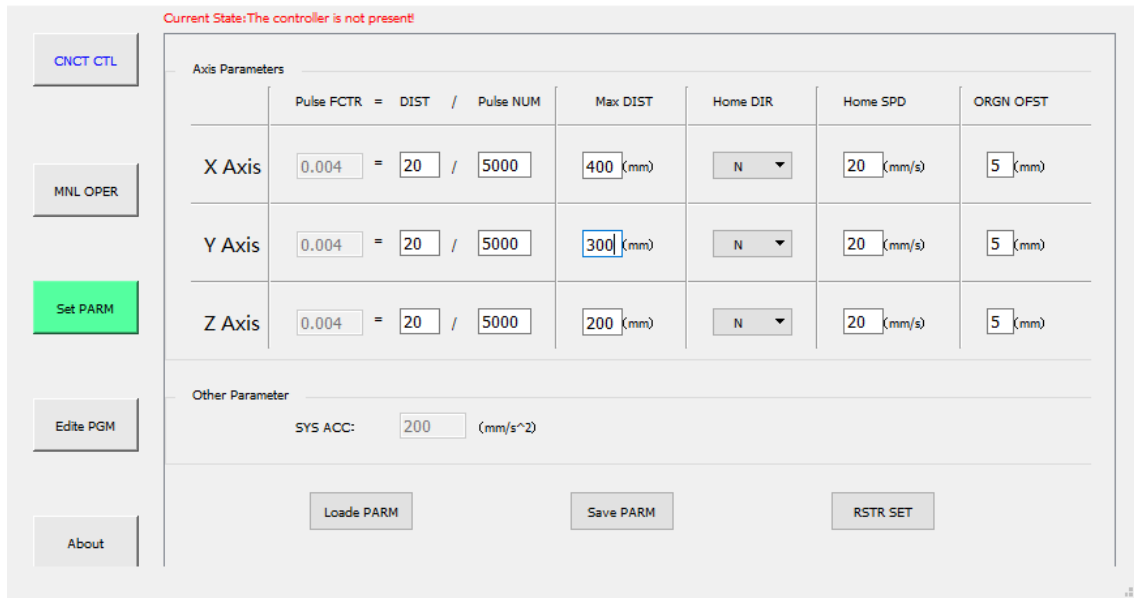


Figura 13. Captura de software AMC4030. Fuente [Fuyumotion.com]

Puede verse que para el robot considerado las dimensiones de los ejes x, y, z son 400, 300, 200 mm respectivamente. La aceleración no se puede modificar y es de 200 mm/s<sup>2</sup>.

## 2.3. Herramientas software utilizadas.

### 2.3.1. ROS2

Como ya se comentó anteriormente el objetivo del desarrollo es la incorporación a un sistema de ROS2. ROS2 emplea ament\_cmake para la compilación de paquetes C/C++.

### 2.3.2. C++/CLI

C++/CLI es una herramienta de interoperabilidad entre .NET y Win32 que permitirá la creación de la interfaz gráfica en Visual Studio de forma sencilla empleando los formularios de Windows (Windows forms). La herramienta está siendo poco a poco sustituida en este campo por el uso de C#.

Ésta permitirá hacer uso de la plantilla de proyecto "CLR Console Application", se muestra su selección en VisualStudio en la imagen inferior.

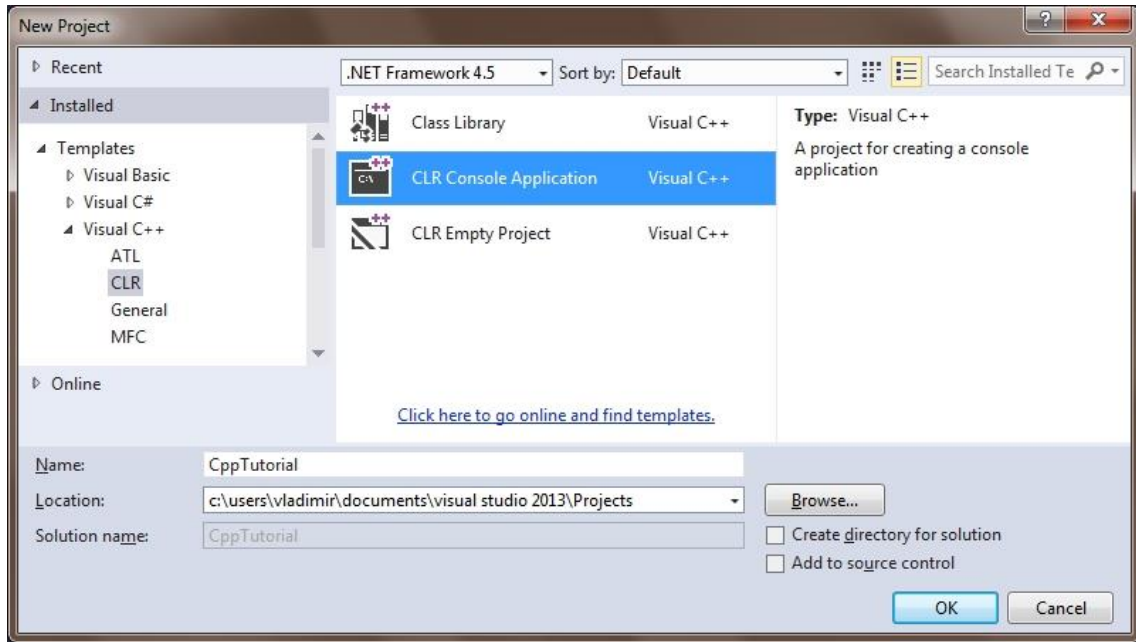


Figura 14. Captura de VisualStudio, plantilla CLR Console Application.

### 2.3.3. AMC4030 Software

Software que proporciona el fabricante, nos permitirá configurar los parámetros del robot (carrera efectiva, velocidad de búsqueda de origen, avance, etc) y otro tipo de funcionalidades.

### 2.3.4. Driver CH341SER

Driver para conexión del controlador al PC mediante puerto USB. Puede descargarse de la página web del fabricante [5]. Para su instalación simplemente ejecute el programa setup.exe con el dispositivo conectado. Puede encontrarse en la sección de la página manual, en el archivo comprimido AMC4030 software, en la carpeta CH340 Driver (USB driver)\_XP\_WIN7.

### 2.3.5. DLL Export Viewer

Programa para visualización de las funciones exportadas y su dirección en memoria de una biblioteca de vínculos dinámicos (DLL). Nos servirá para el análisis del fichero AMC4030.dll, librería que proporciona el fabricante del controlador.

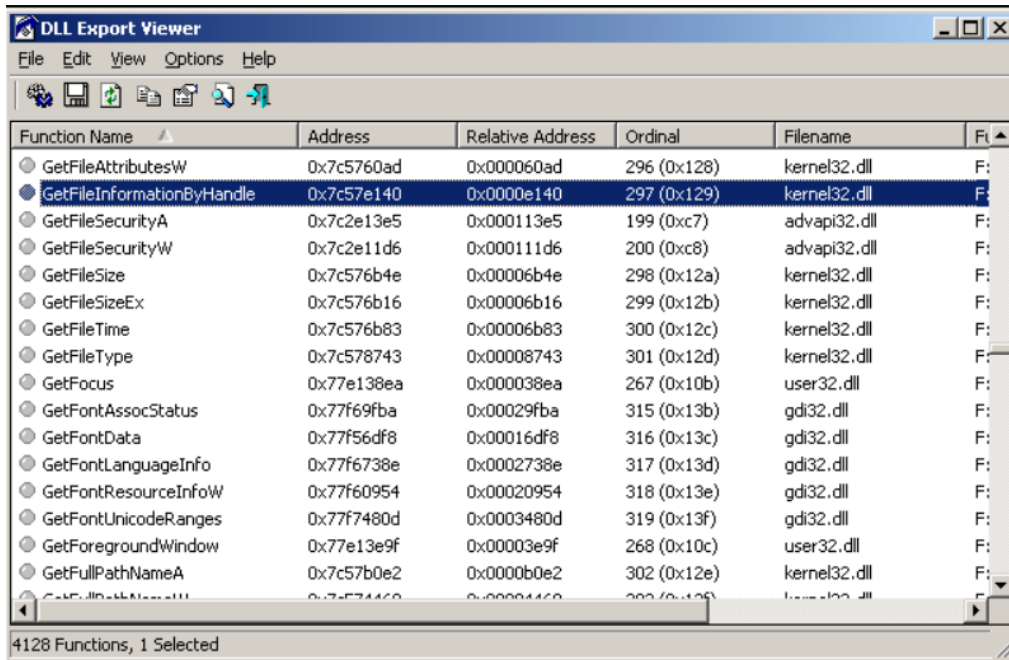


Figura 15. Captura de DLL Export Viewer. Fuente: [https://www.nirsoft.net]

### 2.3.6. OpenCV

Biblioteca empleada en el ejemplo de visión artificial. Su lanzamiento se remonta a 1999 por Gary Bradski.

La librería está escrita en C y C++ y puede usarse en Linux, Windows y Mac OS X. Hay interfaces realizados para Python, Java, MATLAB y otros lenguajes, incluso es posible portar la librería a Android e IOS para el desarrollo de aplicaciones móviles.

Uno de los objetivos principales de OpenCV es proporcionar una infraestructura de visión artificial de uso simple para ayudar al desarrollo de aplicaciones de visión rápidamente.

La biblioteca se ha usado en gran número de campos y aplicaciones como imágenes médicas, seguridad, interfaces de usuario, calibración de cámaras, visión estéreo y robótica.



### 3. ROS2 Introducción básica.

#### 3.1. Origen de ROS y necesidad de ROS2.

ROS (*robot operating system*) nace como un entorno de desarrollo para el robot PR2 de Willow Garage [10], empresa dedicada a la creación de software de código abierto para aplicaciones usadas en robots personales (en aquel entonces).

Pese a que su objetivo principal era servir de soporte y herramienta para la creación de proyectos con el PR2, conocían de antemano que podría resultar útil su uso en otros robots y proyectos, se crearon por tanto diferentes niveles de abstracción que permitirían la reutilización del software desarrollado.

Algunas de las características que definieron el entorno son:

- Empleo de un único robot.
- Sin requisitos de tiempo real.
- Excelente conectividad a la red (cableado o por proximidad inalámbrica de gran ancho de banda).
- Aplicaciones de investigación.
- Máxima flexibilidad. Soporta gran variedad de lenguajes de programación modernos. Hay librerías implementadas en Python, C++, Lisp y alguna experimental en Java y Lua.
- Apropiado en grandes sistemas y procesos de desarrollo.

Con el avance de la tecnología, ROS comenzó a usarse en multitud de aplicaciones y situaciones para las que en un principio no había sido desarrollado o no se habían contemplado.

- Equipos de múltiples robots: se ha logrado desarrollar sistemas de múltiples robots hoy en día con ROS, pero no hay una forma estándar.
- Uso en sistemas embebidos.
- Uso en sistemas de tiempo real: soporte de control en tiempo real.
- Redes no ideales: uso en redes de baja fiabilidad.
- Entornos de producción: pese a que sea la plataforma de elección en laboratorios de investigación, se busca la evolución para su uso en aplicaciones reales.

ROS2 surge como una oportunidad para adaptar el entorno a estas nuevas aplicaciones, no obstante, se insiste en su coexistencia con ROS, y el soporte de la interacción entre ambos sistemas, permitiendo a ROS la compilación de librerías desarrolladas en ROS2.

Se aprovecha además para renovar y rediseñar nuevas APIs, mejorando la experiencia del usuario manteniendo los conceptos base de ROS. Las principales diferencias entre ambos se recogen en la siguiente tabla [11]:

	ROS	ROS2
<b>Plataforma</b>	Probado en Ubuntu, mantenido en otras distribuciones de Linux y también en OS X	Esta probado y con soporte en Ubuntu Xenial, OS X El Capitan y Windows 10
<b>C++ API</b>	C++03	Principalmente C++11, se plantea el uso de
<b>Python API</b>	Python 2	>= Python 3.5
<b>Middleware</b>	Formato de serialización propio	Basadas en el estándar DDS
<b>Tipos de tiempo y duración</b>	Se definen en las librerías cliente, definidos en C++ y Python	Los tipos se definen como mensajes y son por tanto consistentes entre lenguajes
<b>Componentes con ciclo de vida</b>	En ros cada nodo tiene su función principal (" <i>main</i> ") propia.	El ciclo de vida puede ser usado por herramientas como como roslaunch para crear un sistema compuesto por varios componentes de forma determinista.
<b>Modelo de hilos</b>	En Ros el desarrollador solo puede elegir entre una ejecución de hilo único o una ejecución multihilo.	En ROS2 están disponibles modelos de ejecución más granular y se pueden implementar ejecuciones personalizadas fácilmente.
<b>Nodos múltiples</b>	En ros no es posible crear más de un nodo en un proceso.	En ROS2 es posible crear múltiples nodos en un proceso.
<b>roslaunch</b>	En ROS los archivos de roslaunch se definen en XML con capacidades limitadas	En ROS2 los ficheros de lanzamiento se escriben en Python lo cual permite usar una lógica más compleja como por ejemplo el uso de condicionales.

### 3.2. Instalación de ROS2

La instalación de ROS2 puede realizarse en Linux, macOS y Windows. Actualmente hay tres versiones activas Dashing Diademata, Eloquent Elusor y Foxy Fitzroy.

Versión	Fecha de lanzamiento	Soporte
ROS2 Dashing Diademata	Mayo 2019	Mayo 2021
ROS2 Eloquent Elusor	Noviembre 2019	Noviembre 2020
ROS2 Foxy Fitzroy	Junio 2020	Mayo 2023

A continuación, se describe el proceso de instalación de la distribución y plataforma usadas, Eloquent en la plataforma Windows.

### 3.2.1. Requisitos y preparación previa.

- Sistema operativo *Windows 10 Desktop* o *Windows 10 IoT Enterprise x64*: asegúrese de tener *Powershell* instalado e incluido en el *path* del sistema.
- Cree un directorio en el disco C con el nombre *opt* y asegúrese de tener al menos 10 Gb de espacio libre. Excluya el directorio de los escáneres de virus en tiempo real. Si únicamente tiene *Windows Defender* instalado, puede hacerlo en *Seguridad de Windows* -> *Protección contra virus y contra amenazas* -> *Configuración de antivirus y protección contra amenazas* -> *Administrar la configuración* -> *Exclusiones* -> *Agregar o quitar exclusiones*.

### 3.2.2. Instalación de Visual Studio 2019

Debe incluir además la opción de *“Desktop development with C++”*. Si ya lo tenía instalado previamente, puede modificar la instalación con *Visual Studio installer*.

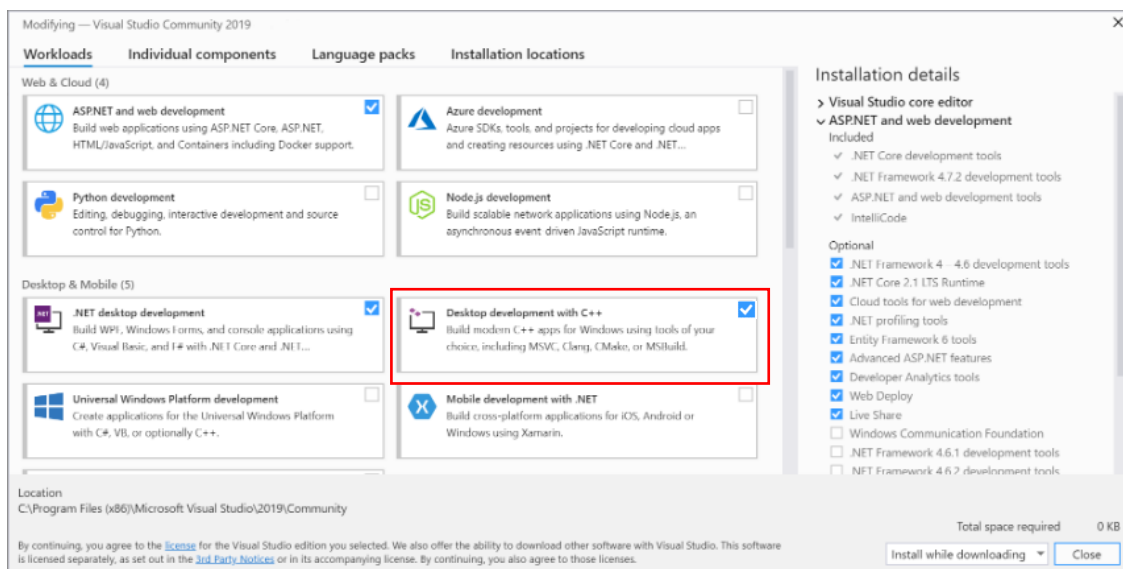


Figura 16. Captura de Visual Studio installer.

Se recomienda hacer uso de la consola de comandos de *Visual Studio “x64 Native Tools Command Prompt for VS 2019”* para la ejecución de comandos de ROS2.

### 3.2.3. Instalación Chocolatey.

Chocolatey se emplea para la instalación de librerías y herramientas para la ejecución y compilación de proyectos de ROS. Ejecute *“x64 Native Tools Command Prompt for VS 2019”* como administrador y copie el siguiente código:

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -InputFormat None -ExecutionPolicy Bypass -Command "iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

### 3.2.4. Instalación de Git

Vuelva a abrir la consola de comandos de visual studio, ejecute los siguientes comandos:

```
choco upgrade git -y
```

```
git -version
```

### 3.2.5. Instalación de ROS2 con Chocolatey

Copie y ejecute los siguientes comandos nuevamente.

```
mkdir c:\opt\chocolatey
set ChocolateyInstall=c:\opt\chocolatey
choco source add -n=ros-win -s="https://aka.ms/ros/public" --priority=1
choco upgrade ros-eloquent-desktop -y --execution-timeout=0 --pre
```

Una vez acabada la instalación podrá observar en C:\opt los directorios que se muestran en la captura inferior:






Nombre	Fecha de modificación	Tipo	Tamaño
 chocolatey	02/07/2020 20:04	Carpeta de archivos	
 python37amd64	02/07/2020 20:08	Carpeta de archivos	
 ros	02/07/2020 20:00	Carpeta de archivos	
 rosdeps	02/07/2020 20:02	Carpeta de archivos	
 vcpkg	02/07/2020 20:00	Carpeta de archivos	

Figura 17. Carpetas en el directorio de instalación.

### 3.2.6. Creación de acceso directo (opcional).

Para hacer uso de ROS2 en Windows es necesario llamar al bat de inicialización desde cada consola de comandos (se comenta en el apartado siguiente). Puede ahorrarse este paso creando un acceso directo con permisos de administrador. Para ello copie la siguiente ruta en el acceso directo, considerando que la versión de *Visual Studio* instalada es *Community* (modifíquese en función de la versión instalada).

```
C:\Windows\System32\cmd.exe /k "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\Tools\VsDevCmd.bat" -arch=amd64 -host_arch=amd64&& set ChocolateyInstall=c:\opt\chocolatey&& c:\opt\ros\eloquent\x64\setup.bat
```

Finalmente haga click derecho en el acceso directo -> propiedades -> opciones avanzadas -> Ejecutar como administrador.

### 3.3. Configuración del entorno de ROS2

Para hacer uso de los comandos de ROS2 necesita ejecutar el archivo bat de inicialización del entorno de ROS2. De haber creado el acceso directo, simplemente ábralo y se ahorrará este paso, de no ser así abra la consola de comandos de visual studio y copie el siguiente comando:

```
call C:\opt\ros\eloquent\x64\local_setup.bat
```

El comando variará en función del directorio de instalación de ROS. Si ha seguido los pasos anteriores debería ser válido.

Puede comprobar que tiene acceso a los comandos de ROS2, simplemente ejecutando ROS2, obteniendo el resultado que se muestra en la ilustración siguiente:

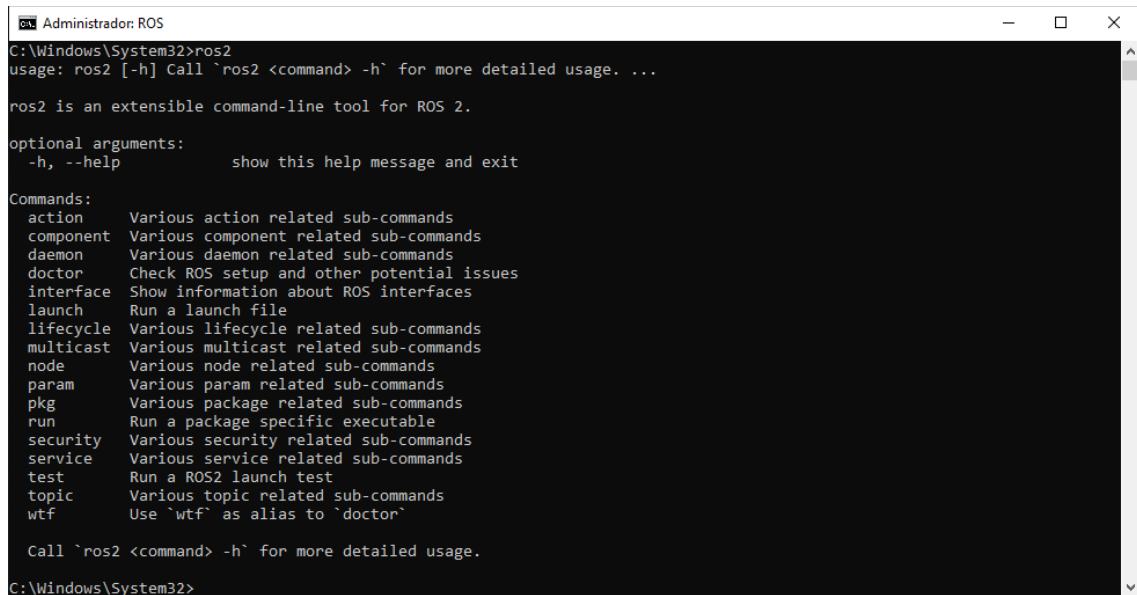


Figura 18. Ejecución del comando ros2.

### 3.4. Conceptos básicos ROS2

A continuación, se muestran los conceptos básicos que permiten definir un sistema de ROS2. La guía [3] emplea un simulador con el que el aprendizaje de ROS2 es mucho más ameno e interesante, los siguientes apartados tratan de mostrar comandos que encontré de mayor utilidad y al mismo tiempo dar una rápida descripción del entorno.

#### 3.4.1. Nodo

Un nodo es un archivo ejecutable con capacidad de comunicación con otros nodos. Cada nodo en ROS debería ser de propósito único (e.g. control de la velocidad de rotación de los motores, control de sensor de proximidad, etc). Cada nodo se comunica enviando y recibiendo información de otros nodos por medio de temas (*topic*), servicios, acciones o parámetros.

Para ejecutar un programa de un paquete concreto el comando es el siguiente:

```
ros2 run <package_name> <executable_name>
```

Otros comandos de gran utilidad son,

```
ros2 node list
```

que muestra todos los nodos actualmente en ejecución y

```
ros2 node info <node_name>
```

Que accede a la información de un nodo concreto (lista de subscriptores, publicadores, servicios y acciones que interactúan con el nodo).

### 3.4.2. Temas

En ROS2 los sistemas complejos se dividen en múltiples nodos que realizan tareas simples. Los temas actúan como bus que permiten el intercambio de información entre dos nodos. Un nodo puede publicar información o estar suscrito a varios temas (comunicación multipunto).

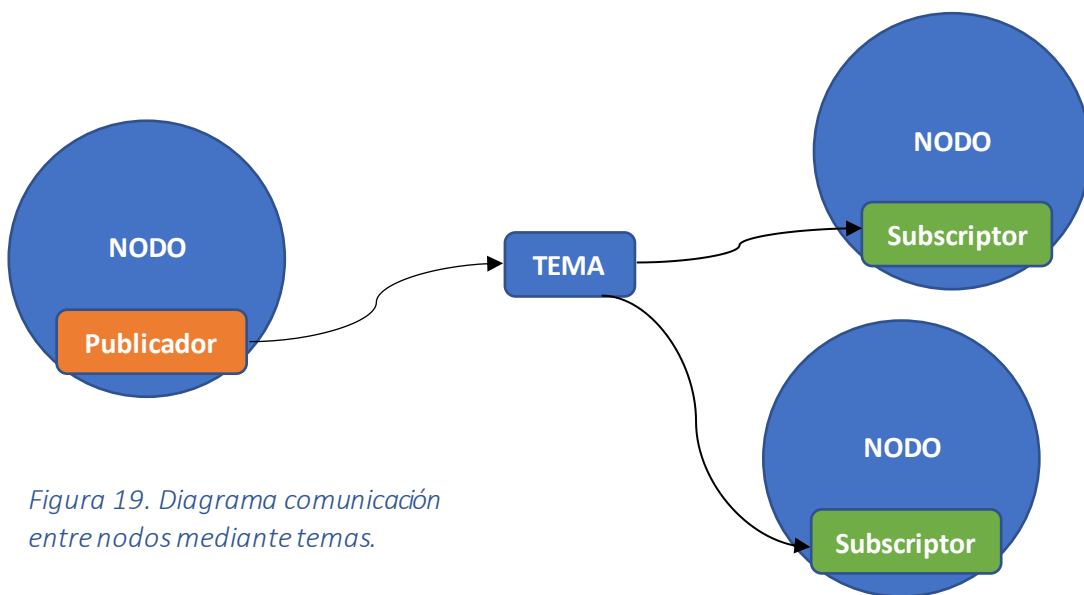


Figura 19. Diagrama comunicación entre nodos mediante temas.

En la ilustración superior puede observarse que el intercambio de información entre tres nodos mediante temas. Uno de los nodos publica información que es recibida por nodos subscriptores. Puede establecerse, por tanto, una comunicación multipunto unidireccional.

En ROS2 se elimina la necesidad del *roscore* de ROS1, en la que se establecía una arquitectura de maestro esclavo. La estructura DDS conforma y define una arquitectura distribuida la cual supone [12]:

- Comunicación entre iguales (*peer-to-peer*).
- No hay punto de fallo único (tolerancia a fallos).
- Calidad de servicio configurable para cada caso de uso (QoS).

En la comunicación mediante temas la información se transmite por medio de mensajes.

Entre los comandos útiles cabe citar:

```
ros2 topic list [-t]
```

Que muestra todos los temas activos en el sistema en el momento de la ejecución del comando, el parámetro opcional -t sirve para que además se incluya el tipo de los temas.

```
ros2 topic echo <topic_name>
```

Muestra en la consola todos los mensajes que pasan por el tema.

```
ros2 topic info <topic_name>
```

Devuelve el tipo del mensaje, así como el número de nodos que publican o están suscritos a ese tema. Es importante recalcar que para establecer la comunicación entre nodos han de compartir el mismo tipo de mensaje.

```
ros2 interface show <msg_type>
```

Muestra la estructura de un tipo de mensaje. Se analizará más detenidamente en el apartado de Interfaces.

```
ros2 topic pub [--once or --rate [time]] <topic_name> <msg_type> '<args >'
```

Conocido el tipo de mensajes, puede enviarse información a un nodo a través de la consola de comandos. Mediante el flag once se publicará un único mensaje, con --rate se publicarán cada *time* segundos de forma continua.

### 3.4.3. Servicios.

Los servicios son la segunda forma de comunicación entre nodos en ROS2. Nuevamente puede implicar a varios nodos. En la comunicación mediante servicios un nodo actúa como servidor (único) y los demás como clientes. Los nodos clientes realizan peticiones que debe atender el nodo servidor. La información de los servicios consta por tanto de los argumentos de petición y de la respuesta.

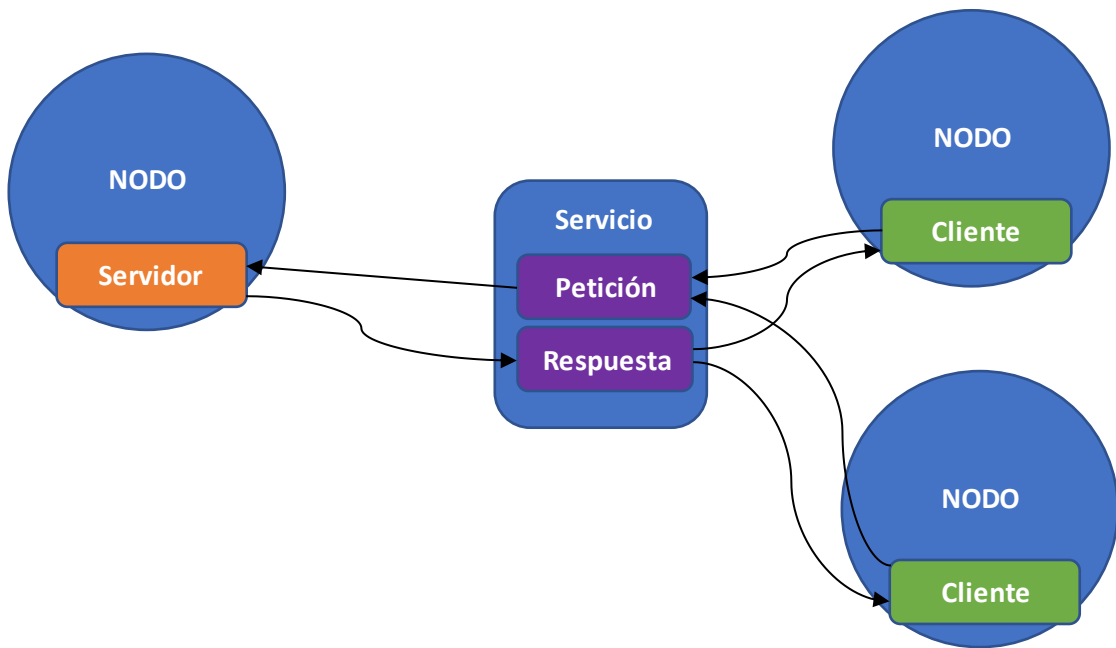


Figura 20. Diagrama de comunicación mediante servicios.

La transmisión de información sí que es ahora entre dos nodos. Cada cliente recibe una respuesta de acuerdo con la petición realizada por parte del servidor.

Los comandos para tratar con servicios son similares a los empleados con los temas, así:

```
ros2 service list [-t]
```

Muestra todos los servicios activos.

```
ros2 service type <service_name>
```

Muestra el tipo de servicio del servicio "service\_name".

```
ros2 service find <type_name>
```

Muestra todos los servicios activos de tipo <type\_name>

```
ros2 interface show <type_name>
```

El comando "interface show" puede emplearse tanto para tipos de mensaje o servicio para conocer su estructura.

```
ros2 service call <service_name> <service_type> <arguments>
```



Una vez conocido el tipo de servicio y la estructura podemos llamar al servicio desde la consola de comandos con “*ros2 service call*”.

#### 3.4.4. Parámetros.

Los parámetros son los valores de configuración de cada nodo. Pueden ser de tipo entero, coma flotante, booleano, cadenas de caracteres y listas. Permiten además reconfiguración dinámica.

Algunos comandos útiles son:

```
ros2 param list
```

Muestra los parámetros de cada uno de los nodos activos en ese momento.

```
ros2 param get <node_name> <parameter_name>
```

Devuelve el valor del parámetro “*parameter\_name*” del nodo “*node\_name*”.

```
ros2 param set <node_name> <parameter_name> <value>
```

Establece el valor del parámetro “*parameter\_name*” del nodo “*node\_name*” en “*value*”. Si la modificación se ha realizado con éxito se observará el mensaje en la consola.

```
Set parameter successful
```

```
ros2 param dump <node_name>
```

Permite almacenar en un fichero los parámetros de un nodo.

```
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>  
>
```

Ejecuta un nodo con los parámetros del fichero “*file\_name*”.

#### 3.4.5. Acciones.

El tercer y último tipo de comunicación en ROS2. Está pensado para tareas de gran tiempo de ejecución. Poseen una funcionalidad similar a los servicios, salvo que las acciones pueden ser

canceladas en tiempo de ejecución. Además, poseen realimentación a diferencia de los servicios que son de respuesta única. Las acciones se componen de:

- Un objetivo: el resultado que se desea obtener al finalizar la acción.
- Realimentación: información que transmite el nodo que actúa como servidor durante la acción.
- Resultado: resultado realmente obtenido al finalizar la acción.

Las acciones se construyen a partir de servicios y temas. Nuevamente la comunicación mediante acciones puede incluir a varios nodos.

En el diagrama inferior representa la comunicación entre dos nodos por medio de acciones. El número indica el orden en que se envía el servicio (s) o mensaje(m). En el transcurso de la acción se generan N mensajes de realimentación.

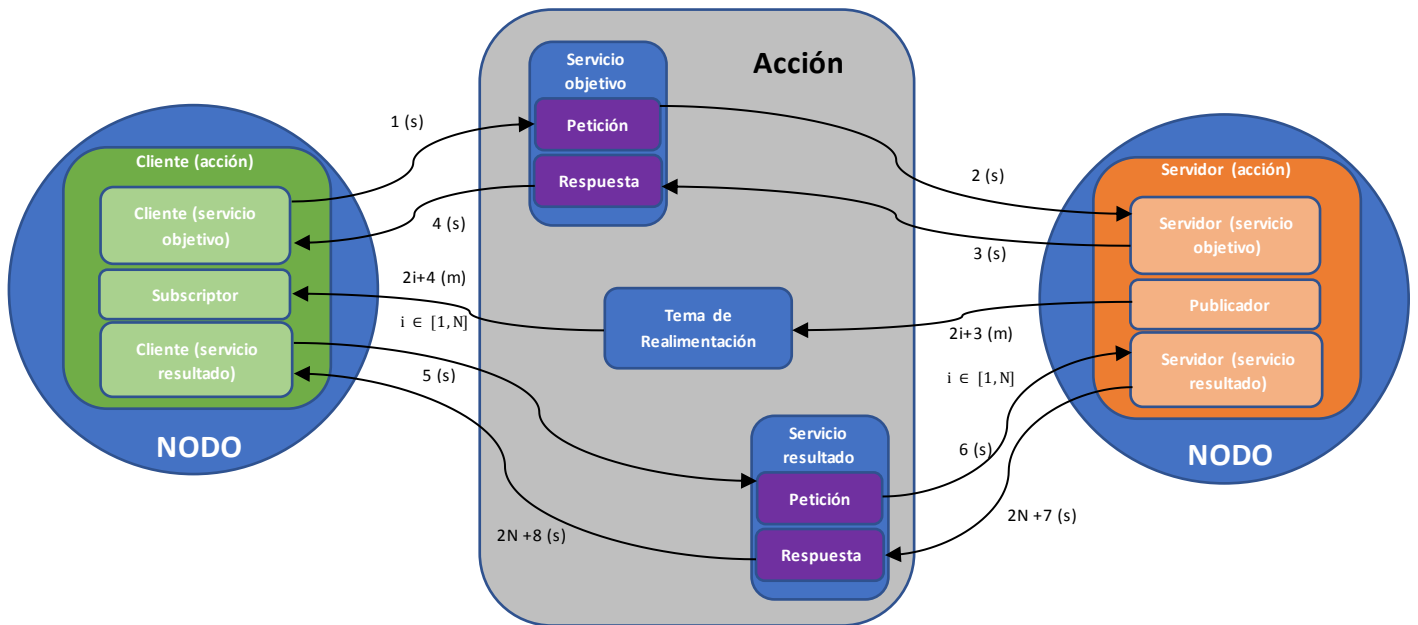


Figura 21. Diagrama de comunicación por medio de acciones.

Nuevamente los comandos vistos anteriormente son aplicables a las acciones.

```
ros2 action list [-t]
```

Muestra el conjunto de acciones entre nodos en ese momento. El flag opcional -t incluye además el tipo de la acción.

```
ros2 action info <action_name>
```

Muestra el tipo de la acción, además del número de nodos clientes y servidores con sus nombres.

```
ros2 interface show turtlesim/action/RotateAbsolute.action
```

Muestra la estructura de la acción. Se comentará detenidamente en el apartado de Interfaces.

```
ros2 action send_goal <action_name> <action_type> <values>
```

Envía el objetivo “*values*” al servidor de “*action\_name*” siendo el tipo de la acción “*action\_type*”.

### 3.4.6. Ficheros de lanzamiento (launch files)

En sistemas complejos con gran número de nodos resulta tedioso tener que ejecutar o crear cada nodo del sistema mediante el comando “*run*”. Los ficheros de lanzamiento permiten ejecutar varios nodos mediante un único comando, incluyendo además sus parámetros de configuración.

Deben estar en un directorio denominado “*launch*” y a diferencia del código de los nodos, únicamente puede ser escrito en Python.

El comando para ejecutar los ficheros de lanzamiento es:

```
ros2 launch <launch_file_name>
```

### 3.4.7. Creación de espacio de trabajo

El espacio de trabajo es un directorio que contiene paquetes de ROS2. Generalmente los paquetes se incluyen en el subdirectorio *src*. Desde el directorio del espacio de trabajo se ejecuta el comando de compilación como se verá más adelante. Una vez ejecutado se creará el bat de inicialización (“*install.bat*”), el cual mediante la instrucción *call* nos permitirá hacer uso de los paquetes instalados.

### 3.4.8. Paquetes

Un paquete contiene el código de ROS2. Se emplean para la organización del código, que permita su instalación o compartirlo con otros. De esta forma se podrá compilar y usar de forma sencilla. Se describirán los paquetes de CMake empleados para la compilación de código en lenguaje C++.

#### a. Estructura

Un paquete requiere como mínimo de dos ficheros:

- *Package.xml*: contiene metadatos del paquete.
- *CMakeLists.txt*: describe cómo compilar el código de dentro del paquete.

### b. Creación.

El comando de creación de paquetes es el siguiente:

```
ros2 pkg create --build-type ament_cmake <package_name>
```

Debe ejecutarse desde dentro del subdirectorio `src` del espacio de trabajo.

Tras su ejecución aparecerá en la carpeta `src` un directorio con el nombre del paquete que contendrá a su vez otra carpeta `src` donde deberá ir el código de los nodos. La carpeta *include* deberá contener los ficheros de declaración que comparten varios ficheros de código (`.h`, `.hpp`). Finalmente encontrará los dos ficheros anteriormente comentados *“package.xml”* y *“CMakeLists.txt”*. Aunque se analizarán en mayor profundidad a la hora de desarrollar el paquete de control, se comentará brevemente el fichero *“package.xml”*. Tomamos como referencia el ejemplo de la documentación de ROS2 [3].

```
<?xml version="1.0"?>
<?xml-model
  href="http://download.ros.org/schema/package_format3.xsd"
  schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_package</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

En él se puede observar que se ha creado un paquete de nombre `my_package` y es la versión 0.

Debajo puede incluirse una descripción del paquete, así como la persona que le da soporte y la licencia.

Entre las etiquetas de *“<buildtool\_depend>”* se encuentra la herramienta necesaria para la compilación del paquete (en este caso es un paquete de CMake. *“Ament\_cmake”* es el sistema

de compilación en ROS2 para paquetes basados en CMake. Añade funcionalidades y hace más sencilla la compilación de paquetes de ROS2.

Si el paquete que va a crearse depende de otros, simplemente puede añadirse el nombre del paquete del que depende entre las etiquetas `<depend>package_name</depend>`.

### c. Compilación.

Una vez creado puede compilarse mediante el comando:

```
colcon build --merge-install
```

Este comando compilará todos los paquetes del espacio de trabajo. Si se desea compilar únicamente uno de los paquetes puede emplearse:

```
colcon build --packages-select <package_name>
```

Que compilará únicamente el paquete *"package\_name"*.

Tras ejecutar el comando se crean tres directorios, build, install y log desde donde se ejecuta el comando. En el directorio de log se encuentra información acerca del proceso de compilación, puede ser útil en caso de que el proceso falle y se trate de encontrar los errores de todo tipo (e.g sintaxis en código, fallo en dependencias, fallo en búsqueda de ficheros, etc).

En el directorio install se encuentra el fichero de inicialización del paquete (*"setup.bat"*). Tras llamarlo desde la consola de comandos, permitirá ejecutar y hacer uso de los nodos y comandos definidos en el paquete.

## 3.4.9. Interfaces

Hasta ahora se han comentado los tipos de comunicación que existen entre nodos en ROS2, no obstante, se desconoce la estructura de los mensajes enviados.

En ROS2 se emplea un lenguaje simplificado de descripción, el lenguaje de descripción de interfaz (IDL). Este lenguaje permite la generación automática del código de la interfaz en el lenguaje de programación que haga uso de esta interfaz.

En este apartado se comentarán los tipos de datos que soportan, así como la estructura para temas, servicios y acciones.

### a. Estructura y tipos de datos

El componente principal de un mensaje son los campos. La declaración de un campo requiere de un tipo y un nombre. Un ejemplo de declaración sería:

```
bool my_bool  
wstring my_wstring
```

Los tipos soportados y su equivalencia con los lenguajes C++ y Python se muestran en la siguiente tabla, nótese como cada tipo puede ser empleado para la creación de cadenas o vectores.

Type name	C++	Python
<i>bool</i>	bool	builtins.bool
<i>byte</i>	uint8_t	builtins.bytes*
<i>char</i>	char	builtins.str*
<i>float32</i>	float	builtins.float*
<i>float64</i>	double	builtins.float*
<i>int8</i>	int8_t	builtins.int*
<i>uint8</i>	uint8_t	builtins.int*
<i>int16</i>	int16_t	builtins.int*
<i>uint16</i>	uint16_t	builtins.int*
<i>int32</i>	int32_t	builtins.int*
<i>uint32</i>	uint32_t	builtins.int*
<i>int64</i>	int64_t	builtins.int*
<i>uint64</i>	uint64_t	builtins.int*
<i>string</i>	std::string	builtins.str
<i>wstring</i>	std::u16string	builtins.str
<i>static array</i>	std::array<T, N>	builtins.list*
<i>unbounded dynamic array</i>	std::vector	builtins.list
<i>bounded dynamic array</i>	custom_class<T, N>	builtins.list*
<i>bounded string</i>	std::string	builtins.str*

#### b. Propiedades de la declaración de campos.

Se admite la especificación del número máximo de elementos en la cadena o vector. Por ejemplo:

```
int32[] array_ilimitado_enteros
int32[5] array_cinco_enteros
int32[<=5] array_hasta_cinco_enteros
string cadena_ilimitada_caracteres
string<=10 cadena_hasta_diez_caracteres
string[<=5] array_hasta_cinco_cadenas_ilimitadas
string<=10[] array_ilimitado_de_cadenas_hasta_diez_caracteres
```

Se permite la especificación de valores por defecto:

```
uint8 x 42
uint64[] samples [100, 150, 200, 130, 78]
string last_name "García"
```

Las cadenas de caracteres deben ser definidas empleando comillas o comillas dobles y sin barras invertidas (\).

La declaración de constantes se especifica con el símbolo =, además el nombre debe estar en mayúsculas.

```
int8 X=5
string[] FRUTAS=["Manzana","Pera","Plátano","Melocotón"]
bool VERDAD=true
int32 DEFAULT 321
```

Pueden además especificarse constantes con valores por defecto omitiendo el símbolo igual como se muestra en el último DEFAULT.

Los comentarios emplean el símbolo '#'.

### c. Tipos de interfaces.

Los tipos de interfaces disponibles en ROS2 son:

- **Mensajes:** es el tipo de interfaz más simple y empleado por los temas. Poseen extensión .msg y en los paquetes deben introducirse en un directorio del mismo nombre. Consisten en un fichero de declaración de campos, luego el ejemplo anterior puede constituir un mensaje válido.
- **Servicios:** es el tipo de interfaz empleado por los servicios. Tienen extensión srv y si pertenece a un paquete debe incluirse en un directorio srv/. Los servicios constan de dos partes una parte en la que se declaran los campos de petición y otra de respuesta, separados por tres guiones "---". Un ejemplo de servicio puede ser el siguiente.

```
#campos de petición
uint32 id_emisor 0
string peticion
---
#campos de respuesta
uint32 id_receptor 47
string respuesta
```

- **Acciones:** el interfaz empleado por acciones tiene extensión action y al igual que los anteriores debe incluirse en un fichero del mismo nombre. Las acciones las forman la estructura del objetivo, la estructura del resultado y finalmente la estructura de la realimentación. Un ejemplo de acción puede verse a continuación.

```
#campos de objetivo
float32 theta
---
#campos de resultado
float32 delta
---
#campos de realimentación
float32 alpha
```

### 3.4.10. ros2doctor

Con la distribución Eloquent se incluyó una nueva herramienta, `ros2doctor`. Se emplea para el análisis de sistemas de ROS2, comprobando aspectos como plataforma, versión, red, entorno emitiendo avisos acerca de posibles errores y el porqué de los mismos.

Para hacer uso de la herramienta, en un nuevo terminal ejecute el comando:

```
ros2 doctor
```

Si la configuración de ROS2 es correcta verá aparecer el mensaje.

```
All <n> checks passed
```

Si por el contrario está usando una distribución inestable o no probada, verá aparecer el mensaje:

```
UserWarning: Distribution <distro> is not fully supported or tested. To get more consistent features, download a stable version at https://index.ros.org/doc/ros2/Installation/
```

La herramienta reconoce fallos en la estructura de comunicación de un sistema. Así por ejemplo de haber algún nodo publicando y sin suscriptor, aparecerá en la terminal el aviso correspondiente.

Si se desea una información de la herramienta más detallada, puede incluirse en el comando de invocación el argumento `--report`. Al hacerlo se recibirá la información categorizada en:

- Configuración de red.
- Información de plataforma.
- Información acerca de middleware.
- Información de ros2
- Lista de temas.



## 4. Visión artificial

En este apartado se definen aspectos y conceptos que permiten comprender y analizar el sistema de visión desarrollado, para poder interpretar el desarrollo posterior de este.

La visión artificial consiste en la transformación de información obtenida de una cámara en una decisión o una nueva representación. Cada transformación se realiza para alcanzar un objetivo particular. Es un campo en auge y con gran número de aplicaciones como inspección automática de procesos, identificación de tareas, control de procesos y robots industriales, detección de eventos en sistema de vigilancia, modelado de entornos, navegación, organización de la información, en el campo de la medicina...

En primer lugar se analizará la imagen, forma en que se recibe la información de la cámara. Una vez definida se comentarán los filtros, una de las herramientas básicas de procesamiento de imágenes y la transformada de Hough empleada en el sistema real. A continuación se definirán los mapas de profundidad, sin entrar en las técnicas de obtención de estos. Finalmente se analizará la cámara y el modelo estenopeico que permite la interpretación y proyección de un punto de la imagen a un punto tridimensional real.

### 4.1. Imagen

Las imágenes son matrices de píxeles, siendo cada píxel comúnmente la composición de los colores rojo, verde y azul en caso de que el campo de color sea el RGB, el más común.

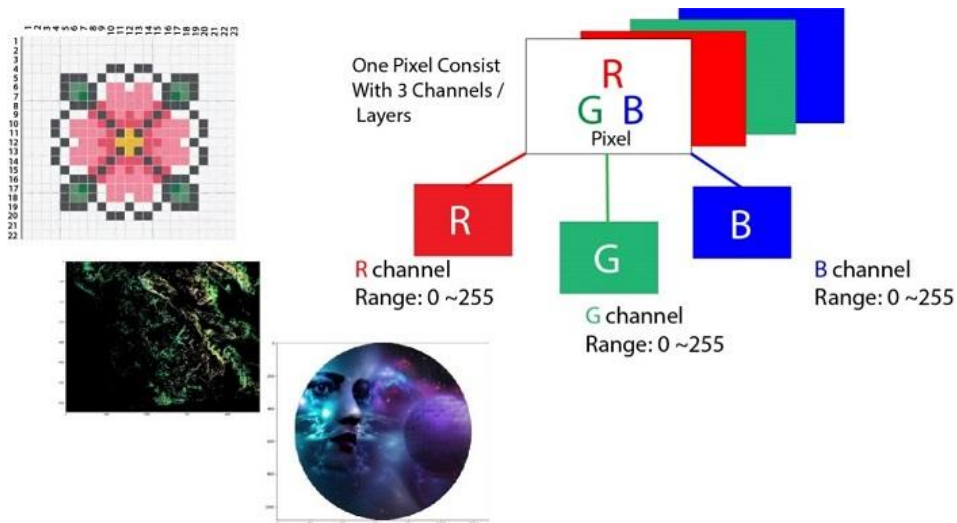


Figura 22. Definición de imagen

Así pues en cada píxel se registra la intensidad de cada uno de los colores, formando por tanto un array tridimensional que se compone de tres matrices para cada color.

Existen muchos otros tipos de imágenes. Por ejemplo una imagen en escala de grises consta de una única matriz en la que cada píxel registra el nivel de brillo en él. Otro ejemplo es una imagen binaria en la que los píxeles pueden tomar solo dos valores. Estas últimas resultan especialmente útiles para la creación de máscaras que permiten analizar regiones de interés de otras imágenes o facilitar el procesamiento de otras imágenes.

## 4.2. Filtros y convolución

Un filtro es un algoritmo tal que dada una imagen  $I(x,y)$ , calcule una nueva imagen  $I'(x,y)$  de forma que en cada píxel de la nueva imagen sea función de un pequeño conjunto de píxeles de la imagen  $I$  alrededor de la posición  $x,y$ .

La plantilla que define el número de elementos y la forma en que se combinan se llama filtro o núcleo (kernel). Si el filtro es lineal la nueva imagen  $I'$  puede expresarse como una suma ponderada de los puntos que recoge el núcleo generalmente incluyendo la posición  $x,y$ . La nueva imagen será por tanto:

$$I'(x,y) = \sum_{i,j \in \text{kernel}} k_{i,j} \cdot I(x+i,y+j)$$

A continuación se muestra un ejemplo de aplicación de un núcleo 3x3 a una imagen en escala de grises 5x5, cada píxel se almacena en un byte. Si la imagen resultado se almacena de la misma forma se perdería información (mínimo valor 0, máximo 255).

50	30	20	87	14
4	84	101	42	201
101	74	6	10	15
124	2	11	33	65
101	45	74	21	10

*Imagen origen I*

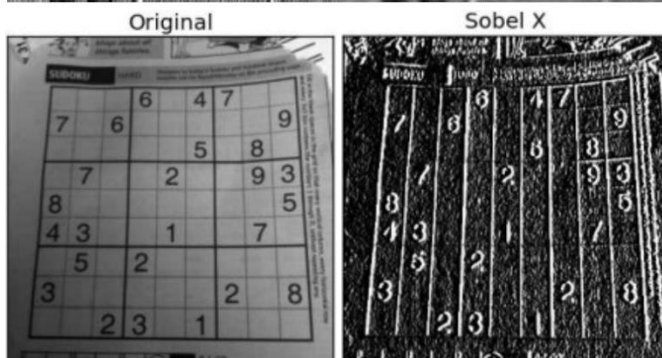
-1	0	1
0	0	0
1	0	-1

*Kernel*

0	0	0	0	0
0	65	121	0	0
0	211	0	46	0
0	-68	-40	73	0
0	0	0	0	0

*Imagen resultado I'*

Unas de las aplicaciones más interesantes de los filtros es la eliminación de ruidos, la aproximación numérica de derivadas y las operaciones morfológicas. La figura siguiente ilustra la aplicación de un filtro gaussiano y un filtro derivativo.



1/273 ·

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

-1	0	1
-2	0	2
-1	0	1

Figura 23. Ejemplo de aplicación filtro gaussiano 5x5 y filtro derivativo Sobel horizontal 3x3  
 Fuentes: [docs.opencv.org] y [es.mathworks.com]

### 4.3. Transformada de Hough aplicada a circunferencias

La transformada de Hough es un método para encontrar líneas, círculos u otras formas simples en una imagen. La original se realizó para detectar líneas y el método fue generalizado para la detección de formas más complejas.

En un espacio bidimensional, como el de una imagen, un círculo puede ser descrito por:

$$(x - a)^2 + (y - b)^2 = r^2$$

Donde a y b son el centro de la circunferencia y r es el radio. Dado que el número de parámetros es 3, se define un acumulador tridimensional con todas las posibles circunferencias que puedan estar contenidas en la imagen. Generalmente se limita un radio máximo y mínimo, así como una resolución en la búsqueda del centro y el radio para que la búsqueda no tenga un tiempo de computación excesivo.

La circunferencia que interseque un mayor número de puntos de la imagen tendrá un valor más elevado en el acumulador. Es por ello que la transformada de Hough requiere una imagen de bordes binaria para que los puntos de intersección estén correctamente definidos. Este tipo de imágenes se pueden lograr aplicando filtros derivativos anteriormente comentados.

En la siguiente figura ilustra la detección en una imagen de únicamente cinco puntos.

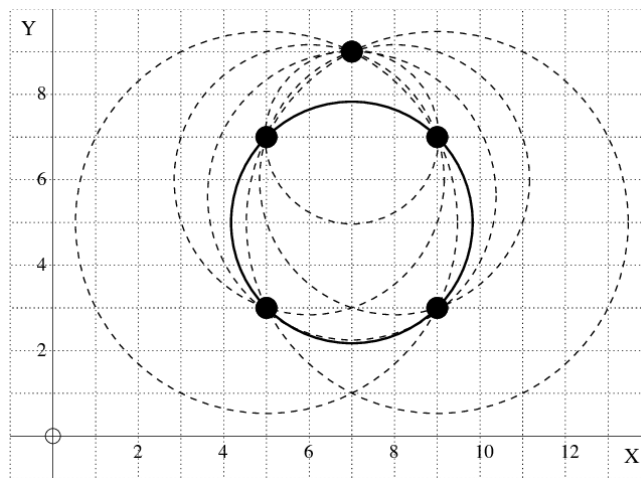


Figura 24. Detección de circunferencia en imagen de cinco puntos. Fuente: [www.researchgate.net]

### 4.4. Mapa de profundidad

Un mapa de profundidad es una imagen que recoge información relacionada con la distancia a la superficie de los objetos respecto desde un punto de vista. En el sistema creado se obtendrá fácilmente haciendo uso de la librería como se ilustrará en el desarrollo del sistema de visión. Una forma alternativa para obtenerlo es a partir de mapas de disparidad empleando imágenes estéreo. La disparidad se refiere a la diferencia de localización en píxeles en imágenes rectificadas desde puntos de vista distintos.

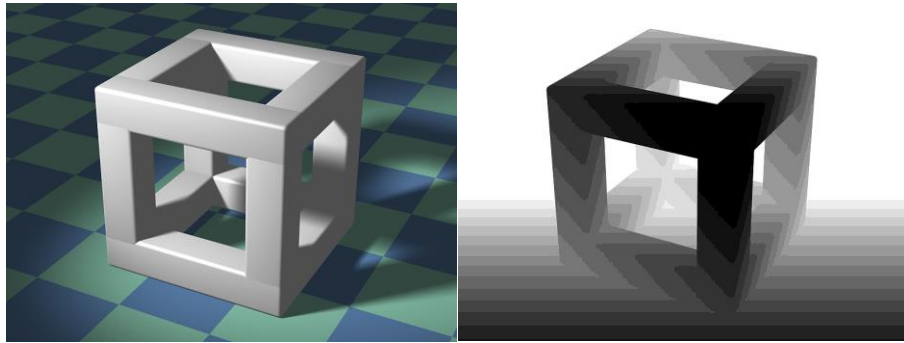


Figura 25. Mapa de disparidad de un cubo. Los negros son puntos más próximos. Fuente: [en.wikipedia.org]

### 4.5. El modelo estenopeico.

Cuando la luz incide sobre un objeto la mayor parte se absorbe, la otra parte que no ha sido absorbida se reflejará y se percibirá por la cámara.

Un modelo simple que explica la captación de imágenes es el modelo estenopeico. El modelo estenopeico plantea una pared imaginaria con un pequeño orificio central, de forma que se bloquean todos los rayos de luz incidentes a excepción de aquellos que atraviesen el pequeño hueco, de forma que únicamente entra un rayo para cada punto de la escena real. Lógicamente el modelo no es aplicable de forma práctica, dado que un único punto no reúne suficiente luz, por ello en los dispositivos reales se emplean lentes que permiten reunir una mayor cantidad de luz a costa de introducir distorsiones. No obstante, permite comprender conceptos fundamentales que más tarde nos permitirán la proyección de los puntos bidimensionales de la imagen al mundo real.

Los rayos de luz que consiguen atravesar el orificio se proyectan en el plano imagen. El tamaño de la imagen se define únicamente por la distancia entre el plano imagen y el plano con el orificio, la distancia focal, como se muestra en la figura:

$$-x = f \cdot \frac{X}{Z}$$

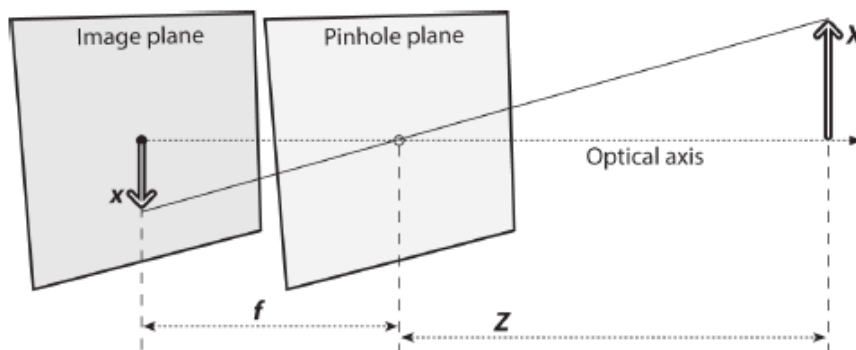


Figura 26. Modelo estenopeico. Fuente: [14]

Si cambiamos de posición el plano imagen y el del orificio, obtendremos un modelo equivalente pero matemáticamente más simple. El orificio se denomina ahora centro de proyección. Como puede verse la imagen ahora no aparece invertida. La interpretación es ahora más sencilla, dado que todo rayo que se refleje de un punto real Q cualquiera pasa por el centro de proyección y la imagen no se encuentra ahora invertida.

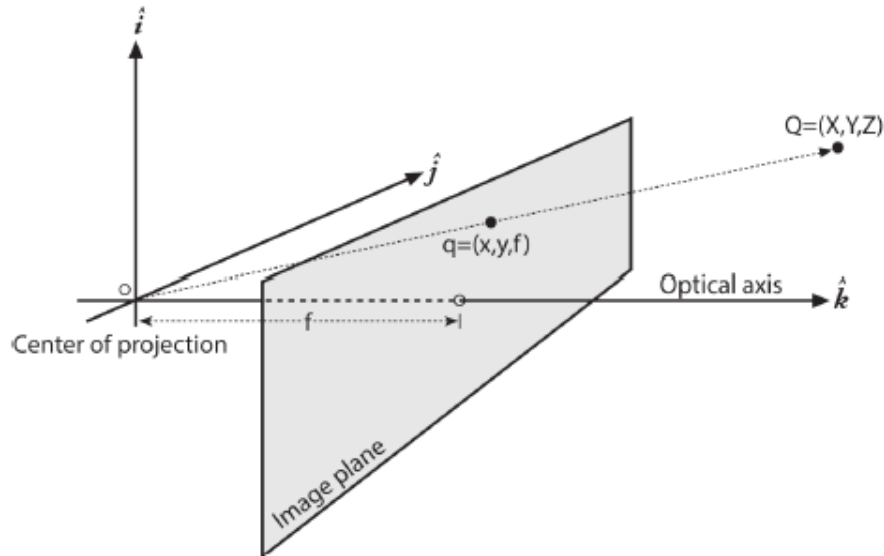


Figura 27. Cambio de posición del plano imagen

El punto de corte entre el plano imagen y el eje óptico se denomina punto principal. Empleando geometría básica puede establecerse la relación entre las coordenadas de un punto real Q y su equivalente en la imagen q, siendo:

$$x_{screen} = f_x \cdot \frac{X}{Z} + c_x$$

$$y_{screen} = f_y \cdot \frac{Y}{Z} + c_y$$

Generalmente debido a imperfecciones en la construcción del dispositivo, el punto principal no siempre coincide con el centro de la imagen, es por ello por lo que se incluyen los factores de corrección  $c_x$  y  $c_y$  que modelan este desplazamiento. Se introducen dos distancias focales debido a que las imágenes suelen ser rectangulares en vez de cuadradas. Así el parámetro  $f_x$  es el producto de la distancia focal física por el tamaño horizontal de la imagen.

#### 4.6. Distorsiones

Teóricamente es posible la construcción de lentes sin distorsiones. No obstante, debido a imperfecciones en el proceso de construcción, no es posible. A continuación se definen los dos tipos principales de distorsión de lentes y cómo se modelan.

#### 4.6.1. Distorsión radial.

Las lentes de las cámaras reales generalmente provocan un cambio en la posición de los píxeles cercanos al borde de la imagen.

La distorsión es de 0 en el centro de la imagen y se incrementa a medida que nos acercamos a la periferia. Existen fórmulas para su corrección. En la práctica la distorsión es pequeña y puede caracterizarse por los primeros términos de la serie de Taylor expandida alrededor de  $r=0$ , los nombraremos  $k_1$  y  $k_2$ . Para cámaras de ojo de pez que introducen una mayor distorsión se emplea un tercer coeficiente  $k_3$ . En general un píxel  $x, y$  de la imagen será trasladado a un punto  $x_{desp}, y_{desp}$  en la forma:

$$x_{desp} = x \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{desp} = y \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$r^2 = x^2 + y^2$$

La siguiente figura ilustra una imagen con y sin distorsión radial.



Figura 28. Eliminación de distorsión radial. Fuente: [<https://github.com/bbenligiray/lens-distortion-rectification>]

#### 4.6.2. Distorsión tangencial

La distorsión tangencial es consecuencia de la colocación de la lente en el dispositivo. Al no estar colocada exactamente paralela al plano imagen se produce distorsión.

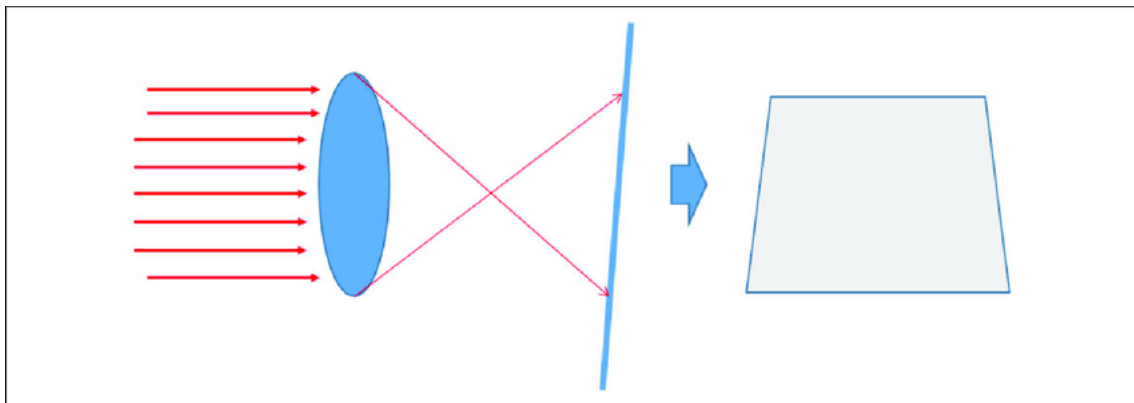


Figura 29. Distorsión tangencial. Fuente: [[www.researchgate.net](http://www.researchgate.net)]

La distorsión tangencial se modela principalmente con dos nuevos parámetros  $p_1$  y  $p_2$  siendo:

$$x_{\text{corregido}} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$

$$y_{\text{corregido}} = y + [2p_2xy + p_1(r^2 + 2y^2)]$$

En total se necesitan cinco coeficientes de distorsión, junto con los cuatro parámetros que definen la cámara (denominados intrínsecos), las distancias focales y los coeficientes de desplazamiento del centro de la imagen. Todos ellos se obtienen durante el proceso de calibración de la cámara, aunque puede proporcionarlos el fabricante dado que puede haber realizado una calibración previa al finalizar la construcción del dispositivo.

En imágenes distorsionadas la correspondencia entre los puntos de la imagen con puntos reales puede ser errónea, luego para establecerla la imagen no debe tener distorsión alguna.

## 5. Desarrollo del proyecto

En este apartado se explica de forma detallada el código desarrollado, describiendo el proceso de realización.

### 5.1. Descripción del problema

Se empleará la estrategia de dividir el problema en partes menores que se resolverán de forma aislada. Así en primer lugar, previo a la creación del paquete de ROS, se desarrollará una clase en C++ que permita el fácil uso del controlador, probándola con la interfaz gráfica o mediante una sencilla aplicación de consola.

El diagrama inferior trata de clarificar el e ilustrar el sistema. Nótese como el sistema que determina la posición del robot, se sustituirá por un único nodo cliente, el cual como se verá más adelante, permitirá introducir la posición desde la consola de comandos.

Otro aspecto relevante es que el nodo servidor no envía directamente instrucciones al controlador, lega la tarea al programa de control. Éste es el único que actúa de forma directa sobre el controlador.

Se escoge el tipo de comunicación entre nodos mediante servicios dado que logra un intercambio de información con confirmación de forma simple como se verá a continuación. También es cierto que podría haber sido aplicado una comunicación por medio de acciones, se opta por los servicios principalmente debido a una mayor sencillez de implementación.

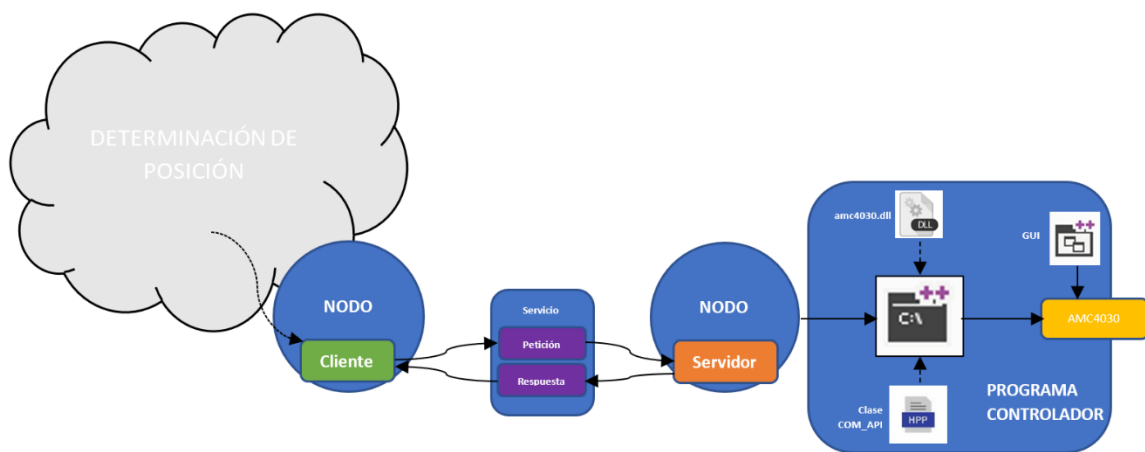


Figura 1. Diagrama de relación entre las partes del proyecto.

El desarrollo comenzará por las partes cercanas al controlador (la clase y el programa de control). Una vez creadas y verificadas se pasará a la creación del paquete de ROS, finalizando con un ejemplo de sistema de visión artificial que determine la posición del robot.



## 5.2. Clase COM\_API

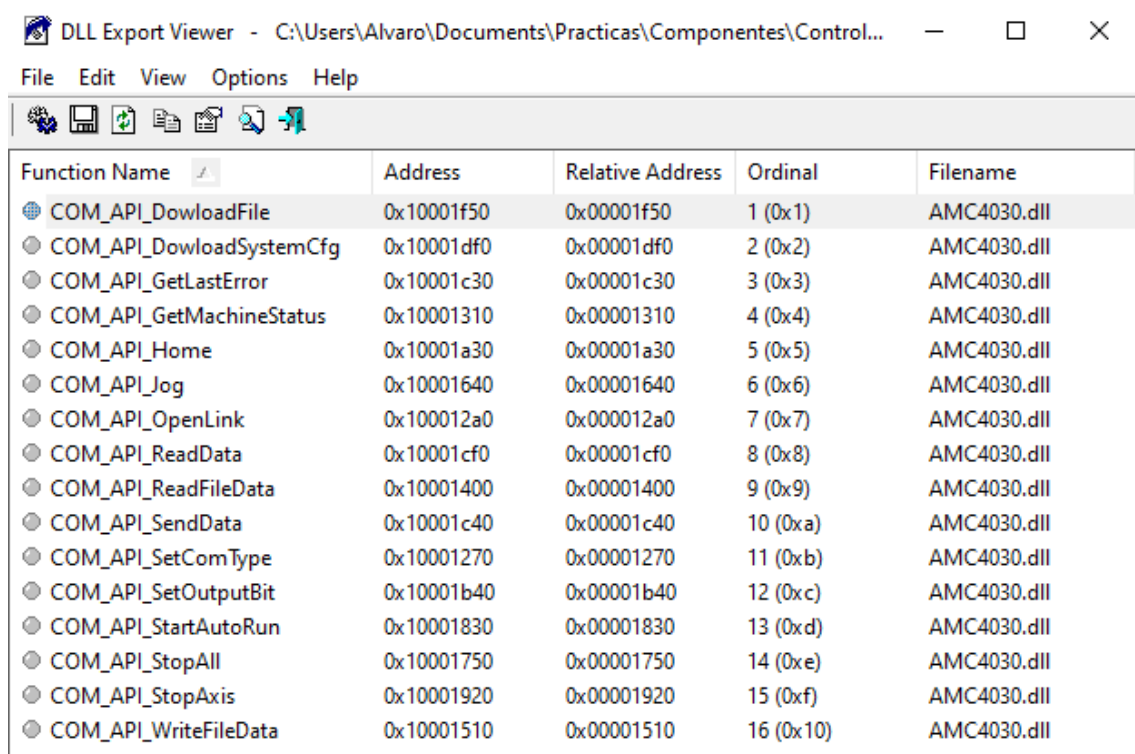
### 5.2.1. Carga dinámica de funciones.

Para interactuar con el controlador la clase COM\_API debe hacer uso de las funciones definidas en el fichero AMC4030.dll. El proceso para poder hacer uso de las funciones es el siguiente:

- Carga del módulo en memoria. Se llevará a cabo mediante la función de Windows “LoadLibrary”.
- Declaración de punteros de funciones para cada una de las funciones de la librería que vayan a usarse.
- Asignación de cada puntero a la dirección de memoria correspondiente, que se obtendrá con la función “GetProcAddress”.

Para el segundo y tercer paso se requiere, además del nombre de la función, el tipo de retorno y parámetros.

Empleando el programa “dll\_export\_viewer” puede conocerse fácilmente el nombre de las funciones.



Function Name	Address	Relative Address	Ordinal	Filename
COM_API_DownloadFile	0x10001f50	0x00001f50	1 (0x1)	AMC4030.dll
COM_API_DownloadSystemCfg	0x10001df0	0x00001df0	2 (0x2)	AMC4030.dll
COM_API_GetLastError	0x10001c30	0x00001c30	3 (0x3)	AMC4030.dll
COM_API_GetMachineStatus	0x10001310	0x00001310	4 (0x4)	AMC4030.dll
COM_API_Home	0x10001a30	0x00001a30	5 (0x5)	AMC4030.dll
COM_API_Jog	0x10001640	0x00001640	6 (0x6)	AMC4030.dll
COM_API_OpenLink	0x100012a0	0x000012a0	7 (0x7)	AMC4030.dll
COM_API_ReadData	0x10001cf0	0x00001cf0	8 (0x8)	AMC4030.dll
COM_API_ReadFileData	0x10001400	0x00001400	9 (0x9)	AMC4030.dll
COM_API_SendData	0x10001c40	0x00001c40	10 (0xa)	AMC4030.dll
COM_API_SetComType	0x10001270	0x00001270	11 (0xb)	AMC4030.dll
COM_API_SetOutputBit	0x10001b40	0x00001b40	12 (0xc)	AMC4030.dll
COM_API_StartAutoRun	0x10001830	0x00001830	13 (0xd)	AMC4030.dll
COM_API_StopAll	0x10001750	0x00001750	14 (0xe)	AMC4030.dll
COM_API_StopAxis	0x10001920	0x00001920	15 (0xf)	AMC4030.dll
COM_API_WriteFileData	0x10001510	0x00001510	16 (0x10)	AMC4030.dll

Figura 30. Vista de funciones del fichero AMC4030.dll.

Resta conocer la definición de cada una de ellas.

Para ello tras analizar el código de la interfaz gráfica del controlador desarrollada en MATLAB por Muhammet Emin [9], en el fichero ComInterface.h se encuentran las definiciones de cada una de las funciones. Las que consideraremos para la clase se muestran a continuación:

```

//#define CALLBACK_DEF int
#define CALLBACK_DEF __declspec(dllexport) int WINAPI

CALLBACK_DEF COM_API_SetComType(int nType);
CALLBACK_DEF COM_API_OpenLink(int nID,int nBound);
CALLBACK_DEF COM_API_GetMachineStatus(unsigned char* unStatus);

CALLBACK_DEF COM_API_Jog(int nAxis,float fDis,float Speed);
CALLBACK_DEF COM_API_Home(int nXAxisSet,int nYAxisSet,int nZAxisSet);
CALLBACK_DEF COM_API_StopAxis(int nXAxisSet,int nYAxisSet,int nZAxisSet);
CALLBACK_DEF COM_API_StopAll();

```

Figura 31. Fragmento de fichero ComInterface.h. Fuente:  
[[https://github.com/meminyanik/AMC4030\\_Matlab\\_Controller](https://github.com/meminyanik/AMC4030_Matlab_Controller)]

Una descripción más detallada de las funciones se puede obtener analizando el código de la interfaz de fábrica del controlador [8]. Se resumen aquí las características principales:

La función `COM_API_SetComType` establece el tipo de comunicación con el controlador. En otro de los ficheros encontramos los tipos y su correspondencia LAN = 0, USB = 1, UART = 2.

La función `COM_API_OpenLink` intenta establecer la comunicación con el dispositivo donde `nId` es el identificador del puerto y `nBound` la velocidad de la transmisión. Devuelve 0 en caso de éxito y uno en caso contrario.

La función `COM_API_GetMachineStatus` devuelve una cadena de caracteres con el estado de la máquina. El primer carácter indica si la máquina se encuentra en pausa = 0, procesando = 1, en movimiento = 4, o buscando origen = 8.

La función `COM_API_Jog` desplaza el eje “`nAxis`” una distancia “`fDis`” mm a una velocidad de “`Speed`” mm/s.

La función `COM_API_Home` realiza el movimiento de búsqueda de origen en el eje cuyo valor de set sea 1 (0 en caso contrario). Por ejemplo para buscar el origen del eje ‘x’ y ‘z’, la instrucción sería `COM_API_Home(1,0,1)`.

La función `COM_API_StopAxis` detiene el eje indicado por Set, con un funcionamiento similar al de la instrucción `COM_API_Home`.

La función `COM_API_StopAll` detiene todos los ejes.

Previa a la declaración de los punteros se definen los siguientes tipos para mejorar la legibilidad de la declaración.

```

typedef int (__stdcall* f_COM_API_SetComType) (int nType);
typedef int (__stdcall* f_COM_API_OpenLink) (int nId, int nBound);
typedef int (__stdcall* f_COM_API_GetMachineStatus) (unsigned char* unStatus);
typedef int (__stdcall* f_COM_API_Jog) (int nAxis, float fDis, float Speed);
typedef int (__stdcall* f_COM_API_Home) (int nXAxisSet, int nYAxisSet, int nZAxisSet);
typedef int (__stdcall* f_COM_API_StopAll) ();
typedef int (__stdcall* f_COM_API_StopAxis) (int nXAxisSet, int nYAxisSet, int nZAxisSet);

```

Figura 32. Creación del tipo de los punteros a las funciones de la librería.

Donde `__stdcall` es la convención de llamada para funciones de la API de Win32. Devuelve una lista de argumentos en la que se indica:

- El orden en que se pasan los argumentos (de derecha a izquierda).
- Cómo se pasan los argumentos (por valor, a menos que se pase un puntero o un tipo referencia).
- La función es responsable de sacar sus propios argumentos de la pila.

Finalmente pueden crearse los punteros privados de la clase que tendrán acceso a esas funciones, se incluye además la declaración de la instancia del proceso de la librería dinámica, que se almacena en `m_hGetProcDLL`.

```
HINSTANCE m_hGetProcDLL = LoadLibrary(L"AMC4030.dll"); //L stands for long pointer to constant wide string.
f_COM_API_SetComType m_SetComType =
(f_COM_API_SetComType)GetProcAddress(m_hGetProcDLL, "COM_API_SetComType");
f_COM_API_OpenLink m_OpenLink = (f_COM_API_OpenLink)GetProcAddress(m_hGetProcDLL,
"COM_API_OpenLink");
f_COM_API_GetMachineStatus m_GetMachineStatus =
(f_COM_API_GetMachineStatus)GetProcAddress(m_hGetProcDLL, "COM_API_GetMachineStatus");
f_COM_API_Jog m_Jog = (f_COM_API_Jog)GetProcAddress(m_hGetProcDLL, "COM_API_Jog");
f_COM_API_Home m_Home = (f_COM_API_Home)GetProcAddress(m_hGetProcDLL, "COM_API_Home");
f_COM_API_StopAll m_StopAll = (f_COM_API_StopAll)GetProcAddress(m_hGetProcDLL,
"COM_API_StopAll");
f_COM_API_StopAxis m_StopAxis = (f_COM_API_StopAxis)GetProcAddress(m_hGetProcDLL,
"COM_API_StopAxis");
```

Figura 33. Declaración de punteros a las funciones de la librería.

### 5.2.2. Descripción de la clase COM\_API

La siguiente tabla recoge las definiciones de las enumeraciones a las que tendrá acceso el usuario (especificación de ejes, tipos de excepciones, etc).

Enum	Valores	Descripción
STATUS_TYPES	PAUSE, PROCESSING, MOVEMENT, HOME_MOVEMENT,	Posibles estados del controlador.
COM_TYPES	COM_LAN, COM_USB, COM_UART	Tipo de comunicación.
N_AXIS	X_AXIS, Y_AXIS, Z_AXIS	Especificación de eje.
EXCEPTION_TYPE	LOAD_DLL_FAILED, CONNECTION_FAILED, NOT_VALID_POINT, CONFIG_READ_ERROR, CONFIG_WRITE_ERROR	Posibles excepciones que puede lanzar la clase.

En la siguiente tabla se muestran las principales funciones miembro de la clase.

Nombre de miembro	DEFINICIÓN
Constructor	Construye el objeto, necesario especificar el tipo de comunicación, identificador del puerto, velocidad de transmisión y tamaño de ejes. Obtiene la primera posición del fichero config.txt
Destructor	Destruye el objeto, deteniendo el movimiento de todos los ejes por seguridad. Registra última posición en el fichero config.txt
pos	Acceso al valor de la posición del robot.
homeSet	Acceso al valor de m_homeSet, que indica si se han establecido los orígenes de cada eje durante el tiempo de vida del objeto.
getMachineStatus	Asigna al char array pasado como argumento el estado de la máquina.
move	Desplazamiento del eje especificado una distancia en mm a una velocidad en mm/s
desp	Desplazamiento al punto (x,y,z) mm a una velocidad speed mm/s
home	Establece la búsqueda de origen de uno o varios ejes.
stopAxis	Detiene uno o varios ejes.
operator<<	Muestra para que ejes se ha establecido el origen desde la creación del objeto y la posición del controlador.

### 5.3. Interfaz gráfica.

#### 5.3.1. Introducción a la herramienta.

Como se comentó, se creará empleando Windows Forms que soportan el lenguaje C++/CLI. Permite la creación de formularios, simplificando significativamente la parte gráfica. De una forma similar al entorno Qt, los elementos se desplazan a la ventana conocidos como herramientas (*tools*), en una vista de diseño que visualiza los elementos declarados en el archivo de encabezado (.h). Pueden variarse multitud de características visuales (color, forma, posición, tamaño...) y de control (habilitado o deshabilitado controles, respuesta ante acciones de usuario, rangos máximos y mínimos, si es únicamente de lectura, etc).

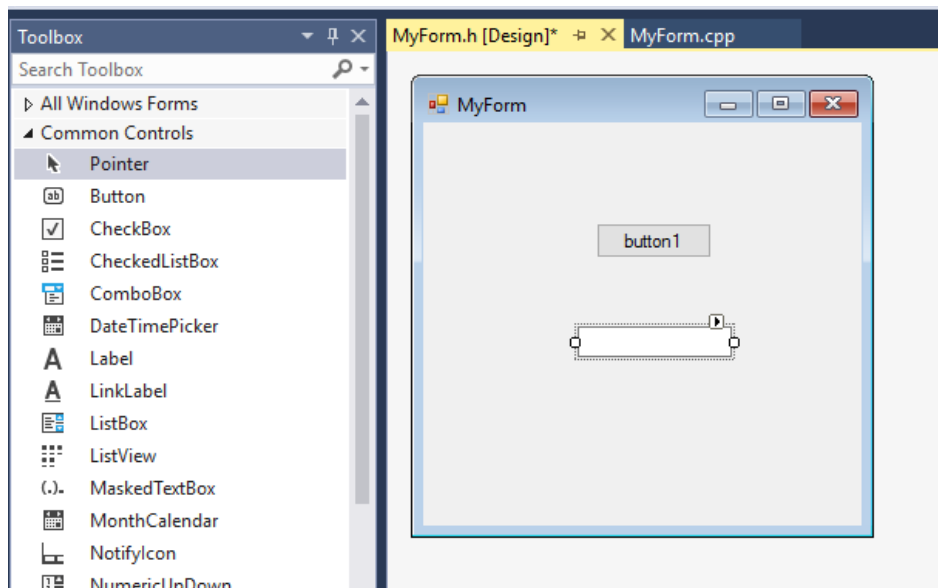


Figura 34. Vista del modo diseño en visual estudio en proyecto c++/cli. Fuente: [\[https://social.msdn.microsoft.com/\]](https://social.msdn.microsoft.com/)

Al importar en el proyecto el archivo Windows Forms, se crea de forma automática una clase referencia con el nombre MyForm. Esta clase es la ventana en la que se introducen las herramientas.

Una clase de referencia es una clase cuya duración de objetos se administra de forma automática. Es decir, cuando el objeto ya no es accesible, se libera de memoria.

La accesibilidad de los miembros predeterminada es privada. Para aquellos objetos que requieran asignación dinámica de memoria, deberán accederse con punteros creados con el símbolo “^” en lugar del empleado para punteros comunes “\*”, indicando que se encuentra bajo el recolector automático de basura de .NET, además de usarse el operador gnew (“*garbage collected new()*”) en lugar de *new()*.

El código autogenerated de cada una de las herramientas que se haya creado se encuentra bajo la guarda *#pragma región Windows Form Designer generated code*. La modificación de esta parte del código hace que no coincida la información de la vista gráfica [designer] y las definiciones en el propio código obteniéndose un error que bloquea la vista gráfica. La solución es recuperar los anteriores ficheros o deshacer los cambios realizados en esa parte.

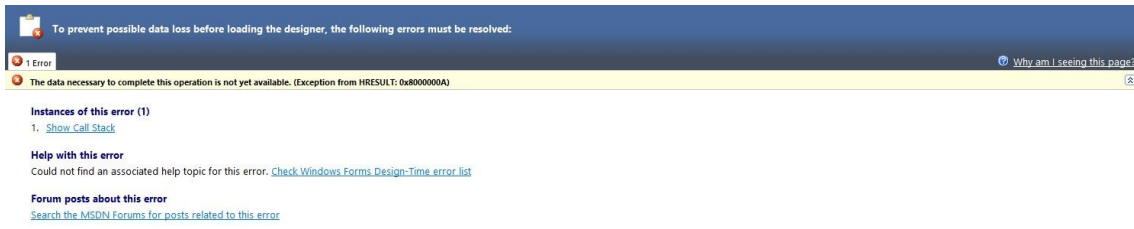


Figura 35. Captura de error en vista de diseño de Windows Forms.

Una vez realizada la distribución de herramientas en el formulario, la única tarea será la conexión y definición de los eventos que registrarán el comportamiento de cada una de ellas.

Se pueden encontrar todos los tipos de eventos asociados a una herramienta concreta, en la ventana de propiedades y seleccionando el símbolo del rayo, como se indica en la figura adjunta:

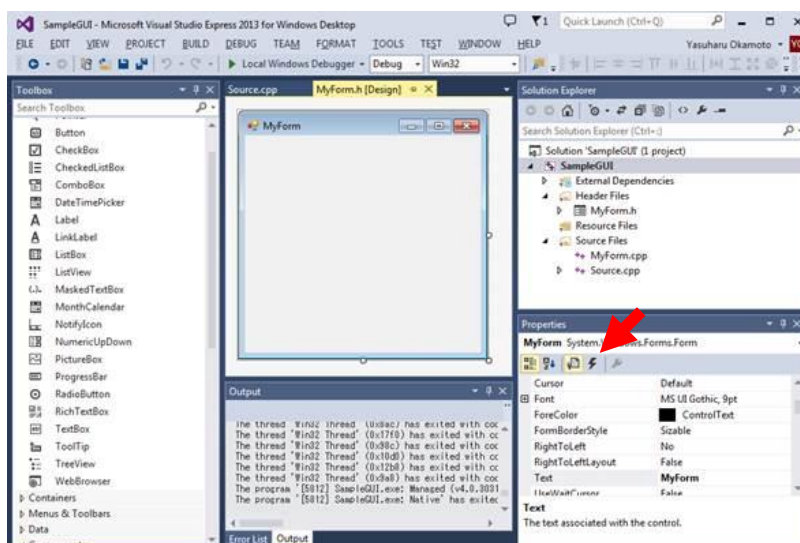


Figura 36. Señalización del especificador de eventos.

Al seleccionar el evento deseado, se crea en el fichero de encabezado de la declaración de la clase una función miembro con el nombre especificado que responderá al evento. Se generan automáticamente los indicadores del cuerpo del controlador (*handler*) del evento “{}” en el fichero de cabecera en vez del fichero fuente (.cpp).

Resultó ser un error de la versión que corrigieron posteriormente, no obstante, por comodidad para no tener que trasladar cada una de las definiciones de los controladores que colocaba el auto generador, opté por definir las en ese lugar empeorando con ello la legibilidad del código.

La forma de las funciones generadas por el creador de eventos es la siguiente:

```
private: System::Void down_button_MouseUp(System::Object^ sender,
System::Windows::Forms::EventArgs^ e) {
}
```

Figura 37. Estructura de función de evento.

Define el controlador del evento “levantar el botón izquierdo del ratón”, aplicado sobre la herramienta “down\_button”.

Puede observarse que la función posee únicamente dos argumentos, la comprensión de ambos es esencial para definir los controladores de eventos:

- *sender*: puntero con el recolector automático de basura de .NET. Proporciona una referencia al objeto que generó el evento.
- *e*: puntero con el recolector automático de basura de .NET. Proporciona una referencia al objeto que se está controlando. A través de él puede obtenerse por ejemplo la posición del ratón en eventos del ratón o datos en cuadros de texto de la interfaz gráfica etc.

Nótese que al definirse como miembros privados de la clase referencia, todos los controladores de eventos poseen acceso a todas las herramientas definidas en ella. Esto permitirá la interacción entre ellas logrando, por ejemplo, de forma sencilla que la habilitación de algunas herramientas dependa del estado de otras.

### 5.3.2. Vista general de la interfaz gráfica.

La interfaz gráfica consta de una única ventana, si bien es cierto que emplea pequeñas ventanas de mensajes para informar en caso de situaciones atípicas o uso erróneo por parte del usuario.

Define por defecto los parámetros del controlador que van a emplearse (tipo de conexión, identificador de puerto, velocidad de transmisión y tamaño de ejes).

En la figura inferior pueden observarse las diferentes partes que componen la interfaz.

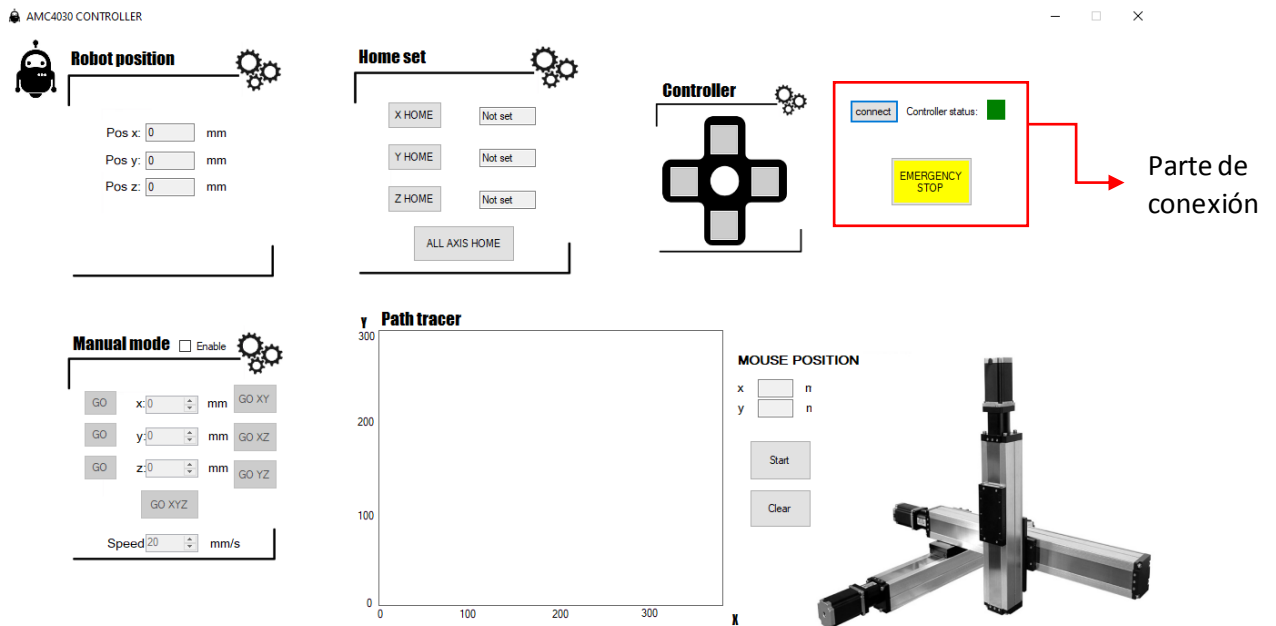


Figura 38. Distintas partes que componen la interfaz gráfica.

A continuación, se realizará una descripción de cada una de ellas:

#### a. Parte de conexión

Se compone únicamente de una herramienta botón y un panel. Al pulsarse el botón de conexión se hace una llamada al constructor de la clase `COM_API` y se intenta crear un objeto de la misma. Dado que su creación puede generar excepciones, (recuérdense los tipos vistos en el apartado 4.2.2), en caso de que sucedan se crean herramientas del tipo `MessageBox` con el mensaje de error correspondiente. Distinguiremos de acuerdo a cada una de las excepciones que se puedan producir:

- Excepción `LOAD_DLL_FAILED`: esta excepción se genera en la clase a la hora de cargar la librería de vínculos dinámicos (`AMC4030.dll`) en caso de no encontrarse. Se considera un error crítico de programa y una vez aceptado se cierra y finaliza la interfaz.
- Excepción `CONFIG_READ_ERROR`: esta excepción es consecuencia de no poderse abrir el fichero `config.txt` a la hora de leerse. Es necesario a la hora de crear el objeto de la clase, dado que constituye el valor de la posición inicial, luego también es considerado error crítico y cierra la aplicación. Otra posible alternativa sería obligar al usuario a inicializar la posición mediante la parte de Home, designada para ello, pero se optó por la primera alternativa.
- Excepción `CONFIG_WRITE_ERROR`: excepción similar a la anterior, pero generada al destruir la clase, luego no debería generarse en el botón `connect`.
- Excepción `CONNECTION_FAILED`: la clase genera esta excepción al hacer uso de la función de la librería `COM_API_OpenLink`. Puede generarse por múltiples razones desde un identificador de puerto erróneo a tener el dispositivo desconectado del PC inconscientemente al hacer uso de la interfaz (situación que experimenté en

innumerables ocasiones), por lo que el mensaje es esta vez algo más genérico en el que se indica que no pudo encontrarse el controlador.

El panel será rojo cuando no se haya podido establecer la conexión con el controlador (no se haya podido crear el objeto de la clase COM\_API) y pasará a verde cuando se haya logrado conectar.

Se contempla también la situación de que el usuario trate de hacer uso de las funcionalidades en caso de no haber establecido conexión, en este caso se mostrará un mensaje de error indicando que debe establecerse conexión con el controlador.

Dado que es en esta zona es la primera que el usuario debe notar (todos los demás controles estarán bloqueados hasta que se conecte el controlador) se escogió poner cerca el botón de emergencia por este motivo. Éste detendrá todos los ejes del robot en caso de que sea pulsado, haciendo uso de la función de la clase StopAxis().

#### b. Display

En esta parte se muestra la posición objetivo cada eje del robot en milímetros.

#### c. Home set

En esta parte se incluyen los controles de retorno a origen del robot. Se cuenta con cuatro botones. Tres de ellos permiten la búsqueda de origen de cada eje por separado y el tercero permite que todos vayan al origen. Cuenta con cuadros de texto que indican si para esta ejecución se ha realizado la búsqueda de origen o no.

#### d. Manual mode

Para que los controles de esta parte estén activados debe accionarse la casilla Enable. Una vez activado los controles de movimiento (botones GO) así como la cruceta de control se habilitarán. En este modo el usuario podrá especificar tanto la posición como la velocidad a la que se realizará el movimiento. Se consideran los valores máximos de cada eje, así como la velocidad máxima del robot, mostrando un mensaje de error en caso de que se sobrepase y sustituyendo el valor introducido por el valor máximo posible.

Los botones de movimiento permiten desplazar de forma aislada cada eje, dos de ellos o todos mediante el botón inferior. La herramienta empleada para especificar los valores de posición se trata de "numericupdown" que permite la variación de una unidad (positiva o negativa) del valor especificado mediante las flechas a su derecha o establecer el valor directamente.

Éste constituye el modo más práctico para el uso manual dado que permite alcanzar todas las posiciones del espacio de trabajo del robot fácilmente. Las siguientes partes solo permiten el movimiento en un plano.

#### e. Controller

Para estar habilitada debe haberse activado la casilla ("checkbox") enable del modo manual. Una vez activada podrá desplazar el robot fácilmente pulsando con el ratón en cada uno de los extremos de la cruz. El botón derecho e izquierdo permiten desplazamiento en el eje x de forma positiva y negativa, el botón superior e inferior en el eje y (positiva y negativa respectivamente).



### f. Path tracer

Únicamente permite también el desplazamiento en un plano. El usuario indicará en el plano de trabajo los puntos a alcanzar haciendo click con el ratón sobre él. La posición se muestra en la casilla mouse position. Todas las posiciones se almacenarán en un vector que el robot recorrerá una vez pulse “Start”. Una vez se pulse el botón “Clear” se borrará la ruta y se podrá empezar nuevamente. Se permite el trazado de la ruta antes de haberse conectado al controlador, no obstante, si se pulsa el botón “Start” se obtendrá el mensaje de error correspondiente.

La velocidad de la ruta es constante y de 50 mm/s, aunque otra alternativa interesante habría sido tomar el valor de la velocidad del modo manual.

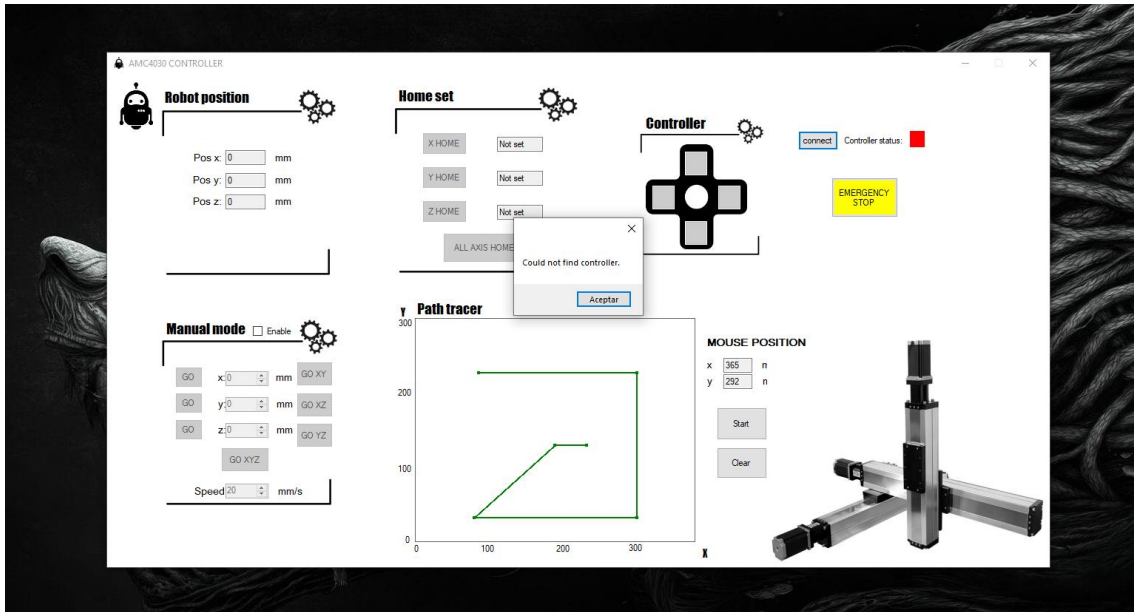


Figura 39. Captura de funcionamiento de interfaz.

### 5.3.3. Detalles de implementación.

A continuación, se muestra de y comenta de forma breve las partes de mayor relevancia del código de la interfaz.

En primer lugar, se encuentran la declaración de parámetros y variables que reúnen características acerca del controlador, la cruz de control y la ruta.

```

///Robot and controller parameters-----
const int port_id = 24;
const int baud_rate = 115200;
const std::vector<int> axis_sizes{ 380,280,200 }; //Real axis sizes (mm): x = 400,y = 300
const int max speed = 200;
Com_api controller;
///-----

///Timer-----
auto start = std::chrono::steady_clock::now();
///-----

///Remote parameters-----
const int remote speed = 20;
std::vector<int> remote_pos = { 0,0,0 };
///-----

///Route data-----

```

```
std::vector<std::vector<int>> graph_points;
bool first_point{ true };
const float route_speed = 50;
```

Figura 40. Fragmento de código de parámetros y constantes en la interfaz.

En primer lugar, se muestran los valores que definen el objeto controlador. Una alternativa de diseño que se contempló fue la personalización de la conexión en la interfaz, pudiendo escoger el tipo de conexión, pero dado que la finalidad era principalmente el funcionamiento automático en el sistema de ROS2 se escogió esta forma por simplicidad.

Debajo encontramos el temporizador, así como parámetros de la cruz (*“Remote parameters”*). En la ruta se definen las variables para almacenar los puntos y reconocer si el punto seleccionado es el primero. La finalidad de esta variable es que a la hora de trazar la línea de la ruta se obvie la primera vez dado que únicamente hay punto trazado.

Debajo se encuentra el código que se crea al incluir un archivo Windows Forms. Se trata de la declaración de la clase referencia MyForm. El constructor únicamente inicializará los componentes. Así al desplazar y agregar herramientas a la ventana en modo diseño se declara un miembro privado de la clase y se le asigna memoria al construirse la clase.

```
public ref class MyForm : public System::Windows::Forms::Form
{
    public:
        MyForm(void)
        {
            InitializeComponent();
        }
    ...
private: System::Windows::Forms::PictureBox^ robot_image;
    ...
    void InitializeComponent(void)
    {
        ...
        this->robot_image = (gcnew System::Windows::Forms::PictureBox());
        ...
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->robot_image))->BeginInit();
        ...
        //
        // robot_image
        //
        this->robot_image->Image =
        (cli::safe_cast<System::Drawing::Image^>(resources->GetObject(L"robot_image.Image")));
        this->robot_image->Location = System::Drawing::Point(1190, 434);
        this->robot_image->Name = L"robot_image";
        this->robot_image->Size = System::Drawing::Size(507, 383);
        this->robot_image->SizeMode =
        System::Windows::Forms::PictureBoxSizeMode::StretchImage;
        this->robot_image->TabIndex = 0;
        this->robot_image->TabStop = false;
        ...
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->robot_image))->EndInit();
        ...
    }
}
```

Figura 41. Fragmento del código necesario para la creación de la imagen del robot.

En el ejemplo de la figura superior se muestran las instrucciones necesarias para crear la imagen del robot. Los puntos suspensivos indican otras líneas no mostradas para visualizar con mayor facilidad el fragmento de interés.

En primer lugar como ya se indicó el constructor de la clase referencia MyForm hace una llamada a la función miembro privada de la clase InitializeComponent(). Ésta asigna memoria dinámicamente al miembro privado robot\_image previamente declarado (como puntero especial de .NET como ya se comentó anteriormente). Se puede observar que la imagen se trata de la herramienta “PictureBox” y se ha establecido en el modo de diseño que reciba el identificador de robot\_image.

Previa a la especificación de propiedades debe estar presente la llamada a *BeginInit()*, y al finalizar *EndInit()*.

La parte de inicialización de componentes corresponde al código autogenerated, la información para rellenar los diversos campos se toma de la parte de diseño.

Una vez inicializados pueden ser modificados en los controladores de eventos, dado que al ser declarados miembros de la clase tienen acceso a todas y cada una de las características de las herramientas. Así por ejemplo se observa que la casilla de verificación del modo manual tiene la capacidad de habilitar otros controles en función de su estado.

```
private: System::Void manual_check_Click(System::Object^ sender, System::EventArgs^ e)
{
    if (this->panel_estado->BackColor == System::Drawing::Color::Red) {
        System::Windows::Forms::MessageBox::Show("Controller not connected");
        manual_check->Checked = false;
    }
    else {
        if (manual_check->Checked) {
            x_go_button->Enabled = true;
            y_go_button->Enabled = true;
            z_go_button->Enabled = true;
            xy_go_button->Enabled = true;
            xz_go_button->Enabled = true;
            yz_go_button->Enabled = true;
            xyz_go_button->Enabled = true;
            up_button->Enabled = true;
            down_button->Enabled = true;
            left_button->Enabled = true;
            right_button->Enabled = true;
            manual_x_value->Enabled = true;
            manual_y_value->Enabled = true;
            manual_z_value->Enabled = true;
            manual_speed_value->Enabled = true;
        }
        else {
            x_go_button->Enabled = false;
            y_go_button->Enabled = false;
            z_go_button->Enabled = false;
            xy_go_button->Enabled = false;
            xz_go_button->Enabled = false;
            yz_go_button->Enabled = false;
            xyz_go_button->Enabled = false;
            up_button->Enabled = false;
            down_button->Enabled = false;
            left_button->Enabled = false;
            right_button->Enabled = false;
            manual_x_value->Enabled = false;
            manual_y_value->Enabled = false;
            manual_z_value->Enabled = false;
            manual_speed_value->Enabled = false;
        }
    }
}
```

Figura 42. Código de controlador de evento de “manual\_check\_Click”

El fragmento de código muestra el ejemplo citado. Una vez se pulsa la casilla enable se comprueba el estado del controlador comprobando el color del panel. Si el controlador está desconectado y se trata de hacer click sobre ella se envía un mensaje de error y se reestablece su valor de no verificado.

En caso contrario, si el estado es del *checkbox* manual es verificado, (el usuario quiere habilitar el modo manual) se habilitan los controles pertinentes modificando la propiedad de Enable d. Si por otro lado éste no está verificado (el usuario quiere deshabilitar el modo manual) se modifica nuevamente el campo enabled de esos controles.

Respecto a los controladores de eventos consistirán principalmente en llamadas a la librería pasando los datos introducidos por el usuario en la interfaz como argumentos.

Otro aspecto que encontré interesante es el inconveniente del uso de tipos del sistema por las herramientas, que impedían el paso directo de información a los argumentos de la clase. Por ello en el código de los controladores de eventos se encuentra de forma reiterada el uso de la función “Convert::To...”. Por ejemplo los valores numéricos de las herramientas “NumericUpDown” son de tipo decimal (System::decimal). Para almacenar la posición objetivo que en la clase se establece como vector de enteros y poder llamar a la instrucción desp() es necesaria la conversión:

```
std::vector<int> objective = { controller.pos () [Com_api::X_AXIS],
Convert::ToInt32 (manual_y_value->Value), controller.pos () [Com_api::Z_AXIS] };
```

Figura 43. Ejemplo uso de función de conversión.

Otro aspecto relevante fue la obtención de la posición del ratón para el trazado de rutas, en el que se hizo uso de los argumentos de evento “e”. A continuación, se muestra el fragmento de código que actualiza las casillas de texto de *Mouse Position* de acuerdo al movimiento del ratón.

```
private: System::Void graph_MouseMove (System::Object^ sender,
System::Windows::Forms::MouseEventArgs^ e) {
    this->x_mouse->Text = Convert::ToString (e->Location.X);
    this->y_mouse->Text = Convert::ToString (300 - e->Location.Y);
}
```

Figura 44. Código de evento movimiento del ratón en el gráfico de rutas.

Al desplazarse el ratón sobre el gráfico se toma la posición del ratón de los campos Location, se convierten a String y se actualizan los campos de posición.

## 5.4. Programa de control.

El programa controlador será llamado por el nodo servidor del paquete de ROS2. Éste aceptará los parámetros de configuración del controlador (tipo de comunicación, puerto, ...) además de la posición del controlador. Controlará las excepciones que pueda generar la creación de la clase y aprovechando que los nodos de ROS han de ejecutarse desde la consola (no siendo necesario si se crean *launch files*), el programa mostrará en ella el correspondiente mensaje de error.

Así pues, el programa controlador consistirá únicamente en la creación del objeto de la clase y el tratamiento de los datos introducidos por la línea de comando. Dada su reducida extensión

se incluye a continuación para comentarlo con mayor facilidad, aunque puede encontrarse en el anexo correspondiente.

```
#include <iostream>
#include <Windows.h>
#include "COM_API.h"

const std::vector<int> axisSizes{ 380,250,200 };
const float max_speed{ 200 };

int main(int argc, char **argv)
{
    if (argc != 8 && argc != 4) {
        std::cout << "Incorrect use of program. Specify initialization parameters,
position and speed:\n " <<
            "COM_TYPES n_id baud_rate x y z speed\n";
        exit(EXIT_FAILURE);
    }
    Com_api controller;
    if (argc == 4) {
        std::cout << "Checking controller status.";
    }
    try {
        controller = Com_api(static_cast<Com_api::COM_TYPES>(std::atoi(argv[1])),
std::atoi(argv[2]), std::atoi(argv[3]), axisSizes);
    }
    catch (Com_api::EXCEPTION_TYPE e) {
        switch (e) {
            case Com_api::LOAD_DLL_FAILED:
                std::cerr << "The file 'AMC4030.dll' could not be loaded or is corrupt.
Setup cannot continue.\n";
                return 1;
            case Com_api::CONNECTION_FAILED:
                std::cerr << "Could not find controller.\n";
                return 2;
            case Com_api::CONFIG_READ_ERROR:
                std::cerr << "Config read error.\n";
                return 3;
            case Com_api::CONFIG_WRITE_ERROR:
                std::cerr << "Config write error.\n";
                return 4;
        }
    }
    if (argc == 8) { //Perform move
        int position_x = atoi(argv[4]);
        int position_y = atoi(argv[5]);
        int position_z = atoi(argv[6]);
        float speed = atof(argv[7]);
        if (speed > max_speed) {
            std::cout << "Max speed exceeded";
        }
        else { //Make movement
            std::vector<int> objective = { position_x, position_y, position_z };
            controller.desp(objective, speed);
            while (controller.status() != Com_api::PAUSE) { //Monitorize movement
                until the robot stops.
                //Possible actions here.
                Sleep(33);
            }
        }
    }
}
```

Figura 45. Programa controlador.

La ejecución del programa controlador es de la siguiente forma:

```
>Controlador_AMC4030.exe [COM_TYPES] [n_id] [baud_rate] [x] [y] [z]
[speed]
```

Los tres primeros parámetros son ya bien conocidos, permiten establecer el tipo de comunicación con el controlador. Posteriormente se incluyen las coordenadas del punto a alcanzar y la velocidad del movimiento. Este es el caso que se puede observar en el código en el que se pasan 8 argumentos.

Se distingue una segunda forma de ejecución en la que únicamente hay cuatro argumentos en el comando.

```
>Controlador_AMC4030.exe [COM_TYPES] [n_id] [baud_rate]
```

Esta forma de ejecución tiene como finalidad comprobar el estado del controlador y permite al nodo servidor realizar una prueba de éste. Nótese como la instrucción de movimiento se contempla únicamente en caso de haberse realizado la primera forma de ejecución.

Tras el adecuado tratamiento de los comandos introducidos, el programa hace uso de la función de la clase `move` para alcanzar el punto deseado. Nótese que se monitoriza el desplazamiento del robot cada 33 ms. Se analiza si el robot está en pausa, tras lo cual finaliza el movimiento y el programa finaliza. Podrían emplearse por ejemplo las señales digitales del controlador para detectar situaciones anómalas de funcionamiento. Puede ser necesario reducir el tiempo de espera para trabajos a grandes velocidades.

## 5.5. Paquete de ROS2

Una vez han sido creadas las herramientas que permiten la interacción con el controlador pasamos ahora al desarrollo del paquete de ROS. Dado que se optó por la creación de una comunicación mediante servicios, el proceso puede dividirse en dos partes bien diferenciadas, la creación del paquete de la interfaz (*controller\_interfaces*) y el paquete del cliente servidor (*cpp\_srvcli*).

### 5.5.1. Paquete de interfaz

En primer lugar, crearemos la interfaz del servicio. Recuérdese que se trataba de la estructura de los datos que intercambian ambos nodos. Para ello se siguen los pasos de creación de los paquetes vistos en la breve introducción a ROS2 en el apartado 2.

#### a. Creación del paquete.

Una vez preparado el entorno de trabajo ejecutamos el comando de creación del paquete que tendrá como nombre *controller\_interfaces* en el directorio fuente (*src*).

```
ros2 pkg create --build-type ament_cmake controller_interfaces
```

#### b. Definición de la interfaz

Creamos el directorio *srv* en ella definimos el servicio que recibe el nombre de *Movement.srv*. La estructura se muestra en la figura inferior.

```
int64 x
int64 y
int64 z
```

```
int64 speed
---
string resp
```

Figura 46. Estructura del servicio *Movement.srv*

Vemos que la petición consta de cuatro campos y la respuesta de uno. La respuesta se compone de cuatro enteros, tres para la posición y otro para la velocidad. El campo respuesta tiene únicamente de una cadena de caracteres, que contendrá información acerca de la realización del movimiento por parte del servidor y de interés para el cliente.

Una vez creado deberán modificarse los ficheros *CMakeLists.txt* y *package.xml* para que el paquete pueda ser compilado correctamente.

### c. Compilación

En el fichero *CMakeLists.txt* ha de incluirse la dependencia del paquete *controller\_interfaces* con el paquete de generación de interfaces de ros *rosidl\_default\_generators*. Este se encarga de traducir la declaración de la interfaz al lenguaje correspondiente que se esté usando (C++ o Python). Se consigue incluyendo la instrucción de búsqueda del paquete y haciendo obligatorio encontrarlo mediante el campo "REQUIRED". El fichero queda de la siguiente forma:

```
cmake_minimum_required(VERSION 3.5)
project(controller_interfaces)

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/Movement.srv"
)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Figura 47. *CMakeLists* del paquete *controller\_interfaces*.

Es necesaria a su vez la modificación del fichero *package.xml* para que concuerde con la información recogida en el fichero *CMakeLists.txt*, será necesario añadir la dependencia de *rosidl\_default\_generators*.

Se aprovecha además para rellenar los campos de descripción, licencias, etc.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>controller_interfaces</name>
  <version>0.0.0</version>
  <description>Package with the interfaces for the controller AMC4030</description>
```

```

<maintainer email="██████████@astibot.es">ASTIBOT</maintainer>
<license>Apache License 2.0</license>

<buildtool_depend>ament_cmake</buildtool_depend>
<build_depend>roslint</build_depend>

<exec_depend>roslint</exec_depend>

<member_of_group>roslint_interface_packages</member_of_group>

<test_depend>ament_lint_auto</test_depend>
<test_depend>ament_lint_common</test_depend>

<export>
  <build_type>ament_cmake</build_type>
</export>
</package>

```

Figura 48. *package.xml* del paquete *controller\_interfaces*.

Una vez modificados correctamente puede ejecutarse el comando de compilación, siempre desde el directorio del espacio de trabajo.

```
colcon build --packages-select controller_interfaces
```

Tras la compilación en el directorio `install` habrá una carpeta con el nombre del paquete. En ella se define la librería para hacer uso del servicio, que recoge la declaración de la estructura y diversas funciones. No se incluye en el documento debido a su extensión, aunque resulta sumamente interesante los ficheros *movement\_struct.hpp* en el que se puede ver la interpretación y conversión por parte del paquete *roslint default generators* del pequeño *Movement.srv* a dos estructuras de C++ plenamente funcionales, una para la respuesta *Movement\_Response\_* y otra para la petición *Movement\_Request\_*.

### 5.5.2. Paquete cliente servidor.

Una vez creada la interfaz del servicio se desarrolla el paquete que define el nodo cliente y el nodo servidor que harán uso de ella. Para la declaración y creación de los nodos se modificará el ejemplo del tutorial de ROS2 *Writing a simple service and client (C++)* y que puede encontrarse en la bibliografía.

El procedimiento es similar al mostrado en la creación del paquete de interfaz.

#### a. Creación del paquete.

En el directorio fuente de espacio de trabajo se ejecuta el comando:

```
ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp controller_interfaces
```

El argumento `--dependencies` modificará los ficheros del paquete *package.xml* y *CMakeLists.txt* añadiendo las dependencias necesarias que permitirán hacer uso de la interfaz.

Tras rellenar los campos de descripción en el fichero *package.xml* el resultado es el siguiente:

```

<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>cpp_srvcli</name>
  <version>0.0.0</version>
  <description>C++ client server for AMC4030 communication</description>

```



```

<maintainer email="██████████@astibot.es">ASTIBOT</maintainer>
<license>Apache License 2.0</license>

<buildtool_depend>ament_cmake</buildtool_depend>

<depend>rclcpp</depend>
<depend>example_interfaces</depend>
<depend>controller_interfaces</depend>

<test_depend>ament_lint_auto</test_depend>
<test_depend>ament_lint_common</test_depend>

<export>
  <build_type>ament_cmake</build_type>
</export>
</package>

```

Figura 49. *package.xml* del paquete *cpp\_srvcli*

En el archivo *CMakeLists.txt* se deberá especificar la creación de dos ejecutables, uno para el nodo cliente y otro para el nodo servidor. Ambos dependerán, como se muestra en el fichero *package.xml* de la interfaz del servicio anteriormente creada.

```

cmake_minimum_required(VERSION 3.5)
project(cpp_srvcli)

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(controller_interfaces REQUIRED)

add_executable(server src/controller_server.cpp)
ament_target_dependencies(server
  rclcpp controller_interfaces)

add_executable(client src/controller_client.cpp)
ament_target_dependencies(client
  rclcpp controller_interfaces)

install(TARGETS
  server
  client
  DESTINATION lib/${PROJECT_NAME})

ament_package()

```

Figura 50. *CMakeLists.txt* del paquete *cpp\_srvcli*

### b. Nodo servidor

Como se comentó anteriormente el nodo servidor se encargará de hacer la llamada al programa que envía instrucciones al controlador. Quedará a la espera de recibir la posición del nodo cliente, que simulará el sistema que determina la posición del robot. El código del servidor se incluye a continuación, se irá comentando parte por parte seguidamente:

```

#include "rclcpp/rclcpp.hpp"
#include "controller_interfaces/srv/movement.hpp"

#include <windows.h>
#include <direct.h> // _getcwd
#include <stdio.h>
#include <tchar.h>
#include <memory>

```

```

#include <iostream>

int64_t com_type;
int64_t port_id;
int64_t n_baud;

void move(const std::shared_ptr<controller_interfaces::srv::Movement::Request>
request,
         std::shared_ptr<controller_interfaces::srv::Movement::Response>
response)
{
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\nx: %ld" " y: %ld" " z:
%ld" " speed: %ld",
              request->x, request->y, request->z, request->speed);
    char* package_path;
    if ((package_path = _getcwd(NULL,0)) == NULL) {
        std::cerr << "Error calling getcwd\n";
        exit(EXIT_FAILURE);
    }

    std::string command = std::string(package_path)+"\\Controlador_AMC4030.exe "+
        std::to_string(com_type)+" "+
        std::to_string(port_id) + " "+
        std::to_string(n_baud) + " "+
        std::to_string(request->x) + " "+
        std::to_string(request->y) + " "+
        std::to_string(request->z) + " "+
        std::to_string(request->speed);
    int cont code = system(command.c_str());

    response->resp = (cont_code == 0) ? "Movement successfull" : "Movement failed";
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%s]", response-
>resp.c_str());
}

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("movement_server");

    com_type = node->declare_parameter<int64_t>("COM_TYPE",2);
    port_id = node->declare_parameter<int64_t>("Port_id",24);
    n_baud = node->declare_parameter<int64_t>("Baud_rate",115200);

    rclcpp::Service<controller_interfaces::srv::Movement>::SharedPtr service =
        node->create_service<controller_interfaces::srv::Movement>("movement", &move);

    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to receive position.");

    rclcpp::spin(node);
    rclcpp::shutdown();
}

```

Figura 51. Código de nodo servidor.

En primer lugar, encontramos la inclusión de dos importantes archivos de cabecera:

- rclcpp.hpp: API estándar que proporciona interacción con ROS2, es decir, permite hacer uso de la mayoría de los elementos de éste. La documentación hace referencia a ella como librería cliente de ROS2 para C++.
- movement.hpp: ésta es la estructura autogenerada que se comentó en el apartado previos, generada tras compilar el paquete interfaz. Recuérdese que contaba con una parte de petición y otra de respuesta. Para acceder a ellas basta con acceder al espacio de nombres adecuado (controller\_interfaces::srv::Movement)

El nodo servidor contará con tres parámetros que definirán el tipo de conexión con el controlador, estos son el tipo de comunicación (COM\_TYPE), el identificador del puerto (Port\_id) y la velocidad de transmisión (Baud\_rate).

Dado que son parámetros del nodo podrán ser modificados o establecidos mediante los comandos de ROS2 oportunos para adaptarse a otro tipo de conexiones.

Las tres variables que almacenarán su valor son las que se definen como:

```
int64_t com_type;
int64_t port_id;
int64_t n_baud;
```

A continuación, encontramos la función de movimiento (move). Consta únicamente de dos argumentos, son punteros compartidos a las estructuras de petición y respuesta. Han de ser compartidos dado que el nodo cliente deberá acceder a ellas como se comprobará en el análisis del código posterior.

Cuando la función es llamada notifica en la consola de comandos que se ha recibido una petición con información acerca de la misma (posición y velocidad). Una vez hecho esto se obtiene el directorio de ejecución del paquete. Éste se empleará en la construcción del comando para ejecutar el programa de control.

Tras la construcción del comando se hace una llamada al sistema para que ejecute el programa de control con los argumentos que envía el cliente. De resultar correcta la conexión con el controlador devolverá 0 y se notificará al cliente a través del campo resp de la respuesta. En caso contrario el programa de control notificará en la consola de comandos del servidor la naturaleza del error (ver gestión de excepciones en la creación de la clase del programa de control) y se informará al cliente de que el movimiento no pudo realizarse correctamente para que pueda realizar una nueva petición una vez se haya solventado el error.

Pasamos ahora a el análisis de la función principal del nodo servidor.

En primer lugar, se inicializa librería cliente de ROS2 con el comando init. A continuación, se crea el nodo servidor con el nombre *“movement server”*.

Una vez finalizada la creación del nodo se declaran los parámetros de éste y se establece la directiva del servicio para el nodo, asignándole la función *“move”*.

Finalmente se establece que el nodo ha sido debidamente creado notificando en la consola de comandos que está listo para recibir posiciones y se dispone el servicio mediante el comando *“spin”*.

### c. Nodo cliente

El nodo cliente actuará como sustituto del sistema que determina la posición del robot. Será por tanto su deber la construcción del mensaje que se envíe al servidor, que contendrá la posición a alcanzar.

Análoga a la descripción del nodo servidor se incluye el código para más tarde comentarlo parte por parte:

```
#include "rclcpp/rclcpp.hpp"
#include "controller_interfaces/srv/movement.hpp"
#include <chrono>
```

```

#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    if (argc != 5) {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: client x y z speed");
        return 1;
    }

    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("movement_client");
    rclcpp::Client<controller_interfaces::srv::Movement>::SharedPtr client =
        node->create_client<controller_interfaces::srv::Movement>("movement");

    auto request = std::make_shared<controller_interfaces::srv::Movement::Request>();
    request->x = atoll(argv[1]);
    request->y = atoll(argv[2]);
    request->z = atoll(argv[3]);
    request->speed = atoll(argv[4]);

    while (!client->wait_for_service(1s)) {
        if (!rclcpp::ok()) {
            RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the
service. Exiting.");
            return 0;
        }
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting
again...");
    }

    auto result = client->async_send_request(request);
    // Wait for the result.
    if (rclcpp::spin_until_future_complete(node, result) ==
        rclcpp::executor::FutureReturnCode::SUCCESS)
    {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), result.get()->resp);
    }
    else {
        RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service movement");
    }

    rclcpp::shutdown();
    return 0;
}

```

Figura 52. Código del nodo cliente.

Los ficheros de inclusión son similares, luego pasamos directamente al análisis de la función principal.

En primer lugar, se ejecuta la instrucción de inicialización de la librería de ROS2. Una vez hecho esto se observa si el número de argumentos pasados por el comando es 5 para verificar que la llamada al nodo servidor es correcta. El comando de ROS2 para lanzar el nodo cliente será de la forma:

```
ros2 run cpp_srvcli client [x] [y] [z] [speed]
```

La posición x, y, z vendrá dada en mm y la velocidad en mm/s. Recuérdese que para lanzar un nodo debe referirse en primer lugar al nombre del paquete, que en este caso recibe el nombre de cpp\_srvcli.

A continuación, se crea el nodo cliente como puntero compartido, dado que debe compartir recursos con el nodo servidor y se crea el cliente para ese nodo asignándole la interfaz de movimiento.

```
std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("movement_client");
rclcpp::Client<controller_interfaces::srv::Movement>::SharedPtr client =
node->create_client<controller_interfaces::srv::Movement>("movement");
```

Una vez creado se pasan los argumentos introducidos por la línea de comandos a sus campos pertenecientes en la petición.

En caso de no encontrar disponible el servidor al lanzar la petición, ésta se vuelve a lanzar cada segundo mientras el servidor no esté activo, notificándose en la consola de comandos la situación. Se permite al usuario salir del bucle de espera con la señal SIGINT (Ctrl+C), esto se logra mediante la comprobación de la función `rclcpp::ok()` que devuelve *true* si la señal aún no se ha lanzado y *false* en caso contrario.

De encontrarse disponible se lanza la petición de forma asíncrona. La guía de ROS2 recomienda el uso de llamadas asíncronas dado que las llamadas síncronas pueden ocasionar bloqueos entre los distintos nodos.

La variable `request` almacenará el resultado de la petición una vez ésta haya concluido.

Con `spin_until_future_complete` se lanza el nodo hasta que la petición haya finalizado. Una vez finalizado se analiza el valor de retorno *FutureReturnCode*. Existen tres posibilidades:

- Éxito: la acción futura está completada y puede accederse al campo de la respuesta haciendo uso de `get()` sin bloqueo.
- Interrupción: la acción futura no está completa (interrupción de la petición de servicio mediante Ctrl-C u otro error).
- Fuera de plazo: la petición se ha excedido de tiempo.

En este caso únicamente se analiza si la petición ha tenido éxito, en este caso se devuelve el mensaje del nodo servidor que contiene la respuesta de éste. De haberse producido una interrupción durante la petición, o ésta se excede de tiempo se muestra un mensaje de error.

## 5.6. Sistema de visión

En este apartado se muestra el desarrollo del sistema que permite el movimiento del robot a puntos detectados por cámaras con sensor de profundidad. Se recuerda que el ejercicio consistirá en la detección de una pelota y mandar el punto tridimensional de su extremo derecho para permitir al robot alcanzarla.

En la siguiente figura se muestra el sistema robótico completo (únicamente se hace uso de la cámara superior izquierda en el ejemplo).

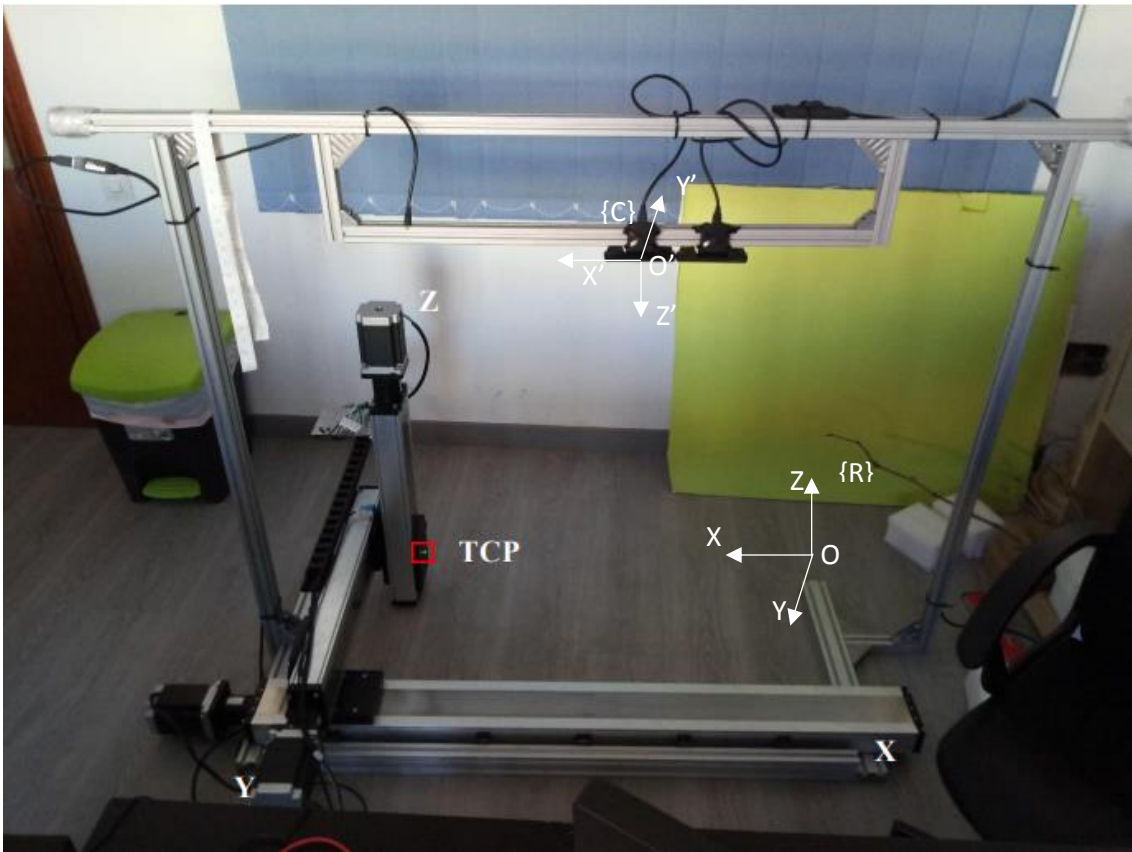


Figura 53. Sistema robótico para ejemplo de visión artificial.

En la figura 53 se muestra el sistema de referencia del robot  $\{R\}$  de la cámara  $\{C\}$  y el que será considerado el TCP del robot, un tornillo fijado a la última plataforma del tercer brazo. En la figura el robot se encuentra en la posición 870,0,0. Las dimensiones de los ejes del robot (útiles) son 870 mm para el eje X, 270 mm para el eje Y y 300 mm para el eje Z.

En los siguientes se comentarán los aspectos más relevantes del código ilustrándose el ejemplo de detección y búsqueda de la bola.

### 5.6.1. Obtención del mapa de profundidad

La siguiente función almacena en dos objetos Mat de OpenCV, que han de entenderse como un contenedor al igual que vector, la imagen de color y su correspondiente mapa de profundidad en metros.

```
rs2_intrinsics get_depth_map(const std::string &serial, cv::Mat &rgb_image, cv::Mat
&depth_map) {
    //Configuración previa de la cámara
    rs2::align align_to_color(RS2_STREAM_COLOR); //Alinea la imagen y el mapa de
profundidad
    rs2::colorizer color_map; //Textura para el mapa de profundidad
    rs2::pipeline pipe; //Abstrae el dispositivo, "equivalente" de la clase VideoCapture
de Opencv
    rs2::config cfg; //Recoge la configuración del sensor
    cfg.enable_device(serial);
    int frame_w = 640;
    int frame_h = 480;
    cfg.enable_stream(RS2_STREAM_COLOR, frame_w, frame_h);
    cfg.enable_stream(RS2_STREAM_DEPTH, frame_w, frame_h);

    //Apertura y obtención del sensor.
    rs2::pipeline profile profile = pipe.start(cfg); //Apertura de dispositivo
```

```

auto sensor = profile.get_device().first<rs2::depth_sensor>(); //Obtención del
sensor
sensor.set_option(RS2_OPTION_VISUAL_PRESET, RS2_SR300_VISUAL_PRESET_DEFAULT);

//Espera por fotografías
rs2::frameset data = pipe.wait_for_frames(); //Confunto de fotografías de la cámara
data = align_to_color.process(data); //Alineación de imagen de color y mapa de
profundidad.
rs2::frame color_frame = data.get_color_frame(); //Imagen RGB
rs2::depth_frame depth = data.get_depth_frame(); //Mapa de profundidad
rs2::frame c_depth_frame = depth.apply_filter(color_map); //Mapa de profundidad
reescalado para visualización

//Conversión de frame a Mat
cv::Mat color_image(cv::Size(frame_w,frame_h), CV_8UC3,
(void*)color_frame.get_data(), cv::Mat::AUTO_STEP);
cv::Mat depth_image(cv::Size(frame_w,frame_h), CV_8UC3,
(void*)c_depth_frame.get_data(), cv::Mat::AUTO_STEP);
cv::Mat depth_mat(cv::Size(frame_w,frame_h), CV_16U, (void*)depth.get_data(),
cv::Mat::AUTO_STEP);
cv::cvtColor(color_image, color_image, cv::COLOR_RGB2BGR); //Opencv uses BGR format.

//Muestra por pantalla el mapa de profundidad y la imagen en color.
cv::imshow("Color image", color_image);
cv::imshow("Depth image", depth_image);
cv::waitKey(0);

//Obtención del mapa de profundidad
float scale = sensor.get_depth_scale(); //Obtención de escala para paso a metros
cv::Mat depth_mat_s(cv::Size(frame_w, frame_h), CV_32F);
for (int x = 0; x < depth_mat.cols; ++x)
    for (int y = 0; y < depth_mat.rows; ++y) {
        depth_mat_s.at<float>(y,x) = scale * depth_mat.at<int16_t>(y,x);
    }

//Copia a matrices pasadas como argumento a la función y retorno de parámetros
intrinsecos
rgb_image = color_image.clone();
depth_map = depth_mat_s.clone();

return depth.get_profile().as<rs2::video_stream_profile>().get_intrinsics();
}

```

Figura 54. Código obtención de mapa de profundidad.

Lógicamente el mostrar el mapa de profundidad por pantalla no es relevante para el funcionamiento posterior luego puede ser eliminado perfectamente. No obstante permite visualizar el mapa de profundidad para verificar el correcto funcionamiento (“aparente”) del sensor.

Devuelve además los parámetros intrínsecos del sensor de profundidad, necesarios para pasos posteriores en los que se obtendrá un punto concreto en coordenadas del sistema de referencia de la cámara. Al hacer uso de la función verá aparecer en la pantalla ambas imágenes como se ilustra en la figura siguiente.

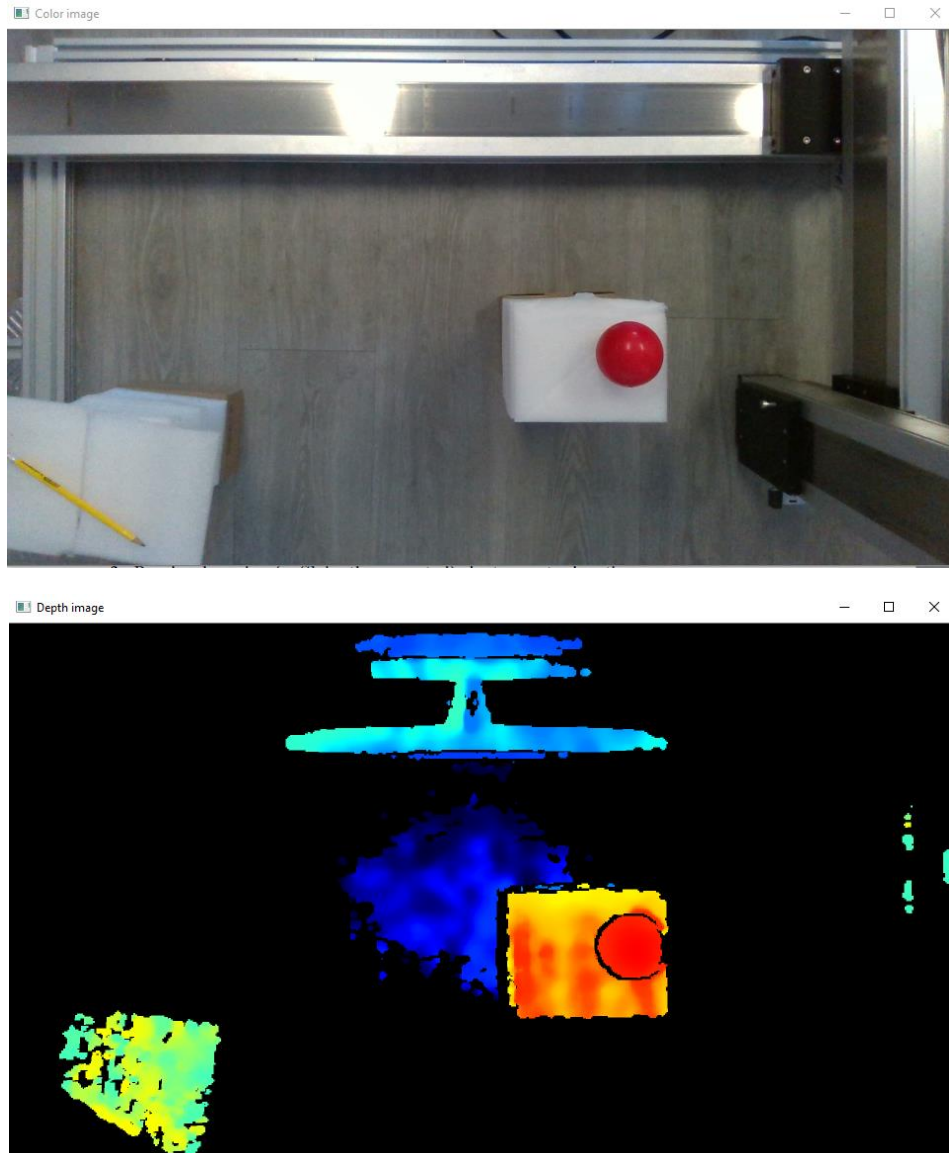


Figura 55. Imagen y mapa de profundidad en el ejemplo de la detección de la bola.

### 5.6.2. Obtención de un punto en coordenadas de la cámara

Tras la obtención del mapa de profundidad, podemos conocer la distancia en metros de cualquier punto respecto al sistema de referencia de la cámara en coordenadas de la imagen [pixel, pixel, m]. Para obtener el punto en coordenadas del mundo real [m,m,m] es necesario hacer uso de los parámetros intrínsecos.

La librería almacena los parámetros intrínsecos en una estructura llamada `rs2_intrinsics` que incluye:

- Altura y anchura de la imagen.
- Las distancias focales de ambos ejes  $f_x$  y  $f_y$ . Son resultado del producto de la distancia focal física por el tamaño de la imagen.
- Modelo de distorsión de la imagen.
- Offset del centro de proyección en ambos ejes ( $ppx$  y  $ppy$ ).
- Coeficientes de distorsión radial y tangencial (`coeffs`).



Haciendo uso del modelo estenopeico podemos referir las coordenadas de un punto en la imagen al sistema de referencia de la cámara siendo:

$$X = \frac{(x_{screen} - pp_x)}{f_x} Z \quad Y = \frac{y_{screen} - pp_y}{f_y} Z$$

Donde Z es la ya conocida distancia al punto de la imagen definido por las coordenadas de un pixel en la imagen ( $x_{screen}, y_{screen}$ ).

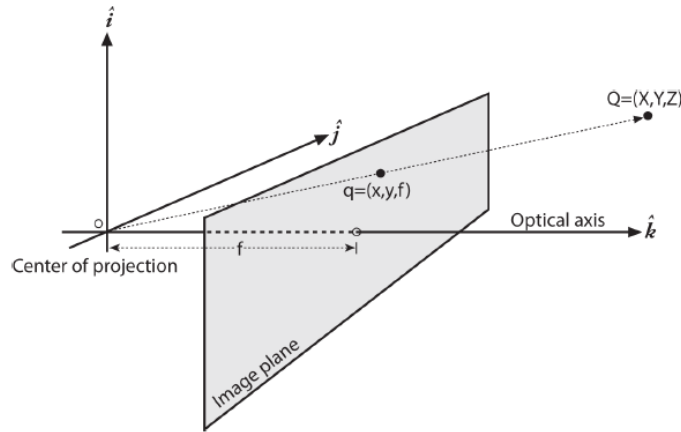


Figura 56. Modelo estenopeico. Un punto Q se proyecta en el plano imagen resultando en el punto q. Fuente [14]

Resta corregir las distorsiones radiales y tangenciales de las imágenes. El modelo empleado en el ejemplo es el de Brown-Conrady. Los coeficientes de distorsión son por tanto 5, de los cuales los dos primeros y el quinto se emplean para la corrección de la distorsión radial y el tercer y cuarto coeficientes para la distorsión tangencial. Las fórmulas de corrección son las siguientes:

$$x_{corregido} = x \cdot f + 2k_2xy + 2k_3rx^2$$

$$y_{corregido} = y \cdot f + 2k_3xy + 2k_2ry^2$$

Siendo:

$$r = x^2 + y^2$$

$$f = 1 + k_0r^2 + k_1r^4 + k_4r^6$$

Con lo visto puede interpretarse la función `deproject_pixel_to_point` a la que se delega esta tarea:

```
//Deproject pixel from a depth image to 3d camera coordinates
cv::Point3f deproject_pixel_to_point(const struct rs2_intrinsics& intrin, const
cv::Point3f& depth_point) {
    assert(intrin.model != RS2_DISTORTION_BROWN_CONRADY);
    assert(intrin.model != RS2_DISTORTION_FTHETA);
    float x = (depth_point.x - intrin.ppx) / intrin.fx;
    float y = (depth_point.y - intrin.ppy) / intrin.fy;
    if (intrin.model == RS2_DISTORTION_INVERSE_BROWN_CONRADY) {
        float r2 = x * x + y * y;
        float f = 1 + intrin.coeffs[0] * r2 + intrin.coeffs[1] * r2 * r2 +
intrin.coeffs[4] * r2 * r2 * r2;
        float ux = x * f + 2 * intrin.coeffs[2] * x * y + intrin.coeffs[3] * (r2 + 2 * x
* x);
        float uy = y * f + 2 * intrin.coeffs[3] * x * y + intrin.coeffs[2] * (r2 + 2 * y
* y);
    }
}
```

```

    x = ux;
    y = uy;
}
return cv::Point3f(depth_point.z * x, depth_point.z * y, depth_point.z);
}

```

*Función 57. Función para obtener puntos en coordenadas de la cámara [m,m,m] a partir de puntos de imagen [pixel, pixel, m]*

### 5.6.3. Obtención de un punto en coordenadas del robot

Para referir las coordenadas de la cámara al sistema de referencia del robot deberá encontrarse la matriz de transformación homogénea entre el sistema de referencia del robot R y el de la cámara C. Los vectores de coordenadas del sistema de referencia C expresados en coordenadas del sistema R son (ver figura 53):

$$\hat{x}_C^R = (1 \quad 0 \quad 0)$$

$$\hat{y}_C^R = (0 \quad -1 \quad 0)$$

$$\hat{z}_C^R = (0 \quad 0 \quad -1)$$

La matriz de transformación sería por tanto:

$$T_C^R = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & -1 & 0 & dy \\ 0 & 0 & -1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Donde dx, dy y dz forman el vector de desplazamiento entre los orígenes de ambos sistemas de referencia. Puede obtenerse mediante medición directa, pero dado que ya pueden obtenerse los puntos de la imagen en el sistema de referencia de la cámara, pueden calcularse de forma mucho más precisa calculando las coordenadas de un punto conocido en el espacio de trabajo del robot.

El punto empleado es el del TCP de la figura 53 que corresponde al punto del robot  $p^R = (0.87 \ 0 \ 0)$ , así pues sustituyendo:

$$p^R = T_C^R p^C; \begin{pmatrix} 0.87 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & -1 & 0 & dy \\ 0 & 0 & -1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_{cx} \\ p_{cy} \\ p_{cz} \\ 1 \end{pmatrix}; \begin{cases} dx = 0.87 - p_{cx} \\ dy = p_{cy} \\ dz = p_{cz} \end{cases}$$

### 5.6.4. Detección de la bola

Para la detección de la bola se emplea la transformada de Hough aplicada en regiones próximas a la cámara. Se crean por tanto tres funciones:

- *remove\_background*: recorre el mapa de profundidad eliminando puntos a una distancia mayor del umbral especificado.
- *find\_rois*: detecta las regiones de interés próximas a la cámara, haciendo uso de contornos. Estas serán las zonas donde se aplique la búsqueda de bolas.
- *ball\_detect*: detecta circunferencias en la región de interés.

El lector interesado puede encontrar el código de estas funciones en los anexos finales. El resultado de la aplicación de las rutinas a la imagen y mapa de profundidad anterior es el siguiente:



*Figura 58. Ejemplo detección de una bola en una de las regiones de interés.*

Finalmente queda referir el punto exterior derecho y conocer su profundidad. Dado que el mapa de profundidad era algo ruidoso en el borde de la bola, la profundidad se obtiene del punto central de la bola y se le resta el radio.

#### 5.6.5. Ejecución del programa de movimiento.

A continuación se configura el comando de movimiento a ejecutar, creando para ello un cliente con las coordenadas correspondientes al punto descrito anteriormente. Se modificó el programa de control para que se alineasen en primer lugar los ejes Y y Z para finalmente mover el X.

## 6. Prueba de funcionamiento

En este apartado se analiza y muestra el uso del paquete de ROS2 y el programa de control en un sistema de visión que indica la posición deseada. Se recogen los comandos comentados para la ejecución de nodos y se describen de forma breve las distintas situaciones que pueden encontrarse a la hora de usarlo y los comandos para ejecutar cada nodo. Se supone que el paquete se ha instalado debidamente, en caso de experimentar problemas durante la instalación refiérase al anexo correspondiente.

### 6.1. Lanzamiento del nodo cliente.

Para lanzar el nodo cliente el comando de ejecución es el siguiente, como ya se explicó en el análisis del código:

```
ros2 run cpp_srvcli client x y z speed
```

Donde x, y, z eran las coordenadas del punto que se quiere alcanzar en mm y speed la velocidad en mm/s. En el ejemplo se lanza un nodo cliente, para el punto 50, 50, 50 mm a una velocidad de 50 mm/s.

```
C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_ame_ws>ros2 run cpp_srvcli client 50 50 50 50
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
[INFO] [rclcpp]: service not available, waiting again...
```

*Figura 59. Lanzamiento de nodo cliente.*

Obtenemos lo esperado, el mensaje de espera aparece cada segundo, indicando que el servidor aún no ha sido creado.

Si la llamada no es correcta se mostrará un mensaje con la forma correcta de uso. En el ejemplo se trata de crear un nodo cliente sin ningún parámetro.

```
C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_ame_ws>ros2 run cpp_srvcli client
[INFO] [rclcpp]: usage: client x y z speed
```

*Figura 60. Error en la creación del nodo cliente.*

### 6.2. Lanzamiento del nodo servidor.

Recordemos que el comando de ejecución del nodo servidor es el siguiente:

```
ros2 run cpp_srvcli server
```

Si éste se crea debidamente se verá aparecer el mensaje en la consola de comandos de que el servidor está preparado para recibir peticiones.

```
C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws>ros2 run cpp_srvcli server
[INFO] [rclcpp]: Ready to receive position.
```

*Figura 61. Lanzamiento del nodo servidor.*

Tras recibir una petición se trata de establecer conexión con el controlador. En caso de no encontrarse el programa de control muestra por pantalla el mensaje de error correspondiente y el servidor informa al cliente. En el siguiente ejemplo se muestra el funcionamiento del nodo servidor tras desconectar la alimentación del controlador una vez ha sido creado.

```
C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws>ros2 run cpp_srvcli server
[INFO] [rclcpp]: Ready to receive position.
[INFO] [rclcpp]: Incoming request
x: 50 y: 50 z: 50 speed: 50
Could not find controller.
[INFO] [rclcpp]: sending back response: [Movement failed]
```

*Figura 62. Llamada fallida al programa de control.*

Por otro lado, si el controlador se encuentra habilitado y se logra la ejecución del programa de control se envía al cliente la respuesta de éxito.

```
C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws>ros2 run cpp_srvcli client 50 50 0 50
[INFO] [rclcpp]: Movement successful

C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws>0 0 0 20
"0" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws>ros2 run cpp_srvcli client 20 20 0 20
[INFO] [rclcpp]: Movement successful

C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws>ros2 run cpp_srvcli client 0 0 0 20
[INFO] [rclcpp]: Movement successful
```

```
C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws>ros2 run cpp_srvcli server
[INFO] [rclcpp]: Ready to receive position.
[INFO] [rclcpp]: Incoming request
x: 50 y: 50 z: 0 speed: 50
[INFO] [rclcpp]: sending back response: [Movement successful]
[INFO] [rclcpp]: Incoming request
x: 20 y: 20 z: 0 speed: 20
[INFO] [rclcpp]: sending back response: [Movement successful]
[INFO] [rclcpp]: Incoming request
x: 0 y: 0 z: 0 speed: 20
[INFO] [rclcpp]: sending back response: [Movement successful]
```

*Figura 63. Funcionamiento normal. Consola superior nodo cliente, consola inferior nodo servidor.*

### 6.3. Aplicación de comandos de ROS2.

Una vez comprobado el correcto funcionamiento de ambos nodos y su interacción, pueden aplicarse los comandos vistos durante la introducción básica a ROS2. Permitirán revisar la estructura del sistema y verificar que todo ha sido creado debidamente.

Una vez lanzado el nodo servidor podemos comprobar que está activo desde otra consola de comandos mostrando el listado de nodos.

```
C:\Windows\System32>ros2 node list
/movement_server
```

*Figura 64. Listado de nodos.*

Vemos que figura el nodo servidor bajo el nombre /movement\_server. Recuérdese que el nodo contaba con parámetros para la configuración de la comunicación con el controlador, pueden obtenerse con el comando de listado de parámetros.

```
C:\Windows\System32>ros2 param list
/movement_server:
  Baud_rate
  COM_TYPE
  Port_id
  use_sim_time
```

*Figura 65. Listado de parámetros.*

En la figura inferior se obtiene el valor de uno de los parámetros (get), y se modifica su valor usando el comando set. En la siguiente figura puede observarse la fácil modificación de la velocidad de transmisión.

```
C:\Windows\System32>ros2 param get /movement_server Baud_rate
Integer value is: 115200

C:\Windows\System32>ros2 param set /movement_server Baud_rate 9600
Set parameter successful

C:\Windows\System32>ros2 param get /movement_server Baud_rate
Integer value is: 9600
```

*Figura 66. Obtención y modificación de parámetros.*

Para guardar la configuración de parámetros puede ejecutarse el comando de la figura inferior.

```
C:\Users\Alvaro\Documents\Practicas\ROS\capturas>ros2 param dump /movement_server
Saving to: .\movement_server.yaml
```

*Figura 67. Comando para guardar fichero con parámetros de nodo servidor.*

Se almacenará un archivo en el directorio de ejecución en sintaxis YAML, para que pueda ser cargado posteriormente. El fichero obtenido se muestra en la figura inferior.

```
movement_server:
  ros__parameters:
    Baud_rate: 9600
    COM_TYPE: 2
    Port_id: 24
    use_sim_time: false
```

*Figura 68. Fichero con parámetros de nodo servidor.*

El parámetro `use_sim_time` es común a todos los nodos de ROS. Especifica si el tiempo se obtiene de un tema externo denominado `/clock` que publica el tiempo de simulación. Si se especifica `false` se emplea el reloj del ordenador.

A continuación, se revisan los servicios disponibles. Se observa que además del servicio `Movement` se incluyen los servicios que permiten obtener y establecer el valor de los parámetros.

```
C:\Users\Alvaro\Documents\Practicas\ROS\capturas>ros2 service list -t
/movement [controller_interfaces/srv/Movement]
/movement_server/describe_parameters [rcl_interfaces/srv/DescribeParameters]
/movement_server/get_parameter_types [rcl_interfaces/srv/GetParameterTypes]
/movement_server/get_parameters [rcl_interfaces/srv/GetParameters]
/movement_server/list_parameters [rcl_interfaces/srv/ListParameters]
/movement_server/set_parameters [rcl_interfaces/srv/SetParameters]
/movement_server/set_parameters_atomically [rcl_interfaces/srv/SetParametersAtomically]
```

*Figura 69. Listado de servicios del sistema.*

Si se revisa la interfaz del servicio de movimiento se observan los campos definidos en el paquete de interfaces.

```
C:\Users\Alvaro\Documents\Practicas\ROS\capturas>ros2 interface show controller_interfaces/srv/Movement.srv
int64 x
int64 y
int64 z
int64 speed
---
string resp
```

*Figura 70. Interfaz del servicio de movimiento.*

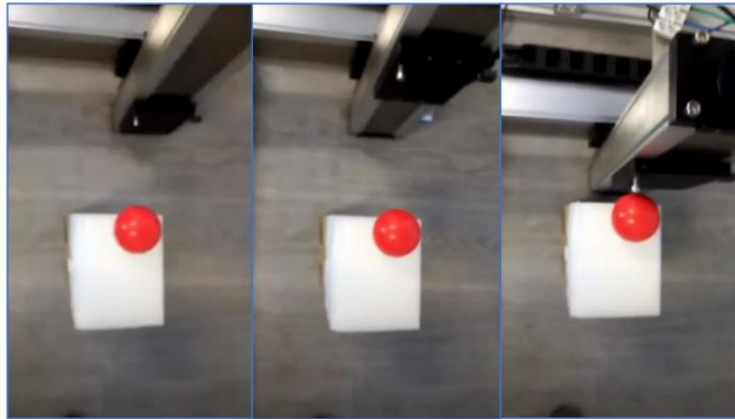
Conocido el nombre y tipo del servicio pueden ejecutarse peticiones asíncronas desde la línea de comandos como se muestra a continuación (el controlador se encuentra desconectado).

```
C:\Users\Alvaro\Documents\Practicas\ROS\capturas>ros2 service call /movement controller_interfaces/srv/Movement "{x:
50, y: 25, z: 145, speed: 150}"
waiting for service to become available...
requester: making request: controller_interfaces.srv.Movement_Request(x=50, y=25, z=145, speed=150)
response:
controller_interfaces.srv.Movement_Response(resp='Movement failed')
```

*Figura 71. Llamada al servicio de movimiento desde la línea de comandos (controlador desconectado).*

## 6.4. Prueba del sistema de visión artificial

En la prueba de funcionamiento para la detección de una bola se observó que el posicionamiento era bastante correcto, no obstante, la precisión del mapa de profundidad empeoraba de forma significativa para puntos lejanos al centro de la imagen, consecuencia del uso de haces de luz estructurada en su obtención. Un ejemplo de secuencia de movimiento se ilustra en la figura inferior.



*Figura 72. Ejemplo secuencia de movimiento tras detección.*

Se probó para diversas posiciones y alturas lográndose el resultado deseado. No obstante, la detección no es muy robusta, obteniendo resultados erróneos en el caso de que parte de la bola no fuera vista por la cámara.

Otra posible fuente de error es la colocación de la cámara que puede no ser perfectamente perpendicular a los brazos del robot.



## 7. Conclusión y proyectos futuros

### 7.1. Conclusión

Se ha logrado la realización de programas capaces de controlar un robot cartesiano y su integración en un sistema de ROS2 cumpliendo con ello los objetivos del proyecto.

Debido a la compatibilidad de ROS permitirá la fácil portabilidad del código realizado a otros sistemas y emplearlo en otro tipo de máquinas. No obstante, solo podrá ser utilizado en sistemas Windows debido al uso de las funcionalidades de Winapi32 en la construcción de la clase.

Se ha mostrado además que el sistema robótico puede ser usado en aplicaciones de diversa índole, como ejemplo práctico se empleó en un sistema de visión artificial empleando cámaras de profundidad.

Otro aspecto importante es que, debido en parte a la simplicidad y tamaño del trabajo realizado, no se ha podido apreciar gran número de características y funcionalidades que ROS2 puede ofrecer, sugiriendo el uso de otras soluciones capaces de obtener resultados similares en menos tiempo.

### 7.2. Proyectos futuros

Existen dos vertientes bien distinguidas para futuros proyectos. Una consistiría en la mejora de funcionalidades del software desarrollado, para permitir su uso por ejemplo en otro tipo de plataformas, introducción de nuevas funcionalidades (trazado de rutas 3D, permitir comunicación mediante acciones y temas, nuevas rutinas de tipos de movimiento) o uso de otro tipo de controladores.

La otra vertiente consistiría en el desarrollo del sistema que determina la posición del robot. Actualmente se está analizando la posibilidad del uso del robot en la poda de viñedos. El sistema determinará los puntos de corte que el robot debe alcanzar.

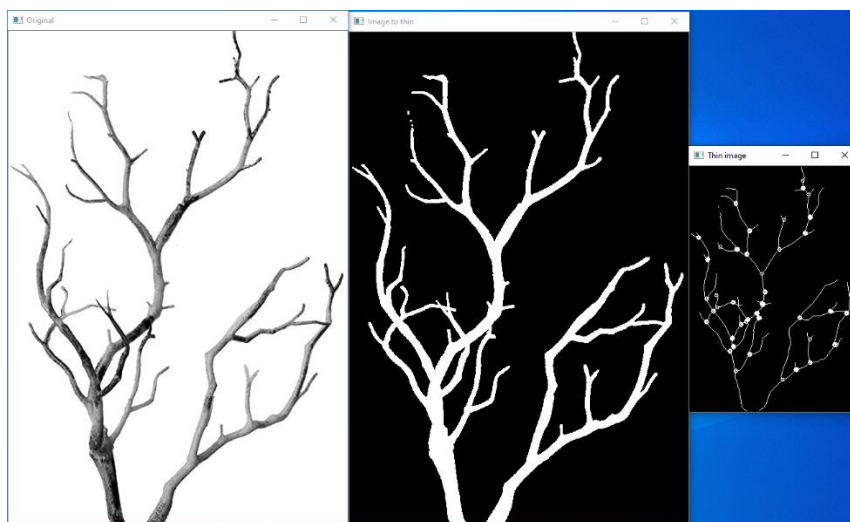


Figura 73. Visualización de puntos de corte obtenidos en la versión inicial.



Por supuesto ambas vertientes son complementarias, logrando así su posibilidad de uso en todo tipo de sistemas e incluyendo además la parte de detección de posición. Se observa aquí una de las capacidades de los sistemas de ROS, que es su gran escalabilidad, permitiendo incorporar nuevas partes o funcionalidades, reutilizando el código anteriormente realizado.

## 8. Bibliografía:

- [1] Fraser, S., 2009. *Pro Visual C++/CLI And The .NET 3.5 Platform*. Berkeley, CA: Apress.
- [2] Joseph, L., 2018. *Robot Operating System (ROS) For Absolute Beginners*. [New York, NY]: Apress.
- [3] Index.ros.org. 2020. *ROS 2 Overview*. [online] Available at: <https://index.ros.org/doc/ros2/>.
- [4] Docs.ansible.com. 2020. *YAML Syntax — Ansible Documentation*. [online] Available at: [https://docs.ansible.com/ansible/latest/reference\\_appendices/YAMLSyntax.html](https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html).
- [5] Fuyumotion.com. 2020. *Linear Motion Guide, Linear Axis Robotic Arm, Cartesian Robot Manufacturers, Motion Controller Suppliers - FUYU Technology Co., Ltd.* [online] Available at: <https://www.fuyumotion.com/>.
- [6] Components101.com. 2020. *Components101 - Electronic Components Pinouts, Details & Datasheets*. [online] Available at: <https://components101.com/>.
- [7] Docs.microsoft.com. 2020. *Technical Documentation, API, And Code Examples*. [online] Available at: <https://docs.microsoft.com/>.
- [8] GitHub. 2020. *J95zh/AMC4030-Qt*. [online] Available at: <https://github.com/j95zh/AMC4030-Qt>.
- [9] GitHub. 2020. *Meminyanik/AMC4030\_Matlab\_Controller*. [online] Available at: [https://github.com/meminyanik/AMC4030\\_Matlab\\_Controller](https://github.com/meminyanik/AMC4030_Matlab_Controller).
- [10] Design.ros2.org. 2020. *Why ROS 2?*. [online] Available at: [https://design.ros2.org/articles/why\\_ros2.html](https://design.ros2.org/articles/why_ros2.html).
- [11] Mazzari, V., 2020. *ROS Vs ROS2*. [online] Génération Robots - Blog. Available at: <https://blog.generationrobots.com/en/ros-vs-ros2/#:~:text=In%20ROS%20it%20is%20not,multiple%20nodes%20in%20a%20process.&text=In%20ROS%20roslaunch%20files%20are,complex%20logic%20like%20conditions%20etc>.
- [12] InfoQ. 2020. *Next-Gen Autonomous System Design Made Easier With DDS And ROS*. [online] Available at: <https://www.infoq.com/articles/ros2-dds-communication/#:~:text=Key%20Takeaways,securely%20as%20an%20integrated%20whole>.
- [13] GitHub. 2020. *Intelrealsense/Librealsense*. [online] Available at: <https://github.com/IntelRealSense/librealsense/tree/master/wrappers/labview>
- [14] Bradski, A., 2016. *Learning Opencv 3*. O'Reilly Media, Inc.
- [15] 3Dnatives. 2020. *Escáner De Luz Estructurada*. [online] Available at: <https://www.3dnatives.com/es/escaner-de-luz-estructurada-06122016>

## 9. Anexos

### 9.1. Instalación

En este anexo se documenta los pasos para poner en funcionamiento el software desarrollado.

- **EXTRACCIÓN DEL PAQUETE**

Tras la extracción del archivo comprimido tendrá una carpeta con los siguientes documentos.

Nombre	Fecha de modificación	Tipo	Tamaño
build	03/08/2020 17:04	Carpeta de archivos	
install	03/08/2020 17:04	Carpeta de archivos	
log	03/08/2020 17:04	Carpeta de archivos	
src	03/08/2020 17:04	Carpeta de archivos	
AMC4030.dll	01/10/2019 21:49	Extensión de la ap...	50 KB
config.txt	27/07/2020 10:28	Documento de te...	1 KB
Controlador_AMC4030.exe	27/07/2020 8:04	Aplicación	188 KB
GUI.exe	27/07/2020 10:31	Aplicación	376 KB
Readme.txt	27/07/2020 11:01	Documento de te...	1 KB

Figura 74. Directorios del paquete extraído.

El programa Controlador\_AMC4030.exe es el programa de control que emplea ROS2 y no está pensado para ser usado por el usuario.

El programa GUI.exe constituye la interfaz gráfica y puede ejecutarla directamente.

El archivo AMC4030.dll es la librería de vínculos del controlador que dispone el fabricante.

El fichero config.txt es el fichero que almacena la última posición del robot y aunque en un principio no está pensado para ser modificado por el usuario, puede ser útil para la configuración inicial.

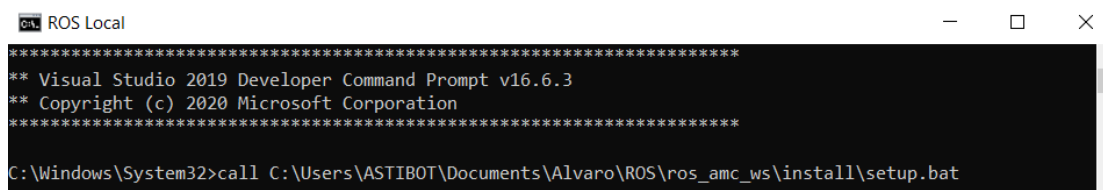
- **INSTALACIÓN DEL PAQUETE DE ROS**

Para que una consola de comandos pueda hacer uso de los comandos del paquete es necesario hacer una llamada al *bat* de inicialización (*setup.bat*).

Abra una consola de comandos e inicie ROS.

```
call C:\opt\ros\eloquent\x64\local_setup.bat
```

El comando puede variar en función de la distribución y directorio de instalación de ROS. Una vez hecho esto llame al archivo de inicialización del paquete. Éste se encuentra en el directorio install.



```
C:\Windows\System32>call C:\Users\ASTIBOT\Documents\Alvaro\ROS\ros_amc_ws\install\setup.bat
```

Figura 75. Llamada al archivo de inicialización del paquete.

Una vez hecho esto ya puede ejecutar los comandos del paquete para crear los nodos de cliente y servidor.

▪ **EJECUCIÓN DE LA INTERFAZ GRÁFICA**

Haga click en el programa GUI.exe, verá aparecer la ventana que se muestra a continuación.

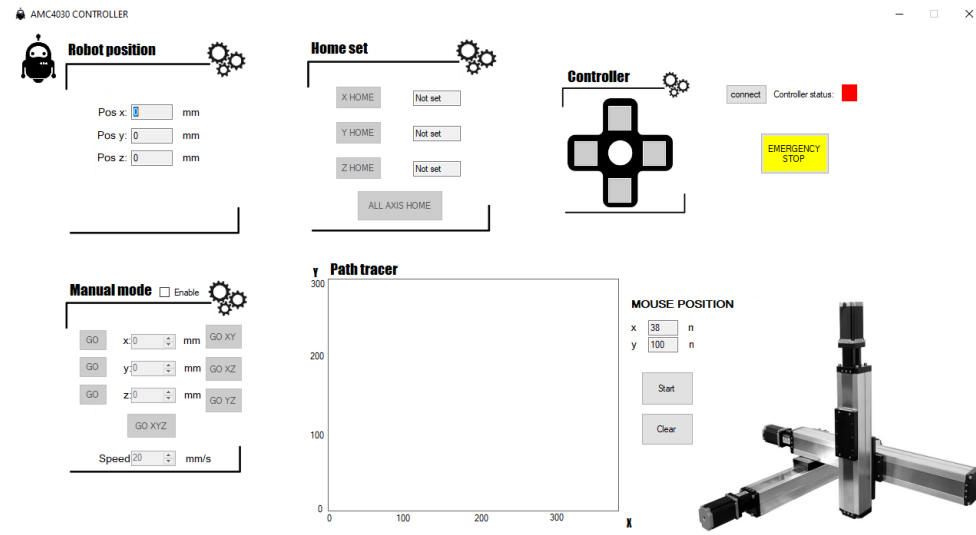


Figura 76. Ventana interfaz gráfica.

Si al pulsar el botón connect aparece la siguiente ventana de error:

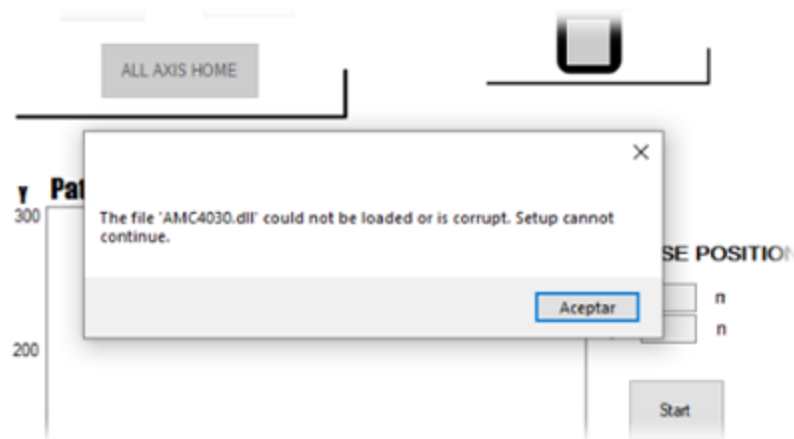


Figura 77. Error en interfaz gráfica al cargar DLL.

Asegúrese de que el paquete incluye el archivo AMC4030.dll. De no estar en el directorio descargado, puede encontrarlo en el software proporcionado por el fabricante en <https://www.fuyumotion.com/es/manual/>.

## 9.2. Índice de figuras

<i>Figura 1. Robot cartesiano de 3 grados de libertad. Fuente: <a href="https://base.imgix.net/">https://base.imgix.net/</a>.....</i>	<i>5</i>
<i>Figura 2. Diagrama de relación entre las partes del proyecto.....</i>	<i>6</i>
<i>Figura 3. Mecanismo interno FSL 120. Fuentes: [<a href="http://fuyumotion.com">fuyumotion.com</a>, <a href="http://barnesballscrew.com">barnesballscrew.com</a>].....</i>	<i>7</i>
<i>Figura 4. Configuraciones FSL 120. Fuente: [<a href="http://fuyumotion.com">fuyumotion.com</a>] .....</i>	<i>8</i>
<i>Figura 5. Esquema de motor bipolar. ....</i>	<i>8</i>
<i>Figura 6. Frontal del controlador AMC4030. Fuente [<a href="http://Fuyumotion.com">Fuyumotion.com</a>] .....</i>	<i>10</i>
<i>Figura 7. Conexión de final de carrera al controlador. Fuente [<a href="http://Fuyumotion.com">Fuyumotion.com</a>] .....</i>	<i>10</i>
<i>Figura 8. Esquema de conexión. ....</i>	<i>11</i>
<i>Figura 9. Esquema luz estructurada. Fuente [15] .....</i>	<i>11</i>
<i>Figura 10. Componentes del dispositivo Intel realsense SR300. Fuente: [<a href="https://www.mouser.com/pdfdocs/intel_realsense_camera_sr300.pdf">https://www.mouser.com/pdfdocs/intel_realsense_camera_sr300.pdf</a>].....</i>	<i>12</i>
<i>Figura 11. Arquitectura de la librería librealsense. Fuente [13] .....</i>	<i>12</i>
<i>Figura 12. Designación de uno de los motores.....</i>	<i>13</i>
<i>Figura 13. Captura de software AMC4030. Fuente [<a href="http://Fuyumotion.com">Fuyumotion.com</a>] .....</i>	<i>14</i>
<i>Figura 14. Captura de Visual Studio, plantilla CLR Console Application. ....</i>	<i>15</i>
<i>Figura 15. Captura de DLL Export Viewer. Fuente: [<a href="https://www.nirsoft.net">https://www.nirsoft.net</a>].....</i>	<i>16</i>
<i>Figura 16. Captura de Visual Studio installer. ....</i>	<i>19</i>
<i>Figura 17. Carpetas en el directorio de instalación. ....</i>	<i>20</i>
<i>Figura 18. Ejecución del comando ros2. ....</i>	<i>21</i>
<i>Figura 19. Diagrama comunicación entre nodos mediante temas. ....</i>	<i>22</i>
<i>Figura 20. Diagrama de comunicación mediante servicios. ....</i>	<i>24</i>
<i>Figura 21. Diagrama de comunicación por medio de acciones. ....</i>	<i>26</i>
<i>Figura 22. Definición de imagen.....</i>	<i>33</i>
<i>Figura 23. Ejemplo de aplicación filtro gaussiano 5x5 y filtro derivativo Sobel horizontal 3x3 Fuentes: [<a href="http://docs.opencv.org">docs.opencv.org</a>] y [<a href="http://es.mathworks.com">es.mathworks.com</a>] .....</i>	<i>35</i>
<i>Figura 24. Ejemplo transformada de Hough. Fuente: [<a href="http://www.researchgate.net">www.researchgate.net</a>] .....</i>	<i>35</i>
<i>Figura 25. Mapa de disparidad de un cubo. Los negros son puntos más próximos. Fuente: [<a href="http://en.wikipedia.org">en.wikipedia.org</a>].....</i>	<i>36</i>

<i>Figura 26. Modelo estenopeico. Fuente: [14]</i> .....	36
<i>Figura 27. Cambio de posición del plano imagen</i> .....	37
<i>Figura 28. Eliminación de distorsión radial. Fuente: [https://github.com/bbenligiray/lens-distortion-rectification]</i> .....	38
<i>Figura 29. Distorsión tangencial. Fuente: [www.researchgate.net]</i> .....	38
<i>Figura 30. Vista de funciones del fichero AMC4030.dll.</i> .....	41
<i>Figura 31. Fragmento de fichero ComInterface.h. Fuente: [https://github.com/meminyanik/AMC4030_Matlab_Controller]</i> .....	42
<i>Figura 32. Creación del tipo de los punteros a las funciones de la librería.</i> .....	42
<i>Figura 33. Declaración de punteros a las funciones de la librería.</i> .....	43
<i>Figura 34. Vista del modo diseño en visual estudio en proyecto c++/cli. Fuente: [https://social.msdn.microsoft.com/] .....</i>	44
<i>Figura 35. Captura de error en vista de diseño de Windows Forms.</i> .....	45
<i>Figura 36. Señalización del especificador de eventos.</i> .....	45
<i>Figura 37. Estructura de función de evento</i> .....	46
<i>Figura 38. Distintas partes que componen la interfaz gráfica.</i> .....	47
<i>Figura 39. Captura de funcionamiento de interfaz.</i> .....	49
<i>Figura 40. Fragmento de código de parámetros y constantes en la interfaz.</i> .....	50
<i>Figura 41. Fragmento del código necesario para la creación de la imagen del robot.</i> .....	50
<i>Figura 42. Código de controlador de evento de "manual_check_Click"</i> .....	51
<i>Figura 43. Ejemplo uso de función de conversión.</i> .....	52
<i>Figura 44. Código de evento movimiento del ratón en el gráfico de rutas.</i> .....	52
<i>Figura 45. Programa controlador.</i> .....	53
<i>Figura 46. Estructura del servicio Movement.srv</i> .....	55
<i>Figura 47. CMakeLists del paquete controller_interfaces.</i> .....	55
<i>Figura 48. package.xml del paquete controller_interfaces.</i> .....	56
<i>Figura 49. package.xml del paquete cpp_srvcli</i> .....	57
<i>Figura 50. CMakeLists.txt del paquete cpp_srvcli</i> .....	57
<i>Figura 51. Código de nodo servidor.</i> .....	58

*Figura 52. Código del nodo cliente..... 60*

*Figura 53. Sistema robótico para ejemplo de visión artificial. .... 62*

*Figura 54. Código obtención de mapa de profundidad. .... 63*

*Figura 55. Imagen y mapa de profundidad en el ejemplo de la detección de la bola. .... 64*

*Figura 56. Modelo estenopeico. Un punto Q se proyecta en el plano imagen resultando en el punto q. Fuente [14]..... 65*

*Función 57. Función para obtener puntos en coordenadas de la cámara [m,m,m] a partir de puntos de imagen [pixel, pixel, m] ..... 66*

*Figura 58. Ejemplo detección de una bola en una de las regiones de interés. .... 67*

*Figura 59. Lanzamiento de nodo cliente. .... 68*

*Figura 60. Error en la creación del nodo cliente. .... 68*

*Figura 61. Lanzamiento del nodo servidor. .... 69*

*Figura 62. Llamada fallida al programa de control. .... 69*

*Figura 63. Funcionamiento normal. Consola superior nodo cliente, consola inferior nodo servidor. .... 69*

*Figura 64. Listado de nodos..... 70*

*Figura 65. Listado de parámetros. .... 70*

*Figura 66. Obtención y modificación de parámetros. .... 70*

*Figura 67. Comando para guardar fichero con parámetros de nodo servidor..... 70*

*Figura 68. Fichero con parámetros de nodo servidor. .... 70*

*Figura 69. Listado de servicios del sistema. .... 71*

*Figura 70. Interfaz del servicio de movimiento. .... 71*

*Figura 71. Llamada al servicio de movimiento desde la línea de comandos (controlador desconectado). .... 71*

*Figura 72. Ejemplo secuencia de movimiento tras detección. .... 72*

*Figura 73. Visualización de puntos de corte obtenidos en la versión inicial. .... 73*

*Figura 75. Llamada al archivo de inicialización del paquete. .... 77*

*Figura 76. Ventana interfaz gráfica. .... 77*

*Figura 77. Error en interfaz gráfica al cargar DLL. .... 77*

*Figura 78. Código de funciones auxiliares de visión. .... 82*



### 9.3. Código de funciones auxiliares en ejemplo de visión.

Se incluye a continuación el código de las funciones de le ejemplo visto en el apartado 5.4.4.

```
// Remove points further than threshold
cv::Mat remove_background(const cv::Mat &color_image, const cv::Mat &depth_map, double
threshold) {
    cv::Mat mask(depth_map.size(), CV_8U, cv::Scalar::all(0));
    for (int i = 0; i < mask.rows; ++i)
        for (int j = 0; j < mask.cols; ++j) {
            float value = depth_map.at<float>(i, j);
            if (value != 0 && value < threshold) mask.at<uchar>(i, j) = 255; //Sensor tag as
0 out of range points.
        }
    // Remove small regions points of the depth map.
    cv::morphologyEx(mask, mask, cv::MorphTypes::MORPH_OPEN, cv::Mat(), cv::Point(), 1);

    // Fill holes
    cv::Mat top_hat;
    cv::morphologyEx(mask, top_hat, cv::MORPH_BLACKHAT, cv::Mat(), cv::Point(), 3);
    mask += top_hat;

    std::vector<cv::Mat> mask_x3 = { mask, mask, mask };
    cv::Mat mask_3C;
    cv::merge(mask_x3, mask_3C);
    cv::Mat image_wb = color_image & mask_3C;
    return image_wb;
}

std::vector<cv::Rect> find_rois(const cv::Mat &image) {
    std::vector<cv::Rect> rois;
    cv::Mat img, img_edge, labels, stats, centroids;
    cv::cvtColor(image, img, cv::COLOR_BGR2GRAY);
    cv::threshold(img, img_edge, 0, 255, cv::THRESH_BINARY);
    int i;
    int nccomps = cv::connectedComponentsWithStats(img_edge, labels, stats, centroids);
    std::cout << "Found: " << nccomps << "components.\n";
    std::vector<cv::Vec3b> colors(nccomps + 1);
    colors[0] = cv::Vec3b(0, 0, 0); //background pixels remain black.
    for (i = 1; i <= nccomps; i++) {
        colors[i] = cv::Vec3b(rand() % 256, rand() % 256, rand() % 256);
        if (stats.at<int>(i - 1, cv::CC_STAT_AREA) < 10) {
            colors[i] = cv::Vec3b(0, 0, 0); //Small regions are painted black.
        }
    }
    img = cv::Mat::zeros(img.size(), CV_8UC3);
    for (int y = 0; y < img.rows; y++)
        for (int x = 0; x < img.cols; x++) {
            int label = labels.at<int>(y, x);
            assert(0 <= label && label <= nccomps);
            img.at<cv::Vec3b>(y, x) = colors[label];
        }

    for (int i = 1; i <= nccomps - 1; i++) {
        std::vector<int> stat = stats.row(i);
        if (stat[cv::CC_STAT_AREA] > 1000) {
            cv::Rect boundary = cv::Rect(stat[cv::CC_STAT_LEFT], stat[cv::CC_STAT_TOP],
stat[cv::CC_STAT_WIDTH], stat[cv::CC_STAT_HEIGHT]);
            rois.push_back(boundary);
        }
    }
    std::cout << "Number of regions of interest = " << rois.size() << std::endl;
    return rois;
}

// Ball = [x, y, radius]
std::vector<cv::Vec3f> ball_detect(const cv::Mat &image) {
    cv::Vec3f null_ball(0, 0, 0);
    if (image.empty()) {
        std::cerr << "The image is empty.\n";
        exit(EXIT_FAILURE);
    }
    cv::Mat ball_image = image.clone();
```

```
cv::Mat gray_image;
cv::cvtColor(ball_image, gray_image, cv::COLOR_BGR2GRAY);
cv::GaussianBlur(gray_image, gray_image, cv::Size(5, 5), 0, 0);
std::vector<cv::Vec3f> circunferences;

cv::HoughCircles(gray_image, circunferences, cv::HOUGH_GRADIENT, 1,
gray_image.rows/16, 100, 30, 1, 100);
if (circunferences.empty()) {
    std::cout << "Could not find any balls.\n";
    return circunferences;
}
else {
    std::cout << "Found ball\n";
    for (const auto &circunference: circunferences)
        cv::circle(image, cv::Point(circunference[0], circunference[1]),
circunference[2], cv::Scalar(255,0,0), 3);
    return circunferences;
}
}
```

Figura 78. Código de funciones auxiliares de visión.