UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Generación de trayectorias y evitación de obstáculos para el robot IRB120 en entorno Matlab"

Nicolás Blanco Fernández

2015

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Generación de trayectorias y evitación de obstáculos para el robot IRB120 en entorno Matlab"

Autor:	Nicolás Blanco Fernández
Universidad:	Universidad de Alcalá
País:	España
Profesor Tutor:	María Elena López Guillén

Presidente: D. Luciano Boquete Vázquez

Vocal 1: D. Carlos Andrés Luna Vázquez

Vocal 2: Dña. María Elena López Guillén

CALIFICACIÓN:.....

FECHA:.....

Agradecimientos

El agradecimiento más profundo y sentido va para mi familia. Sin su apoyo, colaboración y ánimo habría sido imposible llegar hasta aquí.

A mis padres Pilar y Nicolás, por su ejemplo de lucha y honestidad, por enseñarme a luchar por lo que de verdad se quiere y demostrarme que todo esfuerzo tiene su recompensa; a mi hermana Lucía por su compañía y apoyo en todo momento. Por ellos y para ellos, ¡gracias!

También quiero agradecer a mis compañeros, profesores y en especial a mi tutora Elena, por su motivación y entrega en el desarrollo de este proyecto, destacando su disponibilidad y paciencia al adaptarse a las circunstancias que se me presentaron estos últimos meses, lo que me facilitó poder trabajar con comodidad y sin presiones, disfrutando y aprendiendo de este proyecto y de su amplio conocimiento en el mundo de la robótica. Muchas gracias por esta gran experiencia y por todo lo que me habéis enseñado en esta gran Universidad.

Y finalmente no puedo olvidarme de una gran persona que por suerte la vida me ha dado a conocer, Alba. Gracias por el gran apoyo recibido durante todos estos años y hacerme tan agradable el día a día.

Contenido general

Resumen

En este proyecto se abordará el desarrollo de una aplicación que permita una comunicación eficaz y fluida con el brazo robótico IRB120 de ABB desde el entorno Matlab, posibilitando la generación de trayectorias definidas por el usuario a través de unos "puntos de paso" intermedios, así como la detección de nuevos obstáculos presentes en la trayectoria del robot y la planificación de nuevas trayectorias recalculadas para evitarlos.

La detección del área de trabajo se efectuará mediante el sensor Kinect. Para ello se interactuará con el robot y el sensor Kinect en entorno Matlab, implementando el desarrollo de una interfaz gráfica a través de la herramienta GUIDE que permita enviar los comandos al robot a través de un socket de red.

Palabras clave: Robot industrial, Matlab, RobotStudio, Kinect, detección de obstáculos, socket TCP/IP, generación de trayectorias.

Abstract

This Project will address the development of an application which will permit fluid and efficient communication with the robotic arm IRB120 of ABB through the Matlab settings, making possible the creation of well-defined paths for the user through some intermediate ¨crossing points¨, like the detection of new obstacles present in the robots path and the planning of new paths recalculated to avoid these obstacles.

The Kinect sensor will detect the necessary area of work. To do this the robot will interact with the Kinect sensor in the Matlab settings, implementing the creation of a graphical interface through the GUIDE tool, which allows commands to be sent to the robot through a network socket.

Resumen extendido

Este trabajo fin de grado parte del mundo de la robótica, el cual se encuentra en auge debido al continuo desarrollo de aplicaciones prácticas en los últimos años, no sólo en la industria, sino en otros diferentes campos de la sociedad, llegando incluso a los hogares. En nuestro caso, abordaremos un proyecto enfocado al ámbito industrial, sin olvidar la importancia de los operarios humanos que puedan acceder al entorno de trabajo del robot para realizar sus tareas en colaboración al proceso de fabricación.

Para ello complementaremos el mundo de la robótica con el de la visión artificial, que ha tenido un gran desarrollo en los últimos años, más aún en el ámbito de la industria. Esta colaboración permitirá que el robot tenga en cuenta lo detectado mediante visión artificial para que sus movimientos sean más inteligentes y útiles en la práctica real.

Con todo esto, uno de los objetivos del presente trabajo consiste en el desarrollo de una aplicación que permita una comunicación eficaz y fluida con el brazo robótico IRB120 de ABB desde el entorno Matlab a través de un socket basado en el protocolo TCP/IP. Esto posibilita la ejecución de trayectorias definidas por el usuario a través de unos puntos de paso intermedios, haciendo uso de varias interfaces gráficas de comunicación creadas mediante la herramienta GUIDE de Matlab. El envío de dichas trayectorias del cliente al servidor se llevará a cabo mediante datagramas.

Otro de los objetivos a abordar, es la detección de nuevos obstáculos presentes en la trayectoria del robot y la planificación de nuevas trayectorias recalculadas para evitarlos. Para ello se utilizará el sensor Kinect, que posibilitará una detección del entorno de trabajo en la plataforma Matlab.

Tendremos en cuenta para ello la limitación del tiempo, debido a las necesidades de ejecución de la aplicación en tiempo real, tanto en la ejecución de la trayectoria de forma armónica y continuada, como en la evitación de obstáculos en cualquier instante mediante la regeneración de una nueva trayectoria. Para lograrlo, llevaremos a cabo una distinción de objetos de las nubes de puntos adquiridas mediante el sensor Kinect.

Finalmente se expondrán las conclusiones adquiridas, así como las futuras aplicaciones y trabajos derivados del mismo.

Lista de figuras

Lista de tablas

Memoria

1 Introducción

1.1 Antecedentes

El dominio del ser humano sobre la tecnología crece a pasos agigantados, de tal forma que se avanza hacia una automatización global.

La robótica es uno de los segmentos de la tecnología que, sin duda, mayor crecimiento ha experimentado en la última década y que más expectativas de desarrollo muestra. En cuanto a los robots manipuladores industriales, se ha podido contemplar el salto de un proceso semiautomático, con gran dependencia humana de operación y supervisión, a una actividad completamente automática, dotada de equipos auxiliares como la visión artificial o la manipulación robótica, generando un sistema rápido, preciso y eficiente, que supera de lejos las capacidades del ser humano y evita la ejecución de operaciones rutinarias o peligrosas.

Como se ha mostrado, la robótica en la actualidad juega un papel trascendental en la ingeniería, con enorme repercusión sobre la industria; por ello, la implantación de la asignatura de "Sistemas Robotizados" en el nuevo plan de estudios del Grado en Ingeniería en Electrónica y Automática Industrial. El Departamento de Electrónica ha

adquirido un brazo robótico industrial de la compañía ABB (robot IRB120), ideal para fines docentes. Esto posibilita al alumno el acceso a un entorno práctico de control de robots industriales, permitiendo la programación y automatización de los mismos.

Esta asignatura abarca conceptos de estudio como la morfología de un robot manipulador, los distintos métodos de representación espacial, el control cinemático, el control dinámico, la programación de robots y los criterios de implantación de un robot industrial, esenciales para la puesta en marcha de aplicaciones de automatización industrial.

Uno de los propósitos de este proyecto, es lograr un control fluido y en tiempo real del robot mediante una plataforma externa, en este caso Matlab, asistida del sensor Kinect para la detección del área de trabajo.

En la siguiente figura se puede observar de forma esquematizada los bloques principales a emplear en este proyecto.

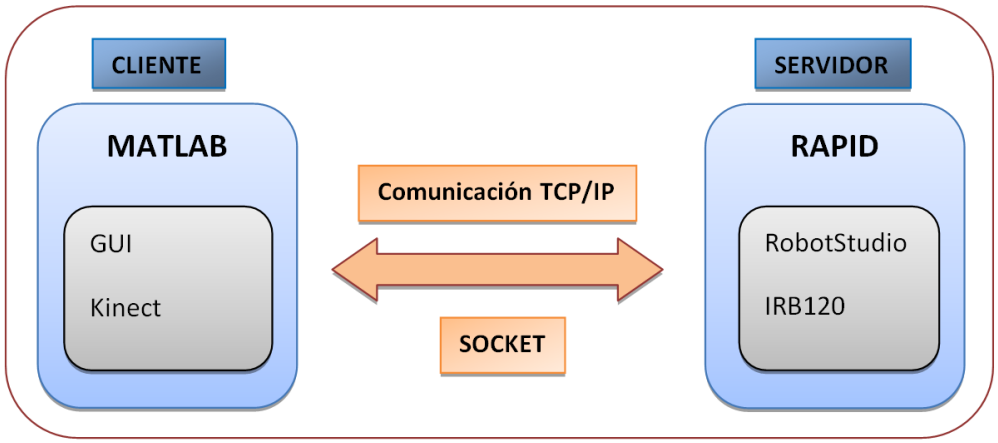


FIGURA 1.1 Esquema básico de los bloques principales del proyecto

Las bases para la realización del proyecto han sido las siguientes:

- Servidor: programa implementado en RAPID (lenguaje específico del software del robot "RobotStudio", que se ejecuta en su controlador IRC5). El punto de partida empleado es el proyecto titulado "Socket based communication in RobotStudio for controlling ABB-IRB120 robot. Design and development of a palletizing station", [1] realizado por Marek Jerzy Frydrysiak. Al cual se le han

realizado modificaciones de mejora generales y específicas para nuestra aplicación, que se justificarán y detallarán más adelante.

- Cliente: programa desarrollado en Matlab que permite la conexión con el servidor (RobotStudio) mediante un socket de comunicación con el protocolo TCP/IP. El punto de inicio es el proyecto titulado "Desarrollo de una interfaz para el control del robot IRB120 desde Matlab", [2] realizado por Azahara Gutiérrez Corbacho. El cual ha sido igualmente modificado para conseguir mejoras generales y específicas para nuestro programa.

Ambos proyectos mencionados anteriormente permiten una comunicación reducida entre cliente y servidor para las necesidades de nuestra aplicación, puesto que posibilitan el envío desde el cliente (Matlab) de un único punto a alcanzar por el brazo robot, mientras que nuestro propósito abarca una comunicación fluida que posibilite el envío de trayectorias completas al brazo robótico generadas por el usuario a través de varios puntos de paso, agregando la capacidad de interrupción a tiempo real tanto por el usuario como por un sensor externo que está a la escucha de estímulos, en este caso se emplea el sensor Kinect.

1.2 Objetivos del proyecto

El proyecto surge de la importancia de profundizar en el conocimiento de los sistemas robotizados, en este caso de los brazos robóticos industriales, y lograr una comunicación eficaz desde diversos entornos o plataformas de trabajo, permitiendo el desarrollo de aplicaciones que amplíen las posibilitadas por el fabricante.

El propósito primordial de este trabajo, es el desarrollo de una aplicación que permita la generación de trayectorias mediante el entorno Matlab, a través de unos puntos intermedios definidos por el usuario y la detección de la presencia de obstáculos en la trayectoria del robot mediante el sensor Kinect así como la planificación de nuevas trayectorias para evitarlos. Se implementará una interfaz gráfica mediante la herramienta GUIDE de Matlab, que permita la interactuación del usuario con el robot, proporcionando las distintas alternativas de la aplicación para el control adecuado tanto del brazo robot como del dispositivo Kinect.

A continuación se muestra un listado con los objetivos específicos de mayor relevancia desarrollados en este trabajo, incluyendo las mejoras en el servidor y el cliente:

- Generación de trayectorias en Matlab para su posterior envío al servidor a través del socket de comunicación.
- Distinción de obstáculos presentes en la trayectoria del robot a partir de las nubes de puntos adquiridas mediante el sensor Kinect y regeneración de nuevas trayectorias para evitarlos.
- Interrupción de trayectorias en ejecución en el servidor para su posible regeneración en tiempo real o detención del brazo robot.

El campo de aplicación puede ser muy variado y útil, ya que además de permitir el control del robot de forma remota a través del entorno Matlab, servirá como medida de seguridad tanto para el lay-out como para el propio robot al evitar colisiones.

1.3 Estructura de la memoria

A continuación se muestra una breve estructura de la memoria de este trabajo.

El primer apartado (en el que nos encontramos) presenta la introducción, muestra los antecedentes y los objetivos del proyecto.

El segundo apartado describe detalladamente cada una de las herramientas que se van a emplear: Robot industrial (ABB), socket de comunicación TCP/IP, el sensor Kinect y la herramienta de software matemático Matlab (indicando las diversas Toolbox empleadas).

El tercer apartado recoge las modificaciones realizadas sobre el socket de comunicación TCP/IP necesarias para el desarrollo de la aplicación.

El cuarto apartado expone la implementación del sensor Kinect mediante Matlab en el área de trabajo del robot y la detección de obstáculos presentes en su trayectoria.

El quinto apartado explica los métodos empleados para generar las trayectorias a través de los "puntos de paso" definidos por el usuario y la planificación de nuevas trayectorias en caso de detectar la presencia de obstáculos en la trayectoria del robot.

El sexto apartado presenta el diseño de la interfaz gráfica de comunicación a través de GUI, así como las limitaciones halladas en su desarrollo y ejecución.

El séptimo apartado muestra las principales conclusiones sobre el sistema realizado y se proponen diferentes líneas de trabajo futuro.

Por último, se incluyen los apartados de manual de usuario del simulador desarrollado, pliego de condiciones, planos y presupuesto.

2 Herramientas utilizadas

En este apartado se muestra un breve resumen acerca de las herramientas, tanto software como hardware, empleadas para la realización de la aplicación.

2.1 Esquema general del sistema

A continuación se adjunta un diagrama básico de la comunicación que se va a llevar a cabo entre el robot, la Kinect y la plataforma Matlab. Posteriormente se tratará cada una de estas herramientas empleadas minuciosamente.

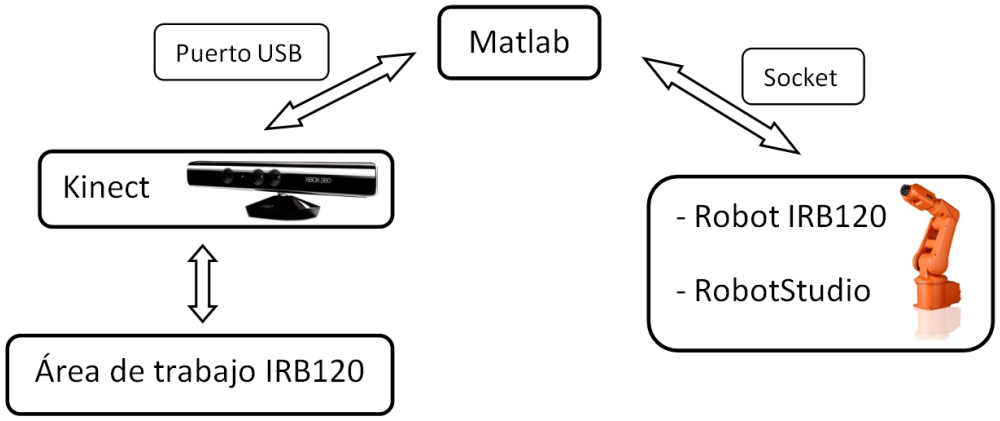


FIGURA 2.1 Esquema general del sistema

2.2 Robot industrial (compañía ABB)

ABB es una compañía líder global en tecnologías de automatización y energía con sede en Zurich (Suiza), cuenta con más de 150.000 empleados y está presente en más de 100 países. Tal éxito ha sido alcanzado esencialmente mediante la inversión en investigación y desarrollo en todas las condiciones de mercado.

Muchas tecnologías empleadas en la actualidad como la transmisión de energía de alto voltaje y determinados enfoques sobre la propulsión marina entre otros se desarrollaron o comercializaron por esta compañía. La compañía ABB es considerada como el mayor proveedor de drives y motores industriales, generadores para la industria eléctrica y redes eléctricas a nivel mundial.

Las operaciones de ABB están compuestas por cinco divisiones de negocio globales organizadas según el tipo de cliente: Power Products, Power Systems, Discrete Automation and Motion (parte de esta área se dedica a la robótica), Low Voltaje Products y Process Automation.

En cuanto a la robótica, ABB es el proveedor líder de robots industriales, sistemas de producción modular y servicios, habiendo instalado más de 250.000 robots en todo el mundo.

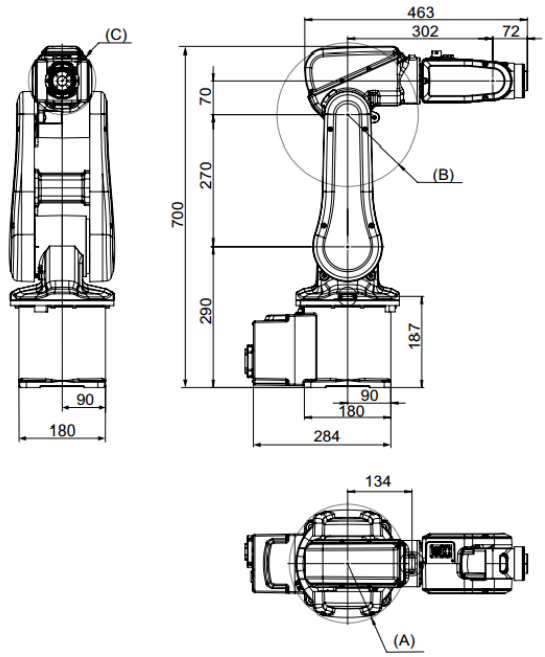
FIGURA 2.2 Robots industriales de ABB en cadena de producción

2.2.1 Brazo robótico IRB120

Para la ejecución de la aplicación desarrollada se requiere un brazo robótico. El robot industrial empleado es el modelo IRB120 de ABB. Es el más pequeño de la compañía, pesando tan solo 25 kg, de tipo articulado y con capacidad de soportar una carga útil de 3kg (4kg en posición vertical de la muñeca) y un alcance de 580 mm.

FIGURA 2.3 Brazo robótico IRB120

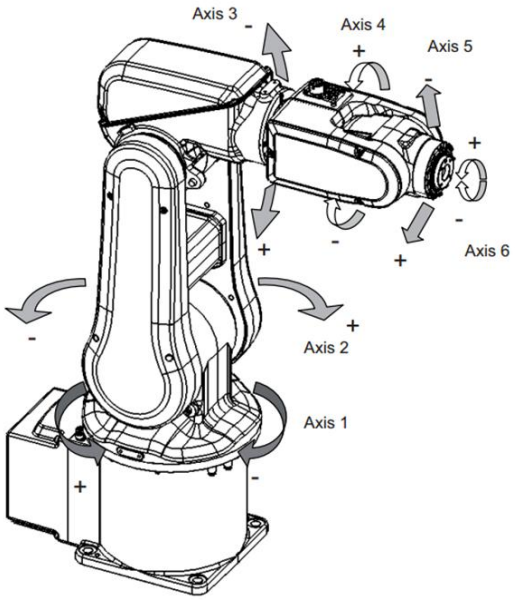
Como se ha mencionado, presenta un tamaño reducido en comparación con el resto de brazos robóticos que oferta esta misma compañía, aunque es más que suficiente para el objetivo que se persigue.



Pos	Descripción
A	Radio mínimo de giro del eje 1 R=121 mm
B	Radio mínimo de giro del eje 3 R=147 mm
C	Radio mínimo de giro del eje 4 R=70 mm

FIGURA 2.4 Dimensiones robot ABB IRB120

Este robot presenta 6 ejes, los tres primeros establecerán la posición del efector final (o herramienta) y los tres últimos la orientación del mismo.



Eje	Tipo de movimiento	Área de movimiento
Eje 1	Movimiento de rotación	De +165° a -165°.
Eje 2	Movimiento del brazo	De +110° a -110°
Eje 3	Movimiento del brazo	+70° a -110°
Eje 4	Movimiento de rotación	De +160° a -160°
Eje 5	Movimiento de doblado	De +120° a -120°.
Eje 6	Movimiento de giro	De +400° a -400°

FIGURA 2.5 Ejes manipulador IRB120

Por último la siguiente figura detalla el área de trabajo del robot, las posiciones extremas del brazo de robot se especifican respecto del centro de la muñeca (eje 5) con sus correspondientes dimensiones (en mm). Éstas también deben tenerse en cuenta al elegir el robot que mejor se adapte a nuestros requisitos, ya que pueden delimitar las posibles aplicaciones a realizar con un robot de este tipo.

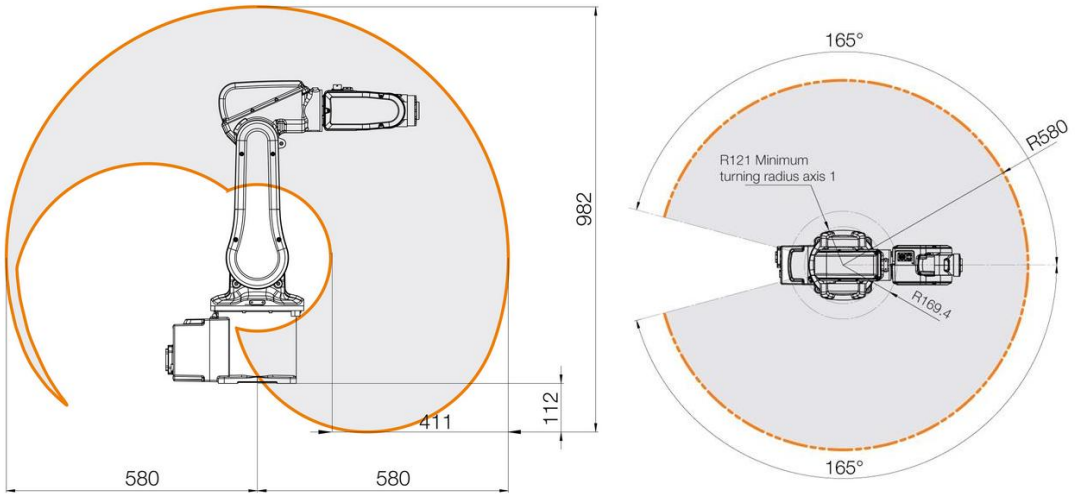


FIGURA 2.6 Limitación del espacio de trabajo del robot IRB120

2.2.2 RobotStudio (Software ABB)

RobotStudio es el software de simulación y programación offline de la compañía ABB, que permite crear y simular estaciones robóticas industriales en 3D en un ordenador con robots idénticos a los empleados en la instalación. Sus características incrementan la diversidad de tareas posibles para llevar a cabo mediante un sistema robótico, como la capacitación, la programación o la optimización.

Este software de simulación proporciona herramientas que aumentan la rentabilidad del sistema robotizado mediante tareas como formación, programación y optimización sin necesidad de interrumpir la producción, lo que genera grandes ventajas. Las principales son la reducción de riesgos, un arranque de mayor rapidez, una transición más corta, y un incremento en la productividad.

RobotStudio está basado en el VirtualController de ABB, que es una copia exacta del software real que emplean los robots reales, permitiendo simulaciones muy reales, con

archivos de configuración y programas de robots idénticos a los empleados en la instalación. Esto posibilita la simulación de la estación robótica diseñada sin riesgo alguno, previamente a una ejecución con el robot real, y evitar costes innecesarios.

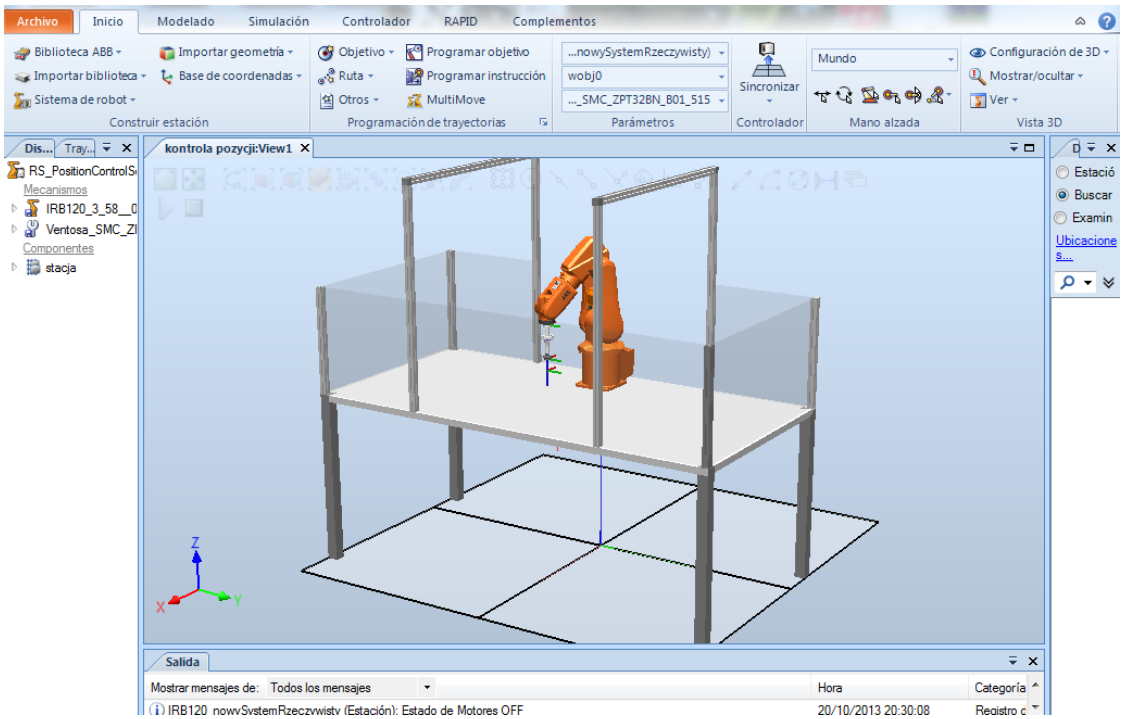


FIGURA 2.7 Interfaz principal del software RobotStudio

El software de ABB nos permite emplear diversas herramientas, entre las que se deben destacar las siguientes:

- **Importar CAD:** permite importar de forma simple la mayoría de los formatos CAD, incluyendo IGES, STEP, VRML, VDAFS, ACIS y CATIA.
- **Tablas de Eventos:** esta opción permite verificar tanto la estructura del programa como su lógica, posibilitando la visualización de los estados de E/S desde la estación de trabajo.
- **Detección de colisión:** es una herramienta orientada a entornos reales de trabajo. Previene posibles colisiones del robot con su entorno de trabajo durante la ejecución de programa.
- **Carga y descarga real:** todos los programas creados se pueden cargar de forma sencilla al sistema real sin que sea necesario emplear medios externos adicionales.

2.2.3 Lenguaje de programación RAPID

El lenguaje de programación empleado en dicho software es el lenguaje RAPID (Robotics Application Programming Interactive Dialogue), desarrollado por la compañía ABB para los sistemas de control S4 y IRC5 (este último será empleado en este proyecto, seguidamente se tratarán sus características). Este leguaje de programación de alto nivel, muy similar a otros lenguajes de propósito general, incluye características importantes, de las cuales cabe destacar las siguientes:

- Utilización de funciones y procedimientos.
- Posibilidad de uso de rutinas parametrizables.
- Posibilidad de declarar rutinas y datos como globales o locales.
- Estructura completamente modular del programa.

Los programas desarrollados en RAPID se denominan tareas. Una aplicación RAPID está formada por un programa y una serie de módulos del sistema. Por otro lado, el programa consta de una secuencia de instrucciones que describen el trabajo del robot, constando generalmente de tres partes:

- Rutina principal (main): se inicia la ejecución.
- Conjunto de sub-rutinas: para lograr una programación modular el programa se divide en partes más pequeñas.
- Datos de programa: definen posiciones, datos numéricos, sistemas de coordenada, etc.

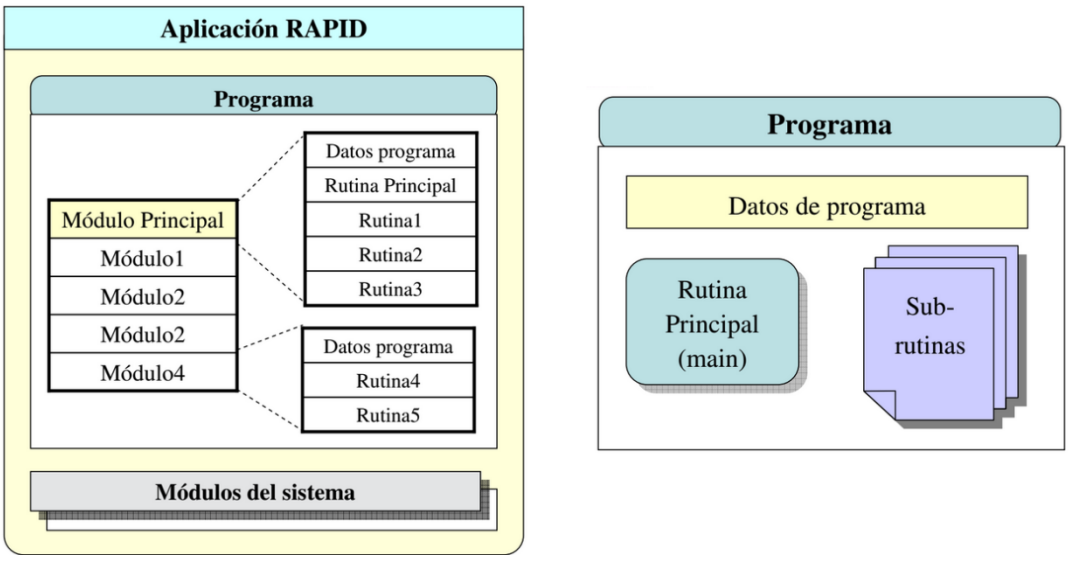


FIGURA 2.8 Estructura de una aplicación y un programa RAPID

Para cada dato, en este lenguaje debe definirse el ámbito de utilización y la persistencia.

Ámbito de utilización o alcance:

- GLOBAL: acceso permitido desde diferentes módulos de programa.
- LOCAL: acceso permitido desde un único módulo.

Persistencia:

- CONST: dato constante, no permite ser modificado.
- VAR: dato variables, puede ser modificado en el programa.
- PERS: dato persistente, en caso de ser modificado en el programa, permanece modificado para nuevas ejecuciones de programa.

2.2.4 Controlador IRC5 ABB y FlexPendant

El FlexPendant es un dispositivo que maneja múltiples funciones relacionadas con el uso del sistema de robot, entre ellas ejecutar programas, monitorizar el estado del manipulador, crear y editar programas de aplicación, etc. Es el encargado de comunicar al hombre con la máquina y viceversa.

Es un mando compuesto de elementos de hardware (botonera, pantalla táctil y joystick) y de software. El FlexPendant se conecta al controlador mediante un cable con conector integrado.

FIGURA 2.9 FlexPendant ABB

Dentro de la familia de controladores de ABB, se encuentra el controlador IRC5. Éste contiene todas las funciones necesarias para controlar y mover el robot, los ejes adicionales y los equipos periféricos. El controlador recibe y envía señales por medio de señales de entrada/salida y almacena programas.

Está disponible en diferentes tamaños y con distintas características dependiendo de las necesidades de la aplicación a realizar. En el proyecto, el controlador empleado para el robot IRB120 es el IRC5 Compact. Éste consiste en un único aparato de medidas 258x450x565 y 27.5 kg de peso, a continuación queda ilustrado.

FIGURA 2.10 Controlador IRC5 Compact

El controlador está provisto de los siguientes módulos:

Módulo de acondicionamiento: incluye el sistema de acondicionamiento que aporta la energía necesaria a los motores.

Módulo de control: contiene el ordenador, el panel de control, el interruptor de alimentación, las interfaces de comunicación, una tarjeta de entradas y salidas digitales, la conexión para el FlexPendant, los puertos de servicio y cierto espacio libre para equipos del usuario. El controlador también posee el software de sistema, es decir, que incluye todas las funciones básicas de manejo y programación (RAPID).

El flujo habitual de trabajo con el IRB120 consiste en desarrollar una aplicación en lenguaje RAPID, simularla en RobotStudio y cargarla en el controlador IRC5 para

poder ejecutarlo posteriormente en el robot real, aunque como se ha indicado previamente, en este proyecto planteamos la posibilidad de llevar a cabo un control externo a través de una comunicación mediante un socket desde un sistema remoto. Esto nos posibilitará ejecutar aplicaciones de mayor complejidad en las que se empleen sensores externos, para detectar el entorno de trabajo del robot y reaccionar ante las nuevas situaciones. En esta aplicación se empleará un sensor Kinect, dotado de una cámara de visión artificial y un sensor de profundidad infrarrojo, para la detección de obstáculos.

2.3 Socket TCP/IP

Como ya se ha mencionado, en el proyecto se abordará el control del robot IRB120 de ABB (servidor) desde el entorno Matlab (cliente), para implementarlo se genera la necesidad de una comunicación fluida y eficaz entre el servidor y el cliente, es decir, es indispensable realizar un socket de red para los protocolos de comunicación con el servidor.

Los sockets (denominados también conectores), son una interfaz de entrada-salida de datos, que permiten la comunicación bidireccional entre procesos que pueden estar ejecutándose en el mismo o en distintos sistemas, unidos mediante una red. El identificador de socket está compuesto por una dirección IP y un puerto.

En este proyecto, el socket de comunicación llevado a cabo sigue el protocolo TCP/IP, a continuación se muestran la justificación de esta elección y los distintos tipos de protocolos de transporte.

2.3.1 Comunicación TCP/IP

TCP/IP son las siglas de Transmission Control Protocol/Internet Protocol (Protocolo de Control de Transmisión/Protocolo de Internet), es un conjunto de protocolos estándar de la industria que está diseñado para redes de gran tamaño formadas por segmentos de red conectados mediante enrutadores. TCP/IP es la plataforma principal que sostiene Internet y que permite la comunicación entre diferentes sistemas operativos en

diferentes computadoras, ya sea sobre redes de área local (LAN) o redes de área extensa (WAN).

TCP: el Protocolo de Control de Transmisión permite establecer una conexión e intercambiar datos entre dos anfitriones. Este protocolo empaqueta datos en los paquetes que se envían en Internet y se vuelven a montar en sus destinos, garantizando que los datos no se pierdan durante la transmisión y que los paquetes sean entregados en el mismo orden de envío (FIFO).

IP: maneja la dirección y el enrutamiento de cada paquete de datos de modo que se envía al destino correcto. Utiliza direcciones que son series de cuatro números octetos (byte), con un formato de punto decimal. Ejemplo: 127.0.0.1 para red loopback (interfaz de red virtual).

Algunas ventajas más a destacar del protocolo TCP/IP son que funciona prácticamente sobre cualquier tipo de medio, sin importar si es una red Ethernet, una conexión ADSL o una fibra óptica; y que el direccionamiento que se asigna a cada equipo conectado posee una dirección única en toda la red a pesar de que sea tan extensa como Internet.

Los Protocolos de Aplicación como HTTP, HTTPs, SMTP, FTP y Telnet se basan y utilizan TCP/IP.

A continuación se muestran los posibles protocolos de transporte que permiten controlar el movimiento de datos entre dos máquinas:

- TCP (Transmission Control Protocol)
- UDP (User/Universal Datagram Protocol): permite el envío de datagramas a través de la red sin necesidad de establece una conexión previamente. No tiene confirmación ni control de flujo.
- IP (Internet Protocol): gestiona la transmisión actual de datos.
- ICMP (Internet Control Message Protocol): gestiona los mensajes de estado para IP, como errores o modificaciones en el hardware de red que afecten a las rutas.
- RIP (Routing Information Protocol): determinan el mejor método de ruta para entregar un mensaje.
- OSPF (Open Shortest Path First): abre primero el path más corto.

En este punto se plantea la posibilidad de emplear un protocolo TCP o UDP, a continuación se muestra una tabla comparativa entre ambos protocolos, con la correspondiente justificación.

TCP	UDP
Es un protocolo orientado a conexiones	Es un protocolo sin conexiones, un ordenador puede mandar paquetes sin establecer una conexión con el dispositivo que los va a recibir
Se usa para enviar mensajes por Internet de una computadora a otra mediante conexiones virtuales	Se usa para transporte de mensajes y/o transferencias. No está basada en conexiones, el programa puede enviar una carga de paquetes de datos y ahí finaliza esta relación
Útil en aplicaciones que necesitan alta confiabilidad y el tiempo de transmisión es menos crítico	Útil en aplicaciones que necesitan transmisión rápida. Al no establecer conexión, es útil en servidores con gran cantidad de peticiones pequeñas de un alto número de clientes
Reordena los paquetes de datos en el orden especificado	Los paquetes son independientes uno del otro, si requieren determinado orden, esto se maneja a nivel de aplicación
Es más lento puesto que ofrece garantía absoluta de que los datos transferidos llegan	Es más rápido, ya que no hace verificación de errores de paquetes, es decir, no hay garantía de que estos lleguen ni de que se envíen de forma duplicada

TABLA 2.1 Comparativa protocolos de transporte TCP y UDP

En nuestra aplicación, la pérdida de un dato al enviar un datagrama, puede generar un error al robot, indicándole una posición incorrecta o enviándole una de sus características principales de funcionamiento de forma equívoca.

Debido a la necesidad de elevada confiabilidad, sin que se pierda información en el envío, el protocolo seleccionado es el TCP/IP.

2.4 Kinect

El dispositivo empleado para la adquisición de imágenes y la detección del área de trabajo en este proyecto es el sensor Kinect de Microsoft, dotado de un hardware avanzado con diversos sensores.

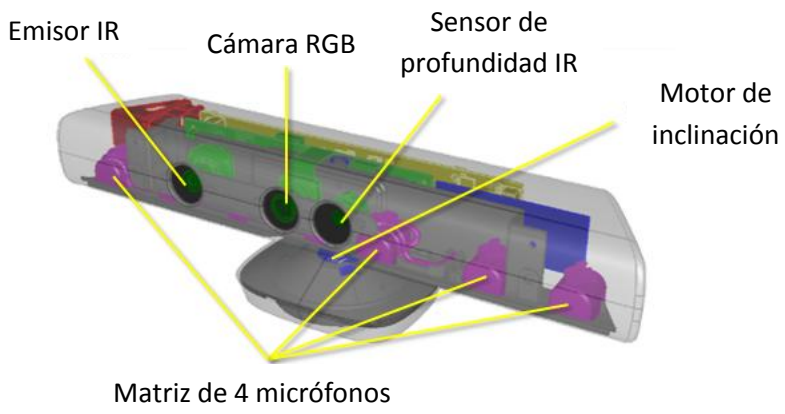


FIGURA 2.11 Sensor Kinect de Microsoft

- Cámara RGB que permite la adquisición de imágenes a color. Es capaz de almacenar tres datos del canal con una resolución de 1280x960 a una velocidad de 30 FPS (frames por segundo).
- Un emisor de infrarrojos y un sensor de profundidad infrarrojo que permiten la captura de una imagen de profundidad. El emisor emite rayos de luz infrarroja y el sensor de profundidad los lee a una velocidad de 30 FPS. Los haces reflejados se convierten en información de profundidad, midiendo la distancia entre un objeto y el sensor. Los datos de profundidad obtenidos se expresan en milímetros, medidos por la cámara como se muestra en la figura adjunta.

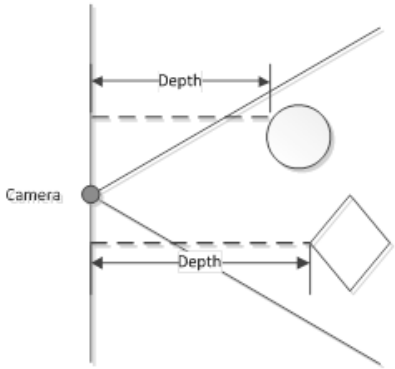


FIGURA 2.12 Datos de profundidad

- Matriz de cuatro micrófonos: además de grabar audio, debido a que hay cuatro micrófonos, es posible encontrar la ubicación de la fuente de sonido y la dirección de la onda de audio.
- Un acelerómetro de 3 ejes que permite determinar la orientación actual de la Kinect. Permite un ángulo de inclinación vertical de ±27º.

2.4.1 Drivers Kinect en Windows

Para poder desarrollar aplicaciones en Windows empleando el sensor Kinect debe instalarse el kit de desarrollo software de Windows (SDK).

Este kit incluye el controlador, el tiempo de ejecución, aplicaciones, interfaces de dispositivos, herramientas y el código necesario para el desarrollo de aplicaciones en Windows.

La última versión disponible[1] es SDK 2.0, compatible con Windows 8, Windows 8.1 y Windows Embedded 8. Debe comprobarse que la versión del kit instalado sea compatible con la versión de Windows del ordenador. En caso de tener problemas, pueden instalarse además los drivers kinectRunTime y Kinect Developer Toolkit compatibles.

2.5 Matlab

En este punto se tratará sobre el software matemático Matlab y las distintas funciones y herramientas empleadas para el desarrollo de este proyecto.

[1] Julio 2015

2.5.1 Generalidades

MATLAB (abreviatura de MATrix LABoratory, "laboratorio de matrices") es un lenguaje de alto nivel y entorno interactivo de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M). Esta herramienta de software matemático está disponible para las plataformas Unix, Windows, Mac OS X y GNU/Linux.

Entre sus principales prestaciones distinguimos la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. Todas estas funciones han sido empleadas en el desarrollo de nuestra aplicación. El paquete Matlab también dispone de la herramienta Simulink (plataforma de simulación multidominio), además pueden ampliarse las capacidades de Matlab mediante las toolboxes, y las de Simulink con los blocksets.

La última versión estable del software de Matlab es la R2014, que será empleada para la realización de este proyecto.

2.5.2 Desarrollo de interfaces gráficas GUI

La herramienta GUIDE, GUI Development Environment (entorno de desarrollo de GUI), proporciona las herramientas necesarias para diseñar las GUI, conocidas como interfaces gráficas de usuario o simplemente interfaces de usuario, que permiten un control sencillo (mediante el cursor del ordenador) de aplicaciones software personalizadas, lo cual suprime la necesidad de aprender un lenguaje y escribir comandos con el propósito de poder ejecutar una aplicación.

Las GUIs permiten la posibilidad de agregar controles como menús, cuadros de diálogo, barras de herramientas, representaciones gráficas, controles de interfaz de usuario (botones y controles deslizantes) y contenedores (como paneles y grupos de botones). Son programas autónomos de Matlab con un frontal gráfico de usuario GUI que automatizan una tarea o un cálculo. GUIDE genera de manera automática el archivo de extensión .M para controlar el lanzamiento e inicialización de la interfaz, el cual puede

ser modificado para definir las propiedades y los comportamientos de todos los componentes de la aplicación.

Para diseñar una interfaz gráfica efectiva, se deben seleccionar los objetos gráficos adecuados que se desean visualizar en la pantalla, distribuyéndolos de una manera lógica para facilidad del usuario, determinando adecuadamente las acciones que se realizarán al ser activados estos objetos por el usuario.

Los gráficos de la plataforma Matlab poseen una estructura jerárquica, compuesta por objetos de distintos tipos.

La siguiente figura esquematiza esta estructura jerárquica:

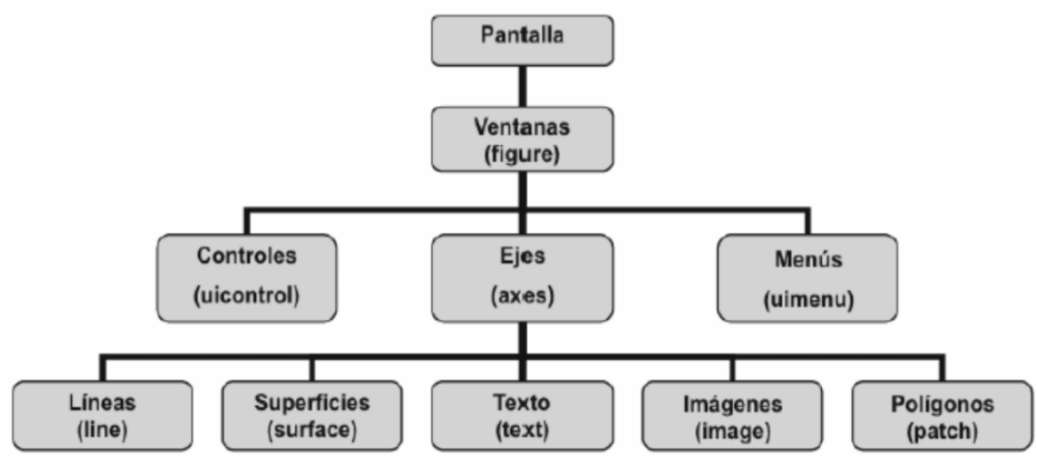


FIGURA 2.13 Jerarquía gráfica GUI en Matlab

Como puede observarse, hay objetos padres e hijos, es decir, si se elimina un objeto de Matlab, los objetos descendientes de éste son eliminados de forma automática.

Cada uno de los objetos posee un identificador único denominado "*handle*". Si una pantalla dispone de varios objetos, cada uno de éstos tendrá asignado un *handle*. El objeto raíz (pantalla) es siempre único y tiene identificador cero, mientras que el identificador de las ventanas es un entero y los de otros elementos son flotantes.

De este modo, Matlab permite tener varias ventanas abiertas, aunque únicamente hay una activa. De igual forma puede haber varios ejes en una misma ventana, pero sólo uno de ellos activo.

Los identificadores de la ventana activa, de los ejes activos, y del objeto activo, se pueden obtener con los siguientes comandos respectivamente: *gcf (get current figure), gca (get current axes), gco (get current object).*

Para abrir la herramienta GUIDE, se puede hacer doble clic sobre GUIDE en el Lanch Path, o simplemente tecleando desde el cuadro de comandos *guide*, abriendo la pantalla mostrada a continuación donde es posible crear una nueva GUI o abrir una ya existente.

Si se desea crear una GUI nueva, debe seleccionarse la opción "Blank GUI (Default)", apareciendo una ventana en blanco que servirá como editor de nuestro nuevo proyecto GUI. En esta podremos ubicar los objetos gráficos necesarios para nuestra aplicación. En otro caso, seleccionamos la pestaña "Open Existing GUI" para editar una GUI ya creada.

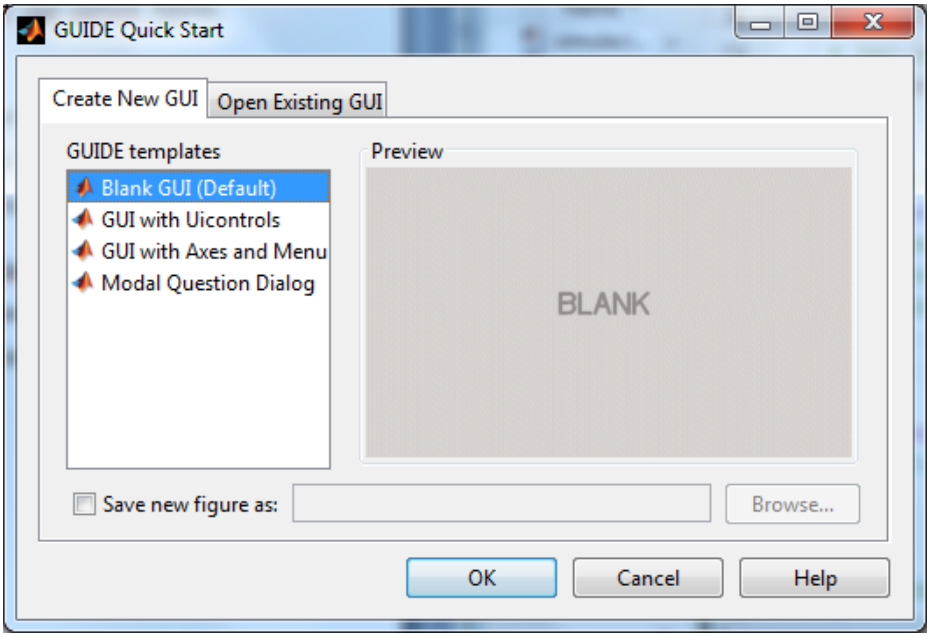


FIGURA 2.14 GUIDE Quick Start

Tras seleccionar la creación de una GUI nueva, el formato del área de trabajo donde vamos a desarrollar la aplicación es el siguiente:

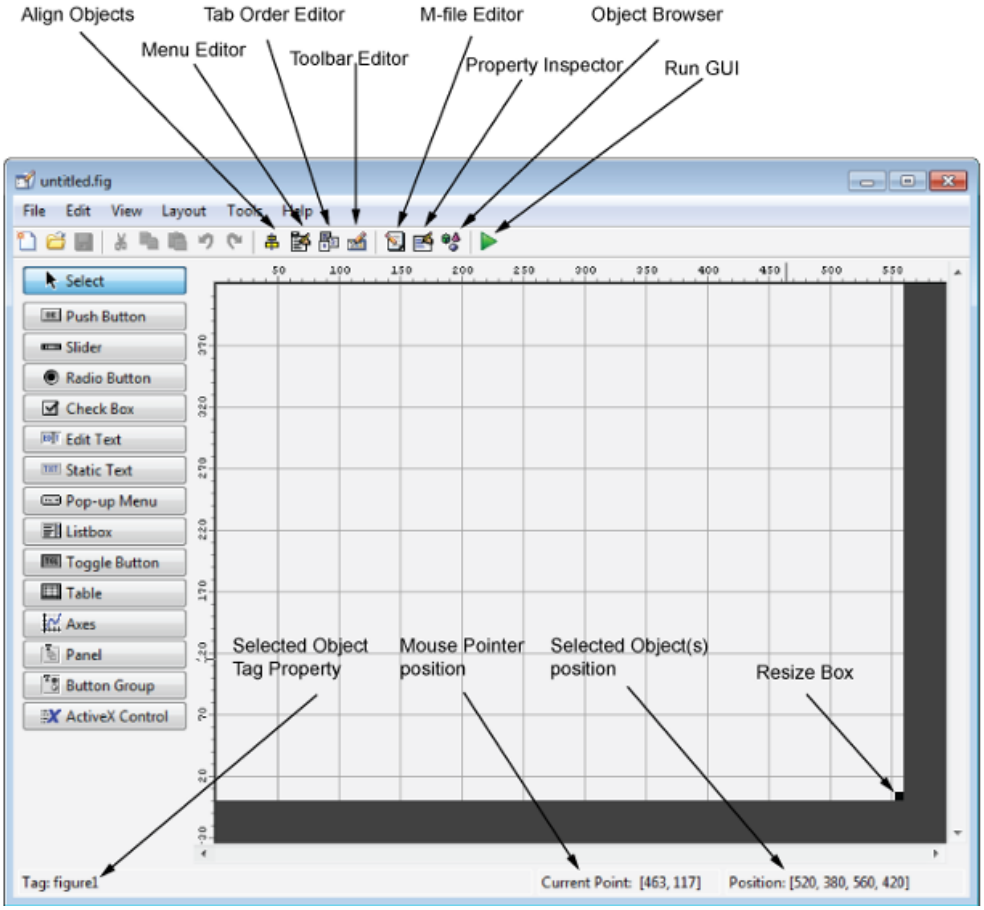


FIGURA 2.15 Blank layout area GUI

La siguiente tabla muestra una breve descripción de los componentes que se pueden insertar en el área de Layout denominados *Uicontrols* (User Interface Controls):

Control	Descripción
Check Box	Indica el estado de una opción o atributo, se activan pulsando sobre el cuadro y aparece un stick
Editable Text	Caja para editar y escribir cadenas de texto
Pop-up menu	Provee una lista de opciones para que el usuario de la aplicación pueda seleccionar alguna/s de ellas
List Box	Muestra una lista deslizable con varias opciones, donde el usuario deberá seleccionar una de ellas
Push Button	Invoca a un evento inmediatamente, tras ser pulsado por el usuario
Radio Button	Indica una opción que puede ser seleccionada
Toggle Button	Genera una acción e indica un estado binario (on/off)
Slider	Usado para representar un rango de valores numéricos. Se muestra

	una barra de carácter vertical u horizontal, la cual tiene asignados cuatro variables: valur, max, min y sliderstep
Edit Text	Sirve para modificar y escribir cadenas de texto
Static Text	Muestra un string de texto en una caja (no puede modificarse en ejecución)
Panel	Agrupa botones como un grupo
Button Group	Permite exclusividad de selección con los radio button
Axes	Permite la inserción de ejes y figuras en el GUI

TABLA 2.2 Uicontrols en GUI

Cuando salvamos la aplicación GUI creada, se generan automáticamente dos archivos, uno de extensión .FIG que contiene la descripción de los componentes que posee la interfaz, y otro de extensión .M que contiene las funciones y los controles del GUI así como el callback[2].

Se distinguen dos tipos de callbacks:

- *Callbacks para objetos gráficos:*
  Los objetos gráficos que son añadidos a la GUI tienen una serie de propiedades que permiten al programador desarrollar funciones callbacks para asociarlas a estas.
- *Callbacks para figuras:*
  Las figuras (*Axes*) tienen otras propiedades que pueden asociarse con sus respectivas funciones callbacks tras las correspondientes acciones del usuario.

2.5.3 Toolboxes empleadas

- **Robotics Vision & Control Toolbox:**

  La Toolbox de Robótica, creada por Peter Corke[3], supone un avance en el mundo del diseño y simulación robótica en Matlab.

[2] Un callback se define como la acción que llevará a cabo un objeto de la GUI cuando el usuario lo active, por ejemplo, en caso de existir un botón en una ventana y ser presionado por el usuario, se ejecutarán una serie de acciones, a eso se le conoce como la función callback.
[3] Véase Peter Corke, Abril 2014, Springer, "Robotics Toolbox for Matlab".

Proporciona gran variedad de funciones útiles para el estudio y simulación de robots de tipo brazo clásicos, como la cinemática, la dinámica o la generación de trayectorias. Permite la creación y diseño de brazos robóticos por el usuario, además proporciona funciones para la manipulación y conversión entre datos tales como vectores, transformaciones homogéneas y trabajo con cuaternios, necesarias para la representación de la posición y la orientación tridimensional del robot.

Esta Toolbox nos va a permitir reproducir un modelo del robot IRB120 de ABB con el que se llevará a cabo la aplicación y la generación de trayectorias.

- **Image Acquisition Toolbox:**

La Toolbox de adquisición de Imágenes, permite la adquisición de imágenes y vídeo desde cámaras y capturadores de fotogramas directamente en Matlab y Simulink, así como la modelación y simulación de sistemas de imágenes en tiempo real. Esta adquisición estará únicamente limitada por la velocidad de la cámara y el PC. Image Acquisition Toolbox tiene soporte para múltiples proveedores de hardware y estándares de la industria, detectando el hardware de forma automática y pudiendo configurar sus propiedades. Proporciona una interfaz consistente a través de los sistemas operativos, dispositivos de hardware y proveedores.

- **Image Processing Toolbox:**

La Toolbox de Procesamiento de Imágenes, proporciona un completo conjunto de algoritmos de referencia estándar, funciones y aplicaciones para el procesamiento de imágenes, análisis, visualización y desarrollo de algoritmos. Permite realizar análisis de imágenes, segmentación de imágenes, mejora de imagen, reproducción de ruido, transformaciones geométricas, y el registro de la imagen.

Soporta un conjunto variado de tipos de imágenes, incluidos los de alta gama dinámica, resolución de gigapixel, perfil ICC incrustado y tomográficas. Proporciona funciones de visualización y aplicaciones que permiten explorar las imágenes y vídeos, examinar una región de píxeles, ajustar el color y el contraste, crear contornos o histogramas y manipular regiones de interés.

- **Spline Toolbox:**

  La Spline Toolbox es una colección de funciones de Matlab para ajuste de datos, interpolación, extrapolación y visualización. Los Splines son una serie de polinomios cúbicos concatenados, escogidos de forma que exista continuidad en la posición y la velocidad del efector final del robot. En este trabajo se empleará dicha Toolbox para la interpolación cartesiana de la trayectoria del robot.

2.5.4 Interactuación con Kinect

Con el fin de confirmar las correctas instalaciones del sensor Kinect en Windows y de las respectivas Toolboxes necesarias en Matlab, para la adecuada disposición del sensor desde el entorno en el que vamos a trabajar, podemos ejecutar el siguiente comando desde la plataforma.

```
>> hwInfo = imaqhwinfo('kinect')

hwInfo =

       AdaptorDllName: 'C:\MATLAB\SupportPackages\R2014a\ki...'
    AdaptorDllVersion: '4.7 (R2014a)'
          AdaptorName: 'kinect'
            DeviceIDs: {[1]  [2]}
           DeviceInfo: [1x2 struct]
```

Para confirmar la correcta instalación, debe aparecer `DeviceIDs: {[1] [2]}` y `DeviceInfo: [1x2 struct]`. Esto indica que se ha leído la presencia de 2 sensores de imagen en nuestro dispositivo `'kinect'` (el de imagen y el de profundidad).

Si se desea obtener información específica de los dispositivos de imagen y profundidad, podrían ejecutarse los comandos `hwInfo.DeviceInfo(1)` y `hwInfo.DeviceInfo(2)` respectivamente. A continuación se muestra una secuencia de comandos que pueden servir de ayuda al usuario en el desarrollo de nuevas aplicaciones en las que se requiera una interactuación con la Kinect desde el entorno Matlab.

Inicialmente deben crearse los objetos de entrada de video para los flujos de color y de profundidad. La adquisición puede ser configurada en ambos flujos con una resolución de 640x480, 320x240 y 80x60, aunque el flujo de color también admite 1280x960. Para ello empleamos los siguientes comandos:

```
colorVid=videoinput('kinect',1,'RGB_1280x960');
depthVid=videoinput('kinect',2,'Depth_640x480');
```

Esto nos permitirá la captura de una simple imagen:

```
colorImage=getsnapshot(colorVid);
depthImage=getsnapshot(depthVid);
```

O la adquisición de flujos de imagen. Con la finalidad de adquirir ambos datos sincronizados debe usarse el disparo manual en vez de la activación inmediata (por defecto) ya que esta última sufre un retraso entre flujos mientras se realiza la adquisición. Posteriormente podemos ajustar la adquisición de frames por disparo suficientes para realizar el seguimiento e iniciamos el dispositivo de color y profundidad. Esto comienza la adquisición pero no registra los datos adquiridos, por lo que se debe comenzar posteriormente el registro y la obtención de los datos. Para finalizar la adquisición deben detenerse los dispositivos.

```
triggerconfig([colorVid depthVid],'manual');
colorVid.FramesPerTrigger=1;
depthVid.FramesPerTrigger=1;
start([colorVid depthVid]);
trigger([colorVid depthVid]);
[colorFrameData,colorTimeData,colorMetaData]=getdata(colorVid);
[depthFrameData,depthTimeData,depthMetaData]=getdata(depthVid);
stop([colorVid depthVid]);
```

3 Comunicación TCP/IP

Una vez justificada la elección del protocolo de comunicación TCP/IP en el apartado 2.3.1 debido a la garantía de entrega que este protocolo conlleva y que es necesaria en nuestra aplicación, se procede al estudio de las implementaciones llevadas a cabo del servidor y cliente, en RAPID y Matlab respectivamente.

A continuación se muestra un esquema básico de comunicación entre el cliente (Matlab) y el servidor (IRB120 o el simulador RobotStudio).

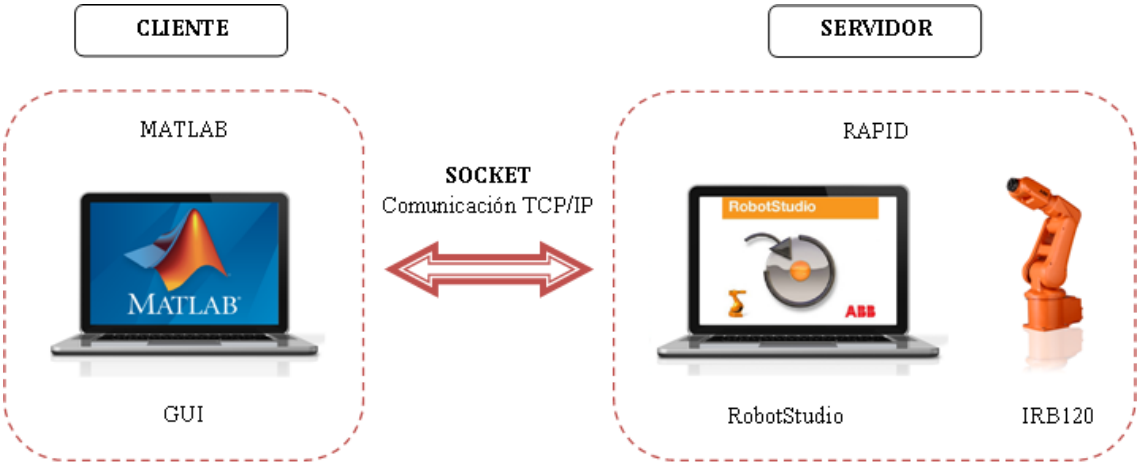


FIGURA 3.1 Esquema básico de comunicación cliente-servidor

3.1 Implementación del servidor (RAPID)

En este apartado se estudiará la implementación del servidor, es decir, el bloque derecho de la figura 2.10. El punto de partida de este proyecto, fue realizado por Marek Frydrysiak en su Trabajo Fin de Grado [1], aunque debido a sus limitaciones en la ejecución de aplicaciones en tiempo real, en las que se requiere una comunicación continua y fluida, se han realizado diversas modificaciones.

El Trabajo Fin de Grado [1] de Marek Frydrysiak permite una conexión con el cliente (Matlab), la recepción de puntos de alcance del robot mediante datagramas a través del socket, y su posterior desconexión. Esta aplicación permite mandar al robot un único punto, generándose cierto retardo en el envío del punto siguiente. En nuestro proyecto, son múltiples los puntos que deben ser enviados para lograr una ejecución en tiempo

real lo más eficaz posible, por lo que ha sido necesario realizar ciertas modificaciones en RAPID.

El código mostrado a continuación, localizado en el módulo "*TCPIPConnectionHandler*" muestra la secuencia de funciones del procedimiento "*EstablishConnection ()*" empleadas para crear el socket y establecer una conexión:

```
MODULE TCPIPConnectionHandler

VAR socketdev socketServer;
VAR socketdev socketClient;
VAR bool      connectionStatus := FALSE;
VAR string    data2ClientType;
LOCAL VAR num       initialVel{2} := [200, 200];
LOCAL VAR string    ipAddress;
LOCAL VAR num       portNumber := 1024;
LOCAL VAR num       err_counter := 0;

    PROC EstablishConnection()

        !============ Default values ==============
        ToolChoiceProcedure 1;
        VelocityChoiceProcedure initialVel;
        PrecisionChoiceProcedure 1;
        pos_limitation.trans.z := 0;  !Define the limitation for the Z
axis
        pos_limitation.trans.x := 0;  !Define the limitation for the X
axis
        !=========================================

        ipAddress := "127.0.0.1";

        SocketCreate socketServer;
        SocketBind socketServer, ipAddress, portNumber;
        SocketListen socketServer;
        SocketAccept socketServer, socketClient,
\ClientAddress:=ipAddress, \Time:=WAIT_MAX;
        connectionStatus := TRUE;

        Target_xyz := CRobT(\Tool:=tool \WObj:=wobj0);

        data2ClientType := "ONLINE";
        SendDataProcedure 0;
        SendDataProcedure 1;

      ERROR    !The timeout has been temporary disabled (see
\Time:=WAIT_MAX instruction above)
        IF ERRNO=ERR_SOCK_TIMEOUT THEN
            Incr err_counter;
            IF err_counter = 3 THEN
                ErrWrite "Fatal connection error", "Total waiting time
ERROR"
                    \RL2:="Connection re-try failure."
```

```
                    \RL3:="Maxiumum attempts to connect a client = 3";
                EXIT;
            ELSE
                ErrWrite \W, "Connection error", "Waiting time ERROR."
                    \RL2:="If the max. waiting time (120s) has been
exceeded"
                    \RL3:="the error handler returns the warning."
                    \RL4:="SOLUTION: Check if the client program is
connected to the socket.";
                RETRY;
            ENDIF
        ENDIF
    ENDPROC

ENDMODULE
```

Las variables del tipo "*socketdev*" son necesarias para el control de la comunicación con otro socket dentro de una red. Son variables de tipo no valor, por lo que es posible emplear más de 32 sockets simultáneamente. Este tipo de variables son utilizadas por las funciones de RAPID para la comunicación TCP/IP, como el envío o la recepción de datos, por ello han sido definidas como variables globales, para que cualquier módulo del programa pueda tener acceso a ellas.

Una vez declaradas las variables, el programa puede comenzar la conexión mediante el procedimiento "*EstablishConnection ()*". Éste utiliza las principales instrucciones para la creación y comunicación mediante sockets en RAPID, inicialmente crea un socket de conexión, posteriormente el socket es vinculado a una dirección IP y a un número de puerto determinado. En este instante el socket actúa como servidor, estando a la escucha de conexiones entrantes. Cuando se obtiene una petición de un cliente, ésta es aceptada, asociando una variable "*socketClient*".

Al ejecutarse el programa desde el inicio, en la función principal se hace referencia al procedimiento anterior para ser ejecutado y así crear y establecer la comunicación mediante el socket.

```
MODULE MainModule

      PROC main()

        EstablishConnection;
        DataProcessingAndMovement;

      ENDPROC

ENDMODULE
```

Como puede comprobarse, tras llamar al procedimiento "*EstablishConnection*" se hace referencia a la función "*DataProcessingAndMovement*". En esta última se trata el envío y la recepción de datos a través del socket, así como el control del movimiento del brazo robot.

```
VAR num contador_j:=1;
VAR num variable:=0;
PROC DataProcessingAndMovement()
        WHILE connectionStatus DO
            IF variable = 3 THEN
                stringHandler(receivedData);
            ELSE
            SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= WAIT_MAX;   !Receive data from the socket
            stringHandler(receivedData);  !Handle the received data
string
            ENDIF
        ENDWHILE
      ERROR
        IF ERRNO=ERR_SOCK_CLOSED THEN
            connectionStatus := FALSE;
            SocketClose socketServer;
            WaitTime 2;
            SocketClose socketClient;
            WaitTime 2;
            main;
        ENDIF
    ENDPROC
```

Como se puede observar, el procedimiento "*DataProcessingAndMovement*" estará a la escucha de recibir información mientras esté establecida una conexión.

Inicialmente, al tener la variable "*variable*" un valor de 0, se recibe la información de la red a través de la instrucción de RAPID "S*ocketReceive*" y se almacena en la variable global "*receivedData*" de tipo string. Posteriormente se llama a la función "*stringHandler*" para llevar a cabo la ejecución del robot.

El socket del servidor, desarrollado en código RAPID, se ha diseñado específicamente para recibir cada uno de los datagramas que se generan en el socket del cliente (Matlab). Este socket está diseñado para recibir numerosos tipos de datagramas, aunque únicamente haremos alusión al que se ha desarrollado para esta aplicación.

A continuación se detalla el tipo de datagrama enviado desde Matlab, para poder comprender la implementación de la función "*stringHandler*" (el desarrollo de dicho datagrama mediante código Matlab está expuesto en el apartado 3.2).

La limitación principal encontrada en el Trabajo Fin de Grado [1] para el desarrollo de nuestra aplicación, es el retardo generado en el envío de sucesivos datagramas del cliente al servidor y la posibilidad de envío de un único punto a alcanzar por el robot a través de un datagrama.

Debido a la necesidad de nuestra aplicación, que precisa del envío de numerosos puntos de alcance del robot, con la orientación y el posicionamiento del efector final adecuados, que definan una trayectoria y que sean ejecutados de forma armónica, se ha optado por desarrollar un nuevo datagrama que contenga datos con toda la sucesión de puntos de la trayectoria.

Este nuevo datagrama se encarga de enviar la información de la posición (ángulos) de los ejes del robot de los distintos puntos de la trayectoria. El programa desarrollado en RAPID identifica la cabecera (ID) de un datagrama entrante, y según su valor realiza una acción u otra. Dado que el robot IRB120 tiene seis grados de libertad, el datagrama contiene seis campos (J1-J6) para los valores absolutos (en grados) de los ángulos de las articulaciones y seis campos para sus correspondientes signos. Al enviarse numerosos puntos, el datagrama se le incluye un valor que indica el número de vectores de puntos que se ha introducido para identificarlos, ya que la función "*SocketReceive*" de RAPID encargada de leer el datagrama enviado por el cliente, almacena los datos en un *array of byte*, pudiendo gestionar un número máximo de 1024 bytes, el resto de campos del array por defecto valen 0.

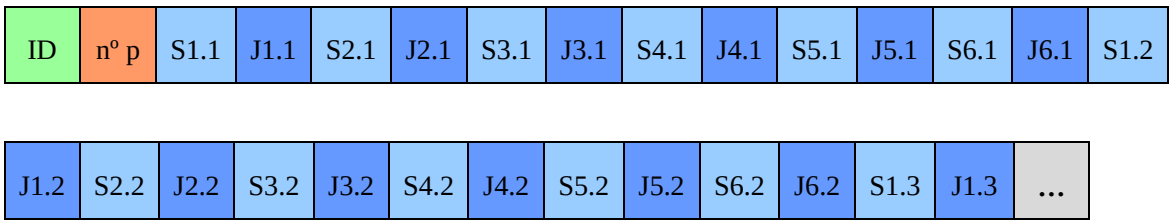


FIGURA 3.2 Estructura del Datagrama Cadena de Puntos Valores Articulares

Como puede observarse en la figura, todos los puntos que determinan la trayectoria del robot son enviados concatenados de forma simultánea en un mismo array, indicándose en un inicio los campos correspondientes al primer punto, posteriormente los correspondientes al segundo, y así sucesivamente. En la tabla siguiente se detalla el significado de cada campo.

ID	**Nombre del cuadro:** **Tipo de dato:** **Valor:**	Identificador del datagrama Byte 6
nº p	**Nombre del cuadro:** **Tipo de dato:** **Valor:**	Número de puntos Byte 1-84
S	**Nombre del cuadro:** **Tipo de dato:** **Valor:**	Valor del signo del siguiente valor Joint Byte 0 ó 1
Jn	**Nombre del cuadro:** **Tipo de dato:** **Valor:**	Valor absoluto del ángulo Joint Byte 0 - 255

TABLA 3.1 Descripción Datagrama Cadena de Puntos Valores Articulares

Tras haber comprendido el modo de envío de los puntos de la trayectoria del robot a través del datagrama, podemos acceder al estudio de la función RAPID "*stringHandler*" desarrollada en el módulo "*IRB120_Control*", encargada de abstraer dichos puntos y ordenar el movimiento de los ejes del robot a las distintas posiciones para reproducir la trayectoria.

```
CASE 6:

        IF contador_j<>1 THEN
        contador:=0;
        contador_j_aux:=contador_j;
        FOR j FROM contador_j_aux TO Data2Process{2} DO
            contador_j:=j;
            FOR i FROM 3+contador+((contador_j_aux-1)*12) TO
13+contador+((contador_j_aux-1)*12) STEP 2 DO
                Incr n;
                IF Data2Process{i} = 0 THEN
                    axMatrix{n} := -Data2Process{i+1};
                ELSEIF Data2Process{i} = 1 THEN
                    axMatrix{n} := Data2Process{i+1};
                ENDIF
            ENDFOR

            ax.rax_1 := axMatrix{1};
            ax.rax_2 := axMatrix{2};
            ax.rax_3 := axMatrix{3};
            ax.rax_4 := axMatrix{4};
            ax.rax_5 := axMatrix{5};
```

```
            ax.rax_6 := axMatrix{6};
            axis_pos := [[ax.rax_1, ax.rax_2, ax.rax_3, ax.rax_4,
ax.rax_5, ax.rax_6] , [ 0, 9E9, 9E9, 9E9, 9E9, 9E9]];

            !====== Security procedure (reachability) ======
            CheckAxesLimitations axMatrix;
            !===============================================

            IF statusAVAILABLE THEN
                ConfJ \Off;
                MoveAbsJ axis_pos, vel, z30, tool
            ELSEIF statusAVAILABLE = FALSE THEN
                SendDataProcedure 2;
            ENDIF

            contador:=contador+12;
            n:=0;
            variable:=1;
            SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= 0.001;

            IF variable = 1 THEN
                variable :=3;
                RETURN;
            ENDIF
        ENDFOR
        ENDIF

        contador:=0;
        FOR j FROM 1 TO Data2Process{2} DO
            contador_j:=j;
            FOR i FROM 3+contador TO 13+contador STEP 2 DO
                Incr n;
                IF Data2Process{i} = 0 THEN
                    axMatrix{n} := -Data2Process{i+1};
                ELSEIF Data2Process{i} = 1 THEN
                    axMatrix{n} := Data2Process{i+1};
                ENDIF
            ENDFOR

            ax.rax_1 := axMatrix{1};
            ax.rax_2 := axMatrix{2};
            ax.rax_3 := axMatrix{3};
            ax.rax_4 := axMatrix{4};
            ax.rax_5 := axMatrix{5};
            ax.rax_6 := axMatrix{6};
            axis_pos := [[ax.rax_1, ax.rax_2, ax.rax_3, ax.rax_4,
ax.rax_5, ax.rax_6] , [ 0, 9E9, 9E9, 9E9, 9E9, 9E9]];

            !====== Security procedure (reachability) ======
            CheckAxesLimitations axMatrix;
            !===============================================

            IF statusAVAILABLE THEN
         ConfJ \Off;
                MoveAbsJ axis_pos, vel, z30, tool
            ELSEIF statusAVAILABLE = FALSE THEN
                SendDataProcedure 2;
            ENDIF
```

```
            contador:=contador+12;
            n:=0;

            variable:=1;
            SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= 0.001;

            IF variable = 1 THEN
                !stringHandler(receivedData);
                variable :=3;
                RETURN;
            ENDIF
        ENDFOR
        variable:=3;
```

Inicialmente la variable "*contador_j*" vale 1, por tanto, al recibir un primer datagrama, el programa ejecuta directamente el segundo bucle *FOR* de la función, el cual abstrae un primer punto de la trayectoria y ordena la ejecución del movimiento del robot. Dado que la finalidad de la aplicación de este proyecto es la evitación de obstáculos en el área de trabajo del robot, el cliente puede enviar en cualquier instante una nueva trayectoria modificada para esquivar cualquier objeto, por lo tanto, en cada iteración se procesa una escucha en el socket durante 0.001 segundos (tiempo eficaz en la escucha e insignificante en la ejecución de la aplicación) generándose una interrupción en caso de recibir algún dato, y procediendo a una nueva ejecución del proceso.

En caso de detectar un obstáculo mediante la Kinect en Matlab que se interponga sobre la trayectoria normal del robot, la trayectoria que se regenera contiene el mismo número de puntos que la trayectoria inicial, pero con la altura suficiente para evitar el objeto:

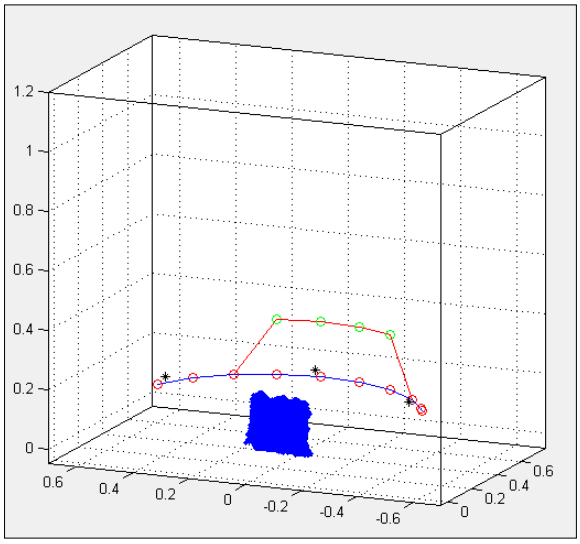


FIGURA 3.3 Trayectoria evitación de obstáculos

Como puede contemplarse en la figura 3.3, la trayectoria regenerada responde a un algoritmo, en el cual los puntos que generen una colisión del robot con el objeto detectado, son modificados con una altura suficiente para evitar el obstáculo, permaneciendo el resto de puntos con el mismo valor. Esto quedará explicado posteriormente de forma más detallada[4].

La variable "*contador_j*" almacena el número del orden del punto en la trayectoria, de tal forma que cuando se reciba un nuevo datagrama, el robot ejecute el correspondiente siguiente punto de la trayectoria.

Debido a la ejecución continuada del robot al recibir una trayectoria, se ha creado otro nuevo datagrama que responde a la detención del movimiento del robot, de modo que pueda estar a la escucha de continuar la reproducción de la trayectoria o recibir nuevas trayectorias durante un tiempo indefinido para ser ejecutadas.

```
        CASE 7:
        StopMove;
        SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= WAIT_MAX;   !Receive data from the socket
        StartMove;
```

De este modo se consigue una evitación de obstáculos realmente efectiva.

3.2 Implementación del cliente (Matlab)

En este apartado se estudiará la implementación del cliente, es decir, el bloque izquierdo de la figura 3.1. El punto de partida de este proyecto, fue realizado por Azahara Gutiérrez Corbacho en su Trabajo Fin de Grado [2], aunque debido a sus limitaciones en la ejecución de aplicaciones en tiempo real, en las que se requiere una comunicación continua y fluida, se han realizado diversas modificaciones.

El Trabajo Fin de Grado [2] de Azahara Gutiérrez Corbacho permite una conexión con el servidor (el robot), el envío de puntos de alcance al robot mediante datagramas a través del socket, y su posterior desconexión. Esta aplicación permite mandar al robot un único punto, generándose cierto retardo en el envío del punto siguiente. En nuestro proyecto, son múltiples los puntos que deben ser enviados para lograr una ejecución en

[4] Ver apartado 5.5 Evitación de obstáculos

tiempo real lo más eficaz posible, por lo que ha sido necesario realizar ciertas modificaciones en Matlab.

El método "*JointTrajectory*" mostrado a continuación, localizado en la clase "*irb120*" desarrollada por Azahara Gutierrez en su proyecto [2], muestra la función creada en el cliente para enviar los datos congruentes con el correspondiente código desarrollado en RAPID a través del socket. Esto será posible tras establecer una conexión adecuada con el servidor mediante la función "*connect*" (también desarrollada en este mismo módulo).

```
function JointTrajectory(r,vectores_q)

      D1=uint8([6 length(vectores_q)/6]); %Segunda pos
indicaremos el numero de vectores q enviados
      contador=3;
      contador_2=1;

      for i=1:(length(vectores_q)/6)
            vector_q(1:6)=vectores_q(contador_2:contador_2+5);
            contador_2=((i*6)+1);

            J1=uint8(round(abs(vector_q(1))));
            signo = sign(vector_q(1));
            if signo == -1
                  sj1 = 0;
            else
                  sj1=1;
            end

            J2=uint8(round(abs(vector_q(2))));
            signo2 = sign(vector_q(2));
            if signo2 == -1
                  sj2 = 0;
            else
                  sj2=1;
            end

            J3=uint8(round(abs(vector_q(3))));
            signo3 = sign(vector_q(3));
            if signo3 == -1
                  sj3 = 0;
            else
                  sj3=1;
            end

            J4=uint8(round(abs(vector_q(4))));
            signo4 = sign(vector_q(4));
            if signo4 == -1
                  sj4 = 0;
            else
                  sj4=1;
            end
```

```
            J5=uint8(round(abs(vector_q(5))));
            signo5 = sign(vector_q(5));
            if signo5 == -1
                  sj5=0;
            else
                  sj5=1;
            end


            J6=uint8(round(abs(vector_q(6))));
            signo6 = sign(vector_q(6));
            if signo6 == -1
                  sj6 = 0;
            else
                  sj6=1;
            end

            d1=[sj1 J1 sj2 J2 sj3 J3 sj4 J4 sj5 J5 sj6 J6];
            D1(contador:contador+11)=uint8(d1);
            contador=(i*12)+3;

      end

      %Enviamos el dato
      fwrite(r.conexion,D1);
      pause(1);
```

Como puede comprobarse en el código anexo, las dos primeras posiciones del datagrama corresponden al dígito 6 (para identificar el tipo de datagrama entrante ID) y al número de puntos a enviar a través del datagrama (número de vectores q) respectivamente. Dado que el robot IRB120 tiene seis grados de libertad, el datagrama contiene seis campos (J1-J6) para los valores absolutos (en grados) de los ángulos de las articulaciones y seis campos para sus correspondientes signos correspondientes con cada uno de los vectores q a enviar. Dichos vectores q se envían de forma concatenada al servidor en un mismo vector "*D1*" mediante la instrucción "*fwrite*".

4 Procesamiento de los datos de la Kinect

Point Cloud Library (PCL) es una biblioteca de código abierto a gran escala para el procesamiento completo de nubes de puntos 2D y 3D en los sistemas operativos Linux, Windows y Mac OS X. Esta biblioteca surgió ante la necesidad de que los robots tengan la capacidad de percibir el entorno tal y como lo podemos hacer los humanos, potencializándose debido al gran avance y al bajo costo de los sensores de adquisición de imágenes 3D, en el que cabe destacar el sensor Kinect de la compañía Microsoft. PCL contiene algoritmos útiles para la percepción en la robótica, destacando algunas herramientas como el filtrado de ruido, el reconocimiento de objetos, la creación de superficies a partir de nubes de puntos, etc. La reciente aparición de sensores capaces de tomar imágenes 3D, hace que el estado del arte de este tipo de imágenes tenga muy pocas herramientas. Otros software como ROS[5] disponen de este sistema de procesamiento de nubes de puntos, en cambio en Matlab no está desarrollado por el momento, por lo que ha sido necesario implementar un algoritmo que permita dicha adquisición de puntos con el sensor Kinect.

4.1 Adquisición de nubes de puntos en el espacio cartesiano del robot

Como se ha mostrado previamente en el apartado 2.4, el sensor Kinect presenta un emisor de infrarrojos y un sensor de profundidad infrarrojo que permiten la captura de una imagen de profundidad, de tal forma que los haces reflejados se convierten en información de profundidad expresada en milímetros.

Inicialmente debe llevarse a cabo una conversión entre la imagen de profundidad 2D al espacio tridimensional. Para ello recurrimos a las ecuaciones de óptica geométrica.

[5] Robot Operating System (Sistema Operativo Robótico)

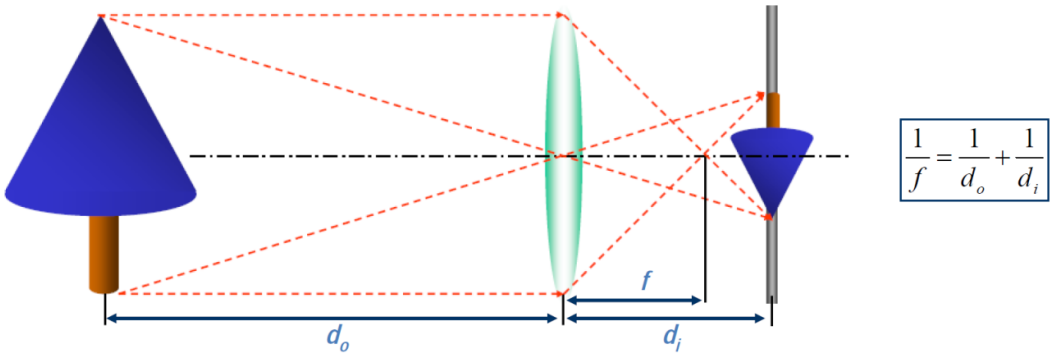


FIGURA 4.1 Fundamentos óptica

- ***f*** es la distancia focal (es función de la forma e índice de refracción de la lente) y en este caso de valor fijo puesto que el sensor Kinect no dispone de zoom.
- d_0 es la distancia del objeto hasta la lente.
- d_i la distancia desde la lente hasta el plano imagen.

La longitud focal de la cámara RGB es algo menor que la de la cámara IR, ofreciendo un mayor campo de visión.

	longitud focal (píxeles)	FOV (grados)
IR	580	57.8
RGB	525	62.7

TABLA 4.1 Longitudes focales y campo de visión sensor Kinect

Teniendo en cuenta la distancia focal en píxeles, aplicamos la ecuación de óptica geométrica para cada uno de los píxeles de la imagen obtenida.

Por otro lado realizamos una transformación del sistema de coordenadas por defecto del sensor Kinect (sistema de referencia cartesiano zurdo) al sistema de coordenadas real. El origen del centro de referencia es el centro de la cámara del dispositivo. El eje X y el eje Z conforman un plano horizontal de modo que el eje X se encuentra paralelo al lado más largo del dispositivo y el eje Z considera valores positivos a aquellos que se encuentren por delante del dispositivo. Por otro lado, el eje Y es vertical, tomando como valores positivos aquellos que se encuentren por debajo del dispositivo.

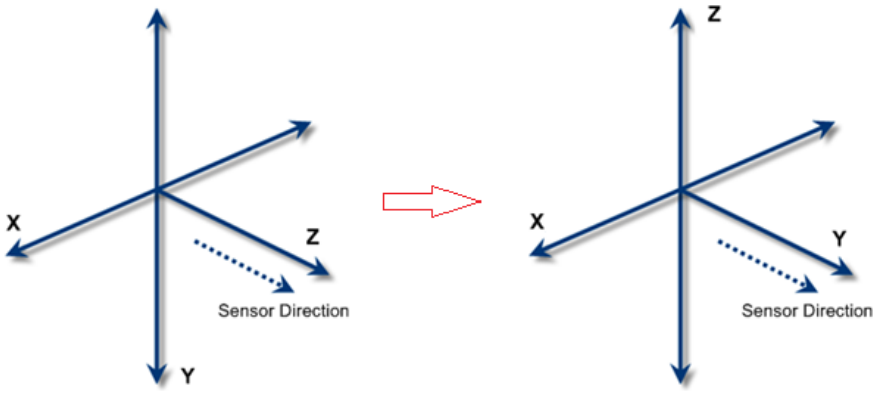


FIGURA 4.2 Sistema de coordenadas Kinect

El código desarrollado para obtener la conversión entre la imagen de profundidad 2D al espacio tridimensional es el siguiente:

```
function [x, y, z] = depthtoxyz(depth)

depth=double(depth);%Para poder realizar operación matricial
elemento a elemento
depth(depth==0)=nan;     %Tomar como not a number para valores
nulos para eliminarlos de la representación

%Datos camara IR
fx=580; %Distancia focal en el eje x en pixeles
fy=580; %Distancia focal en el eje y en pixeles
[v u]=size(depth);    %Numero de pixeles en eje y, eje x
respectivamente
centro=[u/2 v/2];    %Definimos centro en pixeles

matriz_pixeles_x=ones(v,1)*(1:u)-centro(1);    %Definimos valor
pixel matriz x
matriz_pixeles_y=(1:v)'*ones(1,u)-centro(2);  %Definimos valor
pixel matriz y

mm_to_m=10^-3;  %Unidades metros a mm

%Aplicamos ecuaciones de proyecciones de perspectiva
x_kinect=matriz_pixeles_x.*depth/fx*mm_to_m;
y_kinect=matriz_pixeles_y.*depth/fy*mm_to_m;
z_kinect=depth*mm_to_m;

%Cambio coordenadas kinect a reales
x=x_kinect;
y=z_kinect;
z=-y kinect;
```

A continuación se muestra un ejemplo que ilustra la transformación del plano imagen de profundidad al espacio cartesiano:

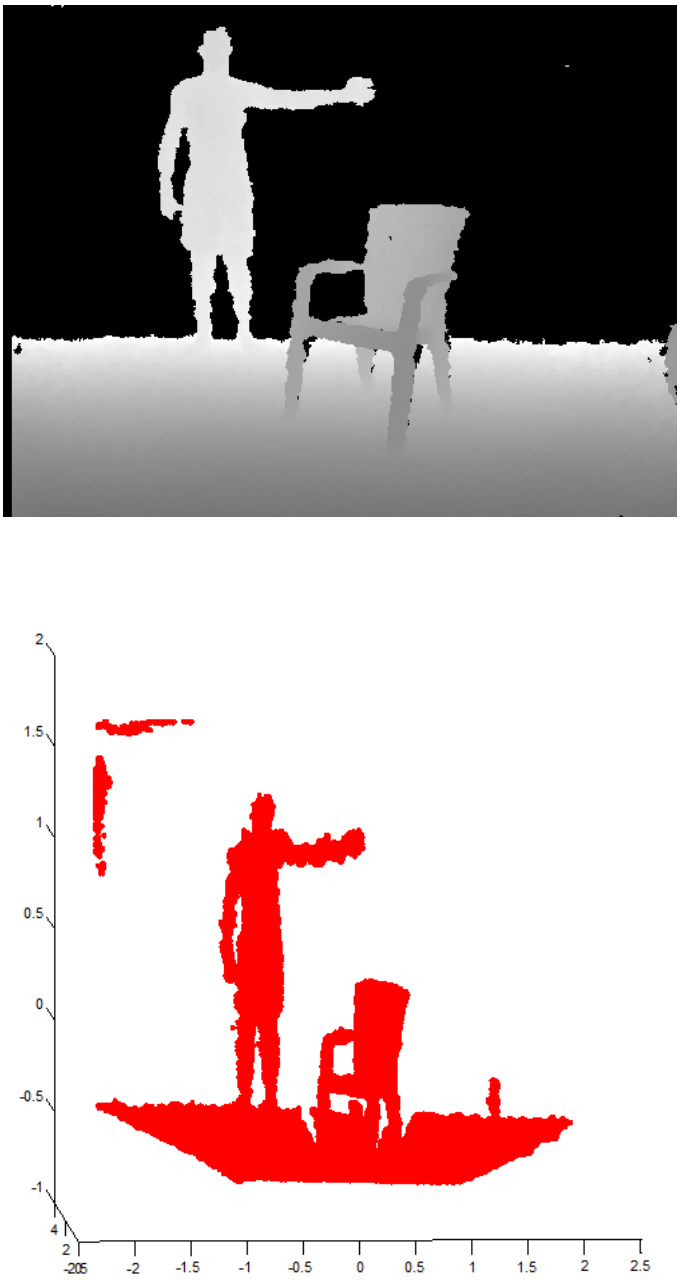


FIGURA 4.3 Ejemplo imagen de profundidad 2D a espacio tridimensional con Kinect

Dado que la generación de trayectorias se llevará a cabo respecto al sistema de coordenadas del brazo robot, para facilitar la programación de la aplicación, conviene conocer la posición de los obstáculos presentes en el área de trabajo del robot con respecto al propio robot en vez de respecto a la Kinect, por lo que se lleva a cabo un nuevo cambio de coordenadas del sensor Kinect al brazo robot. La posición del sensor Kinect con respecto al brazo robótico será conocida mediante un proceso de calibración

previo explicado en el apartado 4.2. Para proceder a este cambio de coordenadas se ha recurrido a la función *homtrans* de Matlab:

```
a=X';A=a(:)';
b=Y';B=b(:)';
c=Z';C=c(:)';
matriz=[A;B;C];
T=transl(0,y_max,0)*trotz(180,'deg')*transl(-((x_max-
x_min)/2+x_min),0,0);
abc=homtrans(T,matriz);
```

La función *homtrans* aplica una transformación homogénea mediante la matriz de transformación T, a los puntos introducidos en una matriz 3xN con coordenadas en x, y, z respectivamente. Dado que las matrices X, Y, Z tienen como dimensiones las de los píxeles de la imagen tomada por la Kinect, éstas deben ser transformadas en forma de vector para poder aplicar posteriormente dicha instrucción. La matriz resultante es de igual modo de 3xN.

```
>> size (X)

ans =

   480   640

>> size(abc)

ans =

           3      307200
```

Una vez lograda una representación espacial con respecto al sistema de coordenadas del brazo robot, se procede a una detección y diferenciación de obstáculos. Para ello se ha desarrollado un proceso iterativo mediante cálculos matriciales en Matlab, que detecta la presencia de aquellos obstáculos presentes sobre la superficie de trabajo del robot, eliminando el resto de objetos del entorno y el ruido que pueda generar la adquisición de puntos de la Kinect. El ejemplo mostrado a continuación ilustra esta detección y diferenciación de obstáculos presentes sobre la superficie de trabajo del robot:

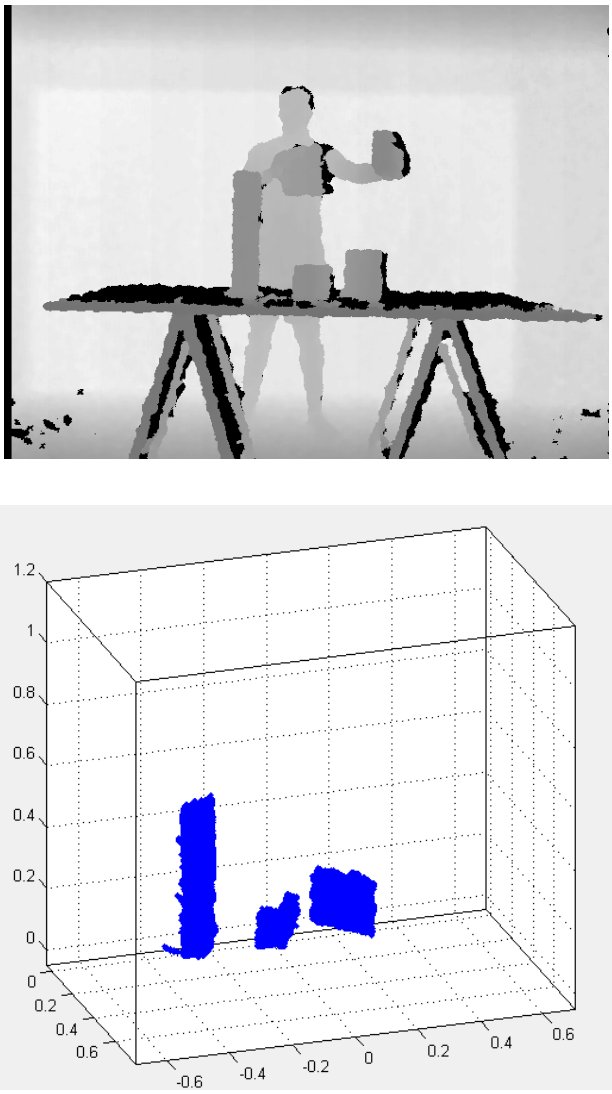


FIGURA 4.4 Diferenciación de obstáculos mediante sensor Kinect

4.2 Calibración y detección de blancos automática

Otro proceso relevante a conseguir con la Kinect es su adecuada calibración, esto nos permite conocer de la forma más exacta posible la posición del sensor Kinect con respecto al robot de forma automática, posibilitando a su vez la detección del área de trabajo del robot. Para ello se ha recurrido a un proceso de calibración automática que consiste en la detección y reconocimiento mediante la cámara RGB del sensor Kinect, de un blanco situado sobre un tablero de dimensiones conocidas que hemos construido artesanalmente. La diana debe estar ubicada en una posición concreta del área de trabajo del robot (a 20 cm de la base del robot). El usuario, con cliquear un único botón, será capaz de detectar con gran precisión el área de trabajo.

Dado que el blanco construido posee forma circular, para lograr este proceso, se ha hecho uso de la función *imfindcircles* de Matlab. Esta función detecta los círculos de una imagen dada, para un margen de radios y una sensibilidad con un rango de variación entre 0 y 1:

```
[centers, radii] = imfindcircles (colorImage, [7 13],
      'ObjectPolarity', 'dark', 'Sensitivity',0.92);
```

Para determinar el radio del círculo nos hemos ayudado de la función *imdistline*.

El blanco construido, está compuesto de una doble diana de colores rojo y verde. Se han elegido estos colores primarios puesto que es sencilla la diferenciación de sus características RGB y más eficaz que si se hubiera empleado un único color.

Tras la correcta detección de la diana, se procede a una captura de imagen de profundidad. Dado que las dimensiones de las imágenes RGB y de profundidad capturada poseen las mismas dimensiones (640 x 480), se emplea la posición del pixel de la diana en la imagen RGB, para ubicarse en la propia diana en la imagen de profundidad. Tras una serie de cálculos y ayudándonos de las dimensiones del tablero en que se encuentra la diana (20cm x 20cm), detectamos la posición de las dos esquinas superiores del tablero, siendo de este modo capaces de conocer la posición de la Kinect con respecto al brazo robot y detectar de forma eficaz el área de trabajo del robot.

A continuación se muestra un ejemplo donde se detecta la diana construida:

FIGURA 4.5 Calibración automática y detección de blancos mediante sensor Kinect

5 Generación de trayectorias y evitación de obstáculos

En este apartado se estudiarán los diversos procesos de generación de trayectorias desde el cliente (Matlab), para enviarlas al robot mediante datagramas a través del socket de comunicación para su posterior ejecución. Ya que como se ha expuesto previamente en el punto 3, la aplicación de comunicación desarrollada entre cliente y servidor, permite el envío de múltiples puntos en tiempo real para ser ejecutados por el brazo robot.

Tras desarrollar en Matlab el modelo cinemático y dinámico del IRB120 en la asignatura de "Sistemas Robotizados", se establecen distintas estrategias de control que redunden en una mayor calidad de sus movimientos.

El control cinemático establece cuáles son las trayectorias que debe seguir cada articulación del robot a lo largo del tiempo para lograr los puntos objetivo. En este caso, dichos puntos son fijados por el usuario, por lo que se recurre a la cinemática inversa, cuyo objetivo es encontrar los valores que deben adoptar las coordenadas articulares del robot para que su extremo se posicione y oriente según una determinada localización espacial.

El usuario del robot indica el movimiento que éste debe realizar especificando las localizaciones espaciales por las que debe pasar el efector final. Puesto que estos puntos están excesivamente separados, es preciso generar puntos intermedios suficientemente cercanos mediante interpolación, como para que el control del robot consiga ajustar no sólo el punto final al especificado, sino también la trayectoria seguida a la indicada en el programa. Para ello es preciso establecer un interpolador entre las localizaciones expresadas en el espacio de la tarea, que dará como resultado una expresión analítica de la evolución de cada coordenada. Para evitar las discontinuidades de velocidad en el caso de paso por varios puntos, pueden emplearse técnicas de interpoladores a tramos o interpoladores cúbicos para el caso de variables articulares.

Para implementar este proceso, se ha recurrido a dos técnicas de interpolación: la articular y la cartesiana (en el espacio de la tarea). En los apartados 5.2 y 5.3 se estudiarán ambas técnicas comparando los resultados obtenidos.

5.1 Orientación del efector final

La finalidad principal del programa desarrollado, es la generación de trayectorias por unos puntos de paso intermedios seleccionados por el usuario, sin una especificación concreta de la orientación del efector final. Por ello se ha implementado un programa que determina esta orientación del efector final, de modo que esté dirigido hacia el codo del brazo robot para cada uno de los puntos a alcanzar. Este programa es el encargado de generar la matriz de transformación homogénea.

La función desarrollada, denominada "*Calcula_T*", se muestra a continuación comentada para una mayor comprensión del código:

```
function T = Calcula_T(xp,yp,zp)

%Determinación del punto FOCO
foco=[0 0 0.56];

%Cálculo del vector de aproximación
a=[(xp-foco(1)) (yp-foco(2)) (zp-foco(3))];

%Normalización del vector de aproximación
a=a./norm(a);

%Cálculo de un vector perpendicular a z y a  (perpendicular al
plano que contiene a ambos)
v_plano=cross(a,[0 0 1]);

%Cálculo del vector n perpendicular al anterior y a "a"
n=cross(a,v_plano);

%Por último, el vector o es ortonormal con los otros dos
o=cross(a,n);

%Construyo la submatriz de rotación noa
R=[n' o' a'];

%Construyo la matriz de transformación homogénea
rotacion=r2t(R);
T=transl(xp,yp,zp)*rotacion;

end
```

Mediante la ejecución de la función anterior, se obtienen las matrices de transformación homogénea para cada uno de los puntos seleccionados por el usuario que conforman la trayectoria, es decir, la entrada a los interpoladores son los puntos de paso con orientación. A continuación se ilustra un ejemplo gráficamente:

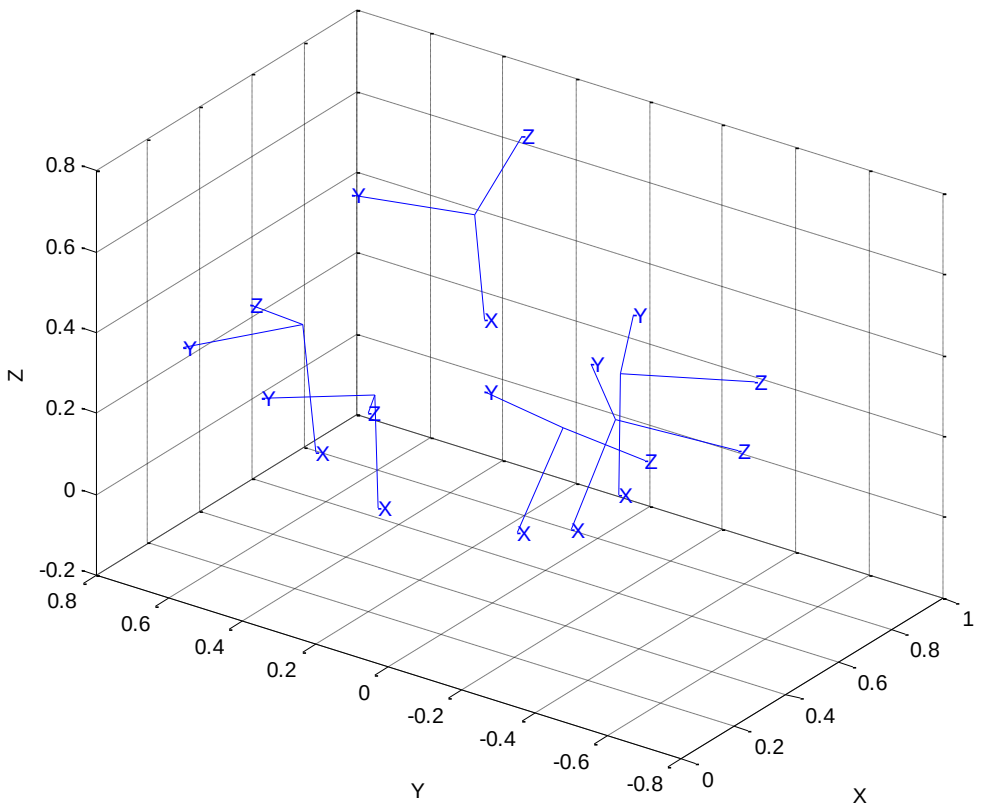


FIGURA 5.1 Matrices de transformación homogénea de los puntos de la trayectoria

Los resultados obtenidos en la ejecución del brazo para dos posiciones distintas se ilustran en las siguientes figuras, en ellas se puede comprobar la orientación del efector final respecto al codo del robot.

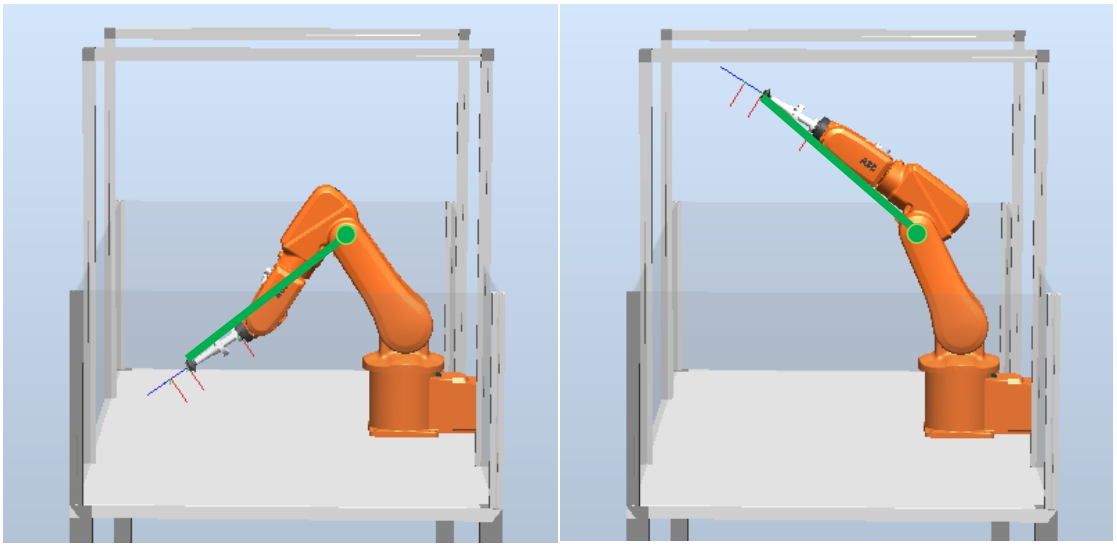


FIGURA 5.2 Orientación del efector final

5.2 Interpolación articular

Uno de los métodos empleados para la interpolación de los puntos de la trayectoria, es la interpolación articular. Este método consiste en la generación de la trayectoria de acuerdo al tipo de movimiento que tienen sus articulaciones, buscando la mayor suavidad posible en la ejecución de cada una de estas articulaciones.

Este proceso de interpolación articular se consigue mediante el cálculo de la cinemática inversa de las posiciones de paso del efector final, para obtener posteriormente las interpolaciones articulares. De este modo el movimiento de las articulaciones es suave, pero no el del efector final.

La función de Matlab que nos permite una interpolación multidimensional con puntos de paso es "*mstraj*". La trayectoria se genera con velocidades inicial y final nulas, pero pasa por (o cerca de) los puntos intermedios sin pararse. Para asegurar esto, el método utilizado por la función usa tramos lineales centrales de velocidad constante, aunque los tramos de aceleración y deceleración se implementan mediante polinomios quínticos para asegurar condiciones de contorno en la posición, velocidad y aceleración. Un aspecto importante de esta función es que utiliza un movimiento simultáneo y coordinado de todas las variables, ajustando la velocidad máxima de cada variable en cada tramo, de manera que todas ellas alcancen los puntos intermedios en el mismo instante de tiempo (dado por la variable más lenta).

Tras calcular la matriz de transformación homogénea T correspondientes a cada uno de los puntos fijados por el usuario y obtener la cinemática inversa de las configuraciones articulares q haciendo uso de las funciones desarrolladas "*Calcula_T*" y "*irb120_ikine*" respectivamente, se procede a la interpolación de las configuraciones articulares utilizando la función "*mstraj*".

En la función "*mstraj*", el primer argumento es la matriz con los puntos de paso (una fila por punto), el segundo un vector con las velocidades máximas, el tercero un vector de duración de cada segmento, el cuarto la posición inicial, el quinto el intervalo de muestro y el sexto el tiempo de aceleración. Si se desea que el efector final alcance exactamente los puntos de paso, este último argumento (el tiempo de aceleración) deberá ser nulo, siendo cada tramo lineal (velocidad constante); pero si se desea un movimiento armonizado y de mayor suavidad por los puntos de paso (como es la

situación requerida), se pondrá un tiempo de aceleración no nulo, y en este caso se pasará cerca de los puntos de paso.

Las siguientes figuras ilustran ambas situaciones:

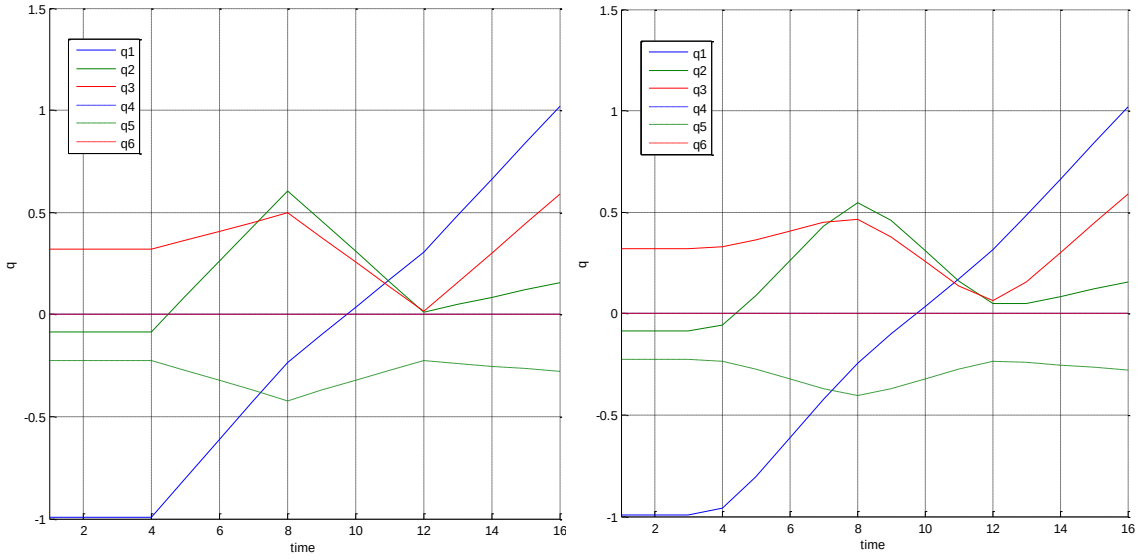


FIGURA 5.3 Interpolación

Como puede comprobarse, el primer caso muestra una situación con un tiempo de aceleración nulo, donde cada tramo es lineal, es decir, con velocidad constante. El movimiento es mucho más brusco en comparación con el segundo caso, donde el tiempo de aceleración es no nulo.

El código desarrollado para realizar una interpolación articular es el siguiente:

```
for i=1:length(puntos(:,1))
    T(:,:,i) = Calcula_T(puntos(i,1),puntos(i,2),puntos(i,3));
end

%Aplicamos la cinemática inversa
q_puntos_via=irb120_ikine(T,[0 0 0 0 0 0]);

duracion_segmento=2;
puntos_por_segmento=4;

%Interpolamos esta trayectoria
q=mstraj(q_puntos_via,[],duracion_segmento*ones(1,length(puntos(:
,1))),q_puntos_via(1,:),duracion_segmento/puntos_por_segmento,1);

Tseguida=irb_120.fkine(q);
traslacion=transl(Tseguida);
traslacion=traslacion';
```

A continuación, se ilustra un ejemplo de trayectoria obtenida mediante interpolación articular, con la respectiva representación de las trayectorias articulares en función del tiempo dibujadas mediante la instrucción "*qplot*" de Matlab. Los puntos de paso seleccionados por el usuario están representados por asteriscos negros, mientras que los puntos intermedios generados en la interpolación están representados por círculos rojos.

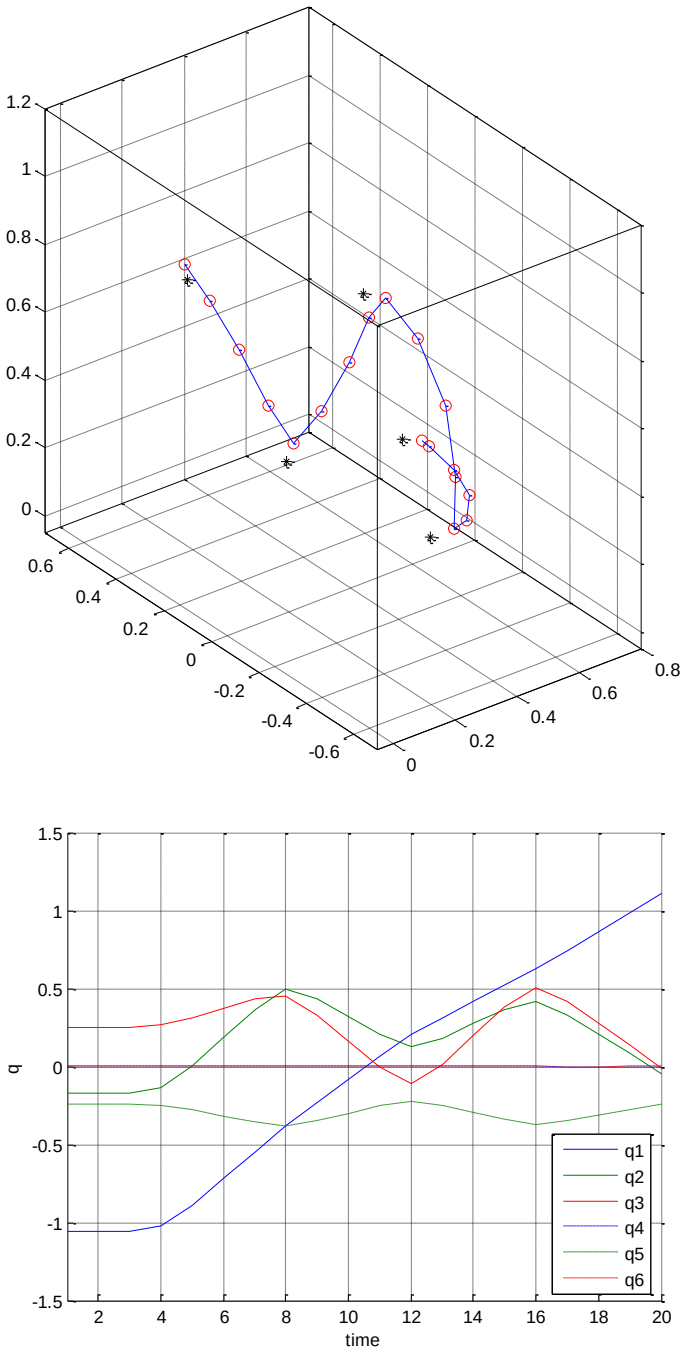


FIGURA 5.4 Interpolación articular

Como se puede concluir mediante la visualización de la figura, el movimiento de las articulaciones es suave, pero no el del efector final.

5.3 Interpolación cartesiana

El otro método empleado para la interpolación de los puntos de la trayectoria, es la interpolación cartesiana (en el espacio de la tarea). Este método consiste en la generación de la trayectoria de acuerdo a la trayectoria seguida en el espacio tridimensional, buscando una ejecución visualmente armonizada y lo más suave posible del efector final del robot.

Contrariamente a la interpolación articular, este proceso de interpolación cartesiana se lleva a cabo mediante la interpolación de los puntos de paso del efector final, para aplicar posteriormente la cinemática inversa a estos. De este modo el movimiento del efector final es suave, pero no el de las articulaciones del brazo robot.

Para asegurar que la trayectoria que une los puntos por los que tiene que pasar el efector final presente continuidad en velocidad, puede recurrirse a utilizar un polinomio de grado 3 que una cada pareja de puntos adyacentes. De esta forma, al tener cuatro parámetros disponibles se podrán imponer cuatro condiciones de contorno, dos de posición y dos de velocidad. Los valores de las velocidades de paso por cada punto deben ser por lo tanto conocidos a priori. Se consigue de esta forma un trayectoria compuesta por una serie de polinomios cúbicos, cada uno válido entre dos puntos consecutivos. Este conjunto de polinomios concatenados, escogidos de modo que exista continuidad en la posición y velocidad, se denominan splines (cúbicos por ser de tercer grado).

Para poder calcular los valores del polinomio cúbico de la expresión, para generar dicha interpolación, Matlab nos permite usar la función "*spline*". A continuación se muestra el código de programación desarrollado para la generación de este tipo de trayectoria:

```
puntos=puntos';
tam=size (puntos);
F=spline((1:tam(1,2)),puntos);

% Trajectory
step=0.2;
t=[1:step:tam(1,2)];
traslacion=ppval(F,t);

clear T;
for i=1:length(traslacion)
    T(:,:,i) = Calcula_T(traslacion(1,i), traslacion(2,i),
traslacion(3,i));
end

q=irb120_ikine(T,[0 0 0 0 0 0]);
```

La función "*spline*" permite una interpolación spline cúbica, donde el primer argumento son los puntos de paso, y el segundo la matriz con las coordenadas x, y, z de los puntos de paso, de tamaño 3xN.

```
>> F=spline((1:tam(1,2)),puntos)

F =

      form: 'pp'
    breaks: [1 2 3 4 5]
     coefs: [12x4 double]
    pieces: 4
     order: 4
       dim: 3
```

Como se puede observar, esta función devuelve la forma polinomial a trozos del interpolador spline cúbico, para su uso posterior con la función "*ppval*". Por otro lado "*ppval*" permite evaluar el valor del polinomio a trozos introducido en el primer argumento, devolviendo el valor de los puntos del polinomio introducidos en el segundo argumento.

A continuación, se ilustra un ejemplo de trayectoria obtenida mediante interpolación cartesiana, con la respectiva representación de las trayectorias articulares en función del tiempo dibujadas mediante la instrucción "*qplot*" de Matlab. Los puntos de paso seleccionados por el usuario están representados por asteriscos negros, mientras que los puntos intermedios generados en la interpolación están representados por círculos rojos.

Se puede realizar una comparativa con los resultados obtenidos a través de la interpolación articular, mostrados en la *figura 5.4*, y los obtenidos mediante

interpolación cartesiana mostrados a continuación para confirmar la teoría expuesta. En conclusión podemos destacar la diferencia en la trayectoria armonizada que debe seguir el robot en la interpolación cartesiana, en comparación con la trayectoria más pronunciada generada mediante la interpolación articular. En contraposición, la ejecución de las distintas trayectorias articulares se produce de forma más brusca e irregular en la interpolación cartesiana, mientras que en la articular se ejecutan con una mayor suavidad.

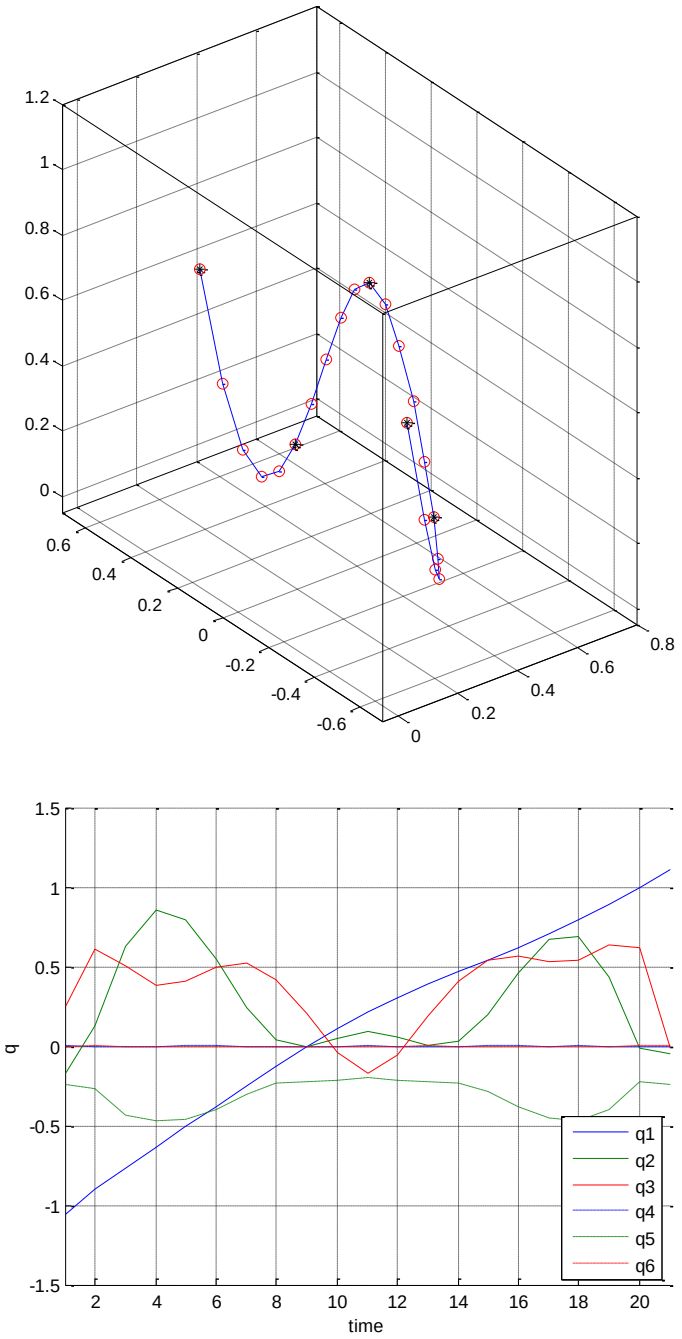


FIGURA 5.5 Interpolación cartesiana

Como se puede concluir mediante la visualización de la figura, el movimiento del efector final es suave, pero no el de las articulaciones.

5.4 Simulación de trayectorias

Uno de los puntos trascendentales en el uso de un brazo robot, es la evitación de colisiones con otros objetos, para prevenir daños en la integridad de dichos objetos o del propio robot. De esta necesidad surge la realización de este proyecto.

Por ello, además de la evitación de obstáculos móviles que se interpongan en la trayectoria del robot, es esencial que la trayectoria generada inicialmente tras la interpolación de los puntos fijados por el usuario, se haya realizado de forma correcta, siendo el robot capaz de alcanzar cada uno de sus puntos sin inconvenientes.

Para conseguirlo, se ha desarrollado una aplicación que permita su simulación del brazo robot diseñado desde Matlab en 3D para cada uno de los métodos de interpolación, previamente a la ejecución en el robot real, evitando así posibles costes innecesarios.

Se ha implementado un modelo del robot IRB120 en la plataforma Matlab, donde se ha tenido en cuenta la magnitud de la herramienta que posee conectada al efector final del brazo robótico, una ventosa de longitud 16 cm.

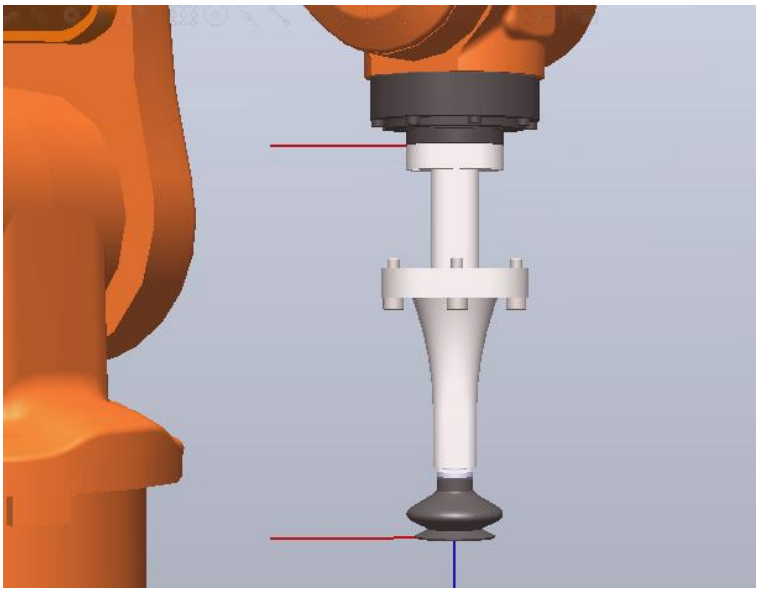


FIGURA 5.6 Herramienta ventosa

A continuación se muestra el código desarrollado para generar el modelo del robot, con la incorporación de la herramienta ventosa:

```
% Creamos el robot
l1=0.290; l2=0.270; l3=0.070; l4=0.302; l5=0.072;
%Formacion del "linkado" de cada uno de los eslabones del robot
L(1)=Link([0 l1 0 -pi/2]);
L(2)=Link([0 0 l2 0]);
L(3)=Link([0 0 l3 -pi/2]);
L(4)=Link([0 l4 0 pi/2]);
L(5)=Link([0 0 0 -pi/2]);
L(6)=Link([0 l5 0 0]);

irb_120=SerialLink(L,'name','Robot Nicolas Blanco');

irb_120.offset=[0 -pi/2 0 0 0 pi];
irb_120.tool=transl(0,0,0.16);
```

Tras generar este modelo del robot, denominado "*irb_120*", y tener calculadas las variables articulares "*q*" de la trayectoria, se realiza una simulación de la trayectoria en la plataforma Matlab, mediante la función "*irb_120.plot(q)*".

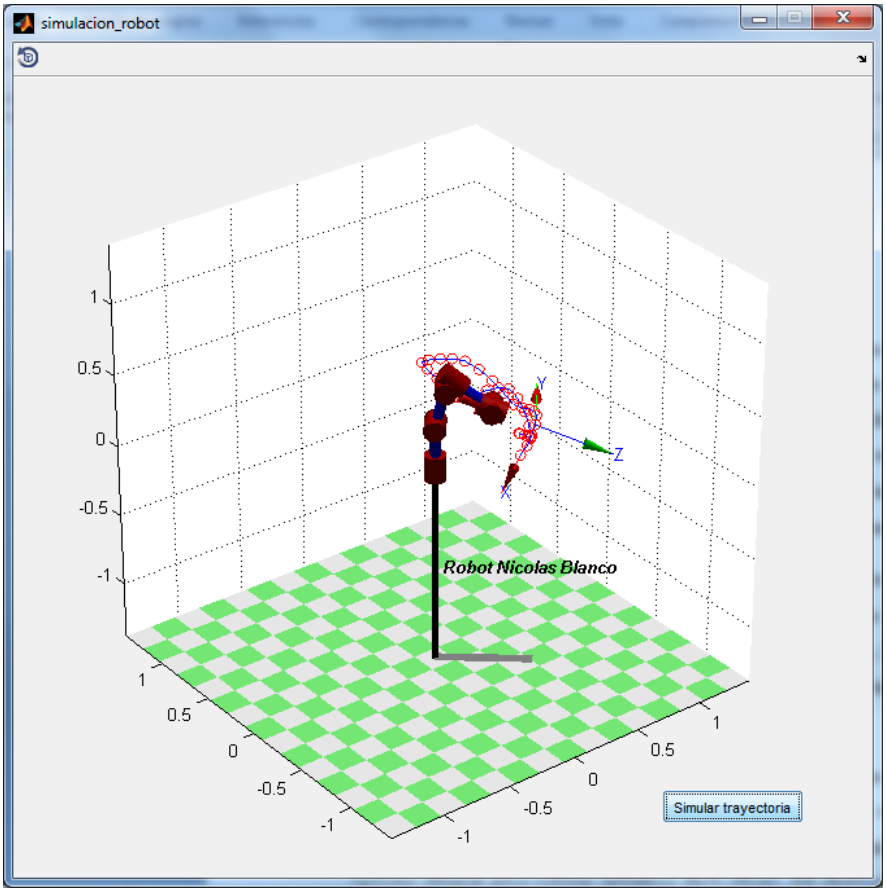


FIGURA 5.7 Simulación de trayectoria en Matlab

Una vez se haya comprobado una ejecución correcta, se puede proceder a la simulación con RobotStudio, el software de ABB. Tras establecer la conexión con la estación de RobotStudio desde Matlab, el robot se moverá a su posición de reposo, a partir de este instante, podemos comunicarnos con la estación y ejecutar la trayectoria deseada.

5.5 Evitación de obstáculos

Tras lograr una generación adecuada de las trayectorias fijadas por el usuario, su posible envío al brazo robot para ser ejecutadas (tanto real como en simulación) a través del socket de comunicación mediante datagramas, y la detección de obstáculos en el área de trabajo del robot mediante el sensor Kinect, el último paso a realizar es la regeneración de trayectorias en caso de que la presencia de un obstáculo se interponga en la trayectoria normal del robot. Este último punto es el que se tratará a continuación.

El problema ha sido abordado de la siguiente forma: en caso de detectar algún obstáculo mediante el sensor Kinect en Matlab, que interfiera en la ejecución normal de la trayectoria del robot determinada inicialmente por el usuario, la trayectoria regenerada contendrá el mismo número de puntos que la trayectoria inicial, pero con la altura suficiente para sortear el objeto.

Para ello es necesario calcular los puntos que limitan la forma del objeto. Dado que se dispone de las nubes de puntos de los obstáculos presentes en el área de trabajo del robot (almacenados en forma de matrices) mediante una sucesión de procesos iterativos se logra obtener estos valores extremos de la silueta del obstáculo.

Por otro lado, puesto que la adquisición de puntos que realiza el sensor Kinect no es uniforme, se han creado variables auxiliares sensiblemente superiores a los límites del objeto. Únicamente, en caso de que éstas sean superadas o el objeto deje de interferir en la trayectoria normal del robot fijada por el usuario, se generará una nueva trayectoria, pues en caso contrario, se producirían nuevas trayectorias en cada iteración de la adquisición. Este proceso es de gran relevancia, ya que en otro caso, se produciría un envío continuado de datagramas desde el cliente (Matlab) al servidor (robot), ocasionando una gran pérdida de tiempo y provocando problemas en la ejecución correcta de la aplicación.

Si el obstáculo se halla a la altura del efector final o por delante de él, éste se evita teniendo en cuenta únicamente el grosor del ancho, en cambio si éste se encuentra por detrás del efector final, se da cierto margen para evitar la colisión de cualquier eje con el objeto.

A continuación se muestran dos vistas de un mismo ejemplo de este proceso:

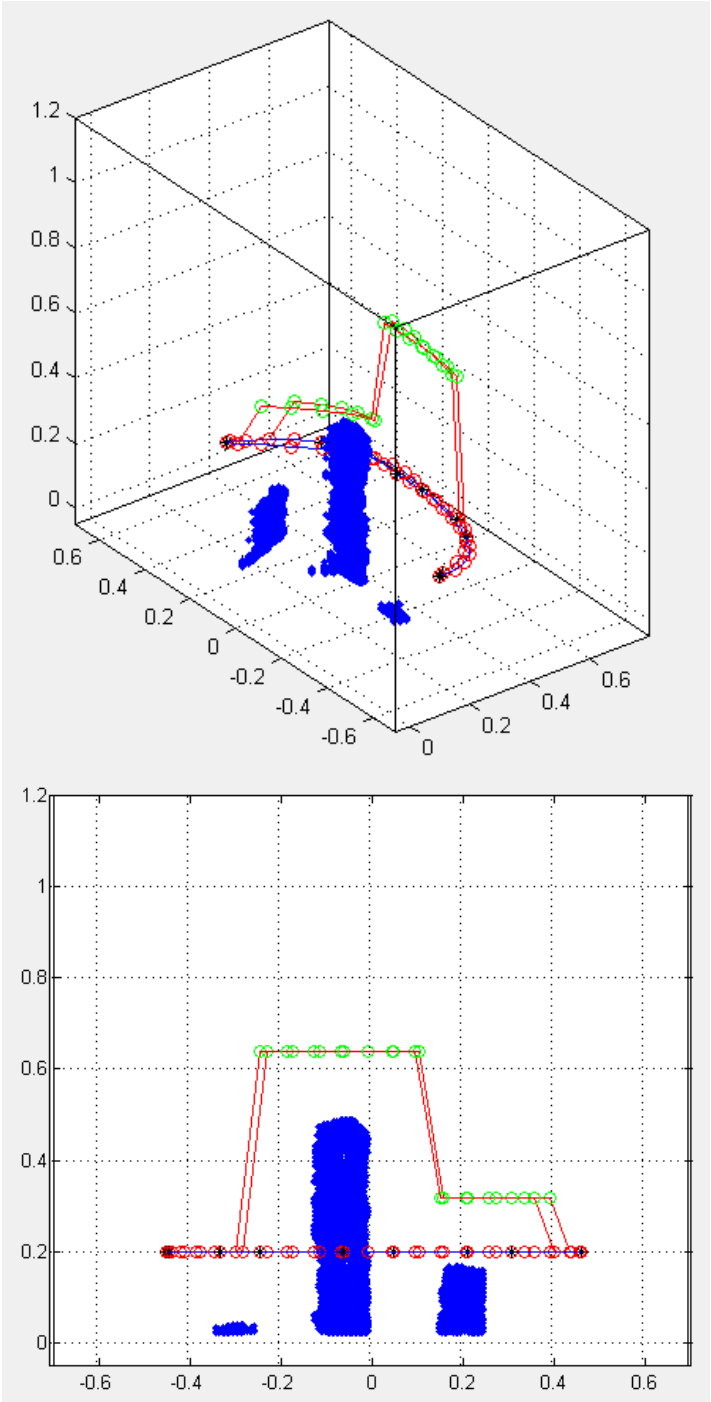
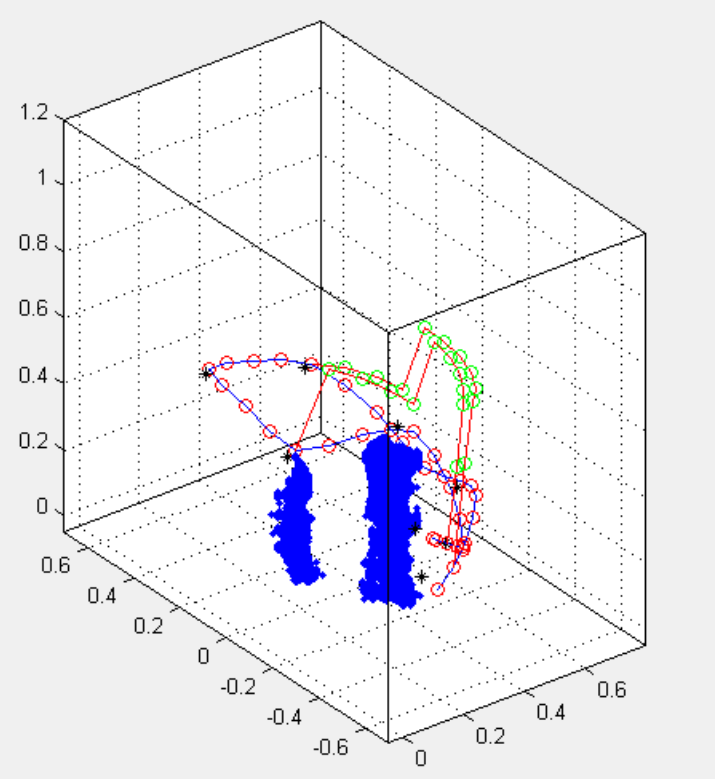


FIGURA 5.8 Regeneración trayectoria 1

Como puede contemplarse en la figura anterior, se indica la trayectoria generada por el usuario (representada en círculos rojos), y la trayectoria regenerada (representada en círculos verdes), que responde a un algoritmo en el cual únicamente los puntos que generen una colisión (o riesgo de colisión debido a su proximidad a la trayectoria) del robot con el objeto detectado, son modificados con una altura suficiente para evitar el obstáculo, permaneciendo el resto de puntos con el mismo valor.

La trayectoria mostrada, ha sido fijada a una altura constante por defecto de 0.2 m, para una mejor apreciación de la evitación de obstáculos, pero este procedimiento es efectivo para cualquier tipo de trayectoria.

A continuación se muestran dos vistas de otro ejemplo:



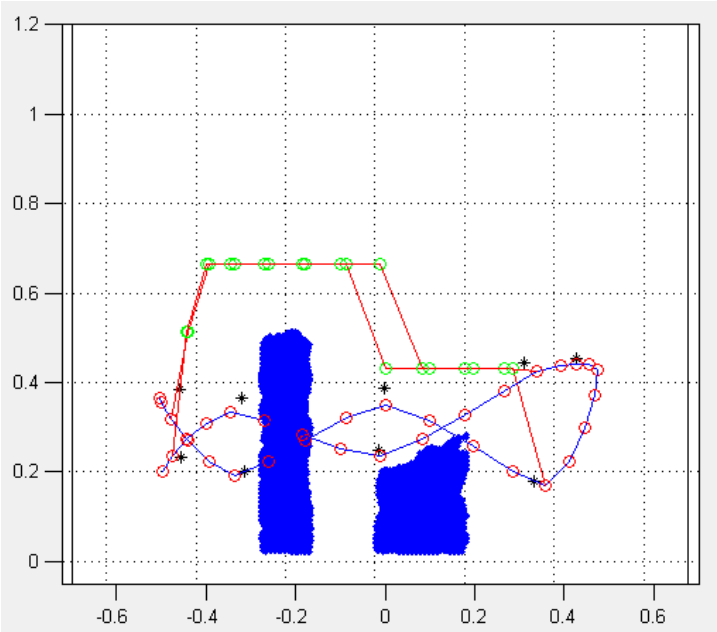


FIGURA 5.9 Regeneración trayectoria 2

Por el contrario, en caso de que no se interponga ningún objeto en la trayectoria generada por el usuario, ésta se ejecutará repetidamente de forma normal.

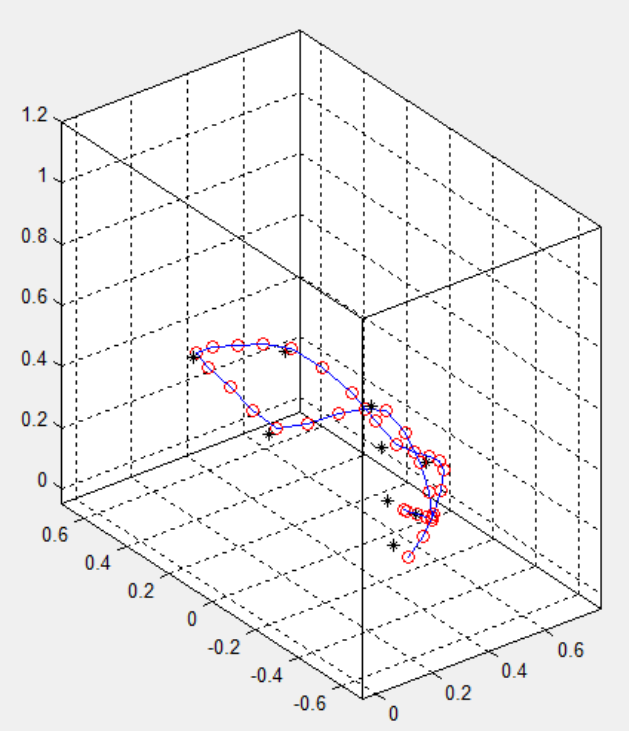


FIGURA 5.10 Trayectoria sin obstáculos

6 Desarrollo de la interfaz gráfica a través de GUI

Tras haber desarrollado el código de funcionamiento, se procede a otro de los puntos clave del proyecto, la implementación de la interfaz gráfica mediante la herramienta Guide de Matlab.

6.1 Diseño de la interfaz gráfica

Nuestra interfaz gráfica GUI posee cierta complejidad, ya que está compuesta de tres archivos de extensión .fig con sus respectivos ficheros de extensión .m para controlar y ejecutar la GUI y las funciones callbacks; además de numerosas funciones que son llamadas en el desarrollo del código para el transcurso de la aplicación. A continuación se procede al análisis de cada uno de los archivos:

- **main_func.m**

Este archivo es el encargado de inicializar la Toolbox de robótica y aquellas variables trascendentales de la aplicación de forma global, de posible utilidad para el usuario o necesarias para la comunicación entre ficheros de la GUI; crear el modelo del robot IRB120 mediante el dimensionamiento de cada uno de sus eslabones con su posterior linkado; y la llamada al archivo principal de la GUI denominado "*gui_final*".

La necesidad de inicialización de las variables de forma global para la comunicación entre ficheros de la GUI se debe a lo siguiente: las funciones de la herramienta Guide no emplean la base de workspace para el almacenamiento de las variables, sino que son locales a la función. Para ser almacenadas en el workspace principal de Matlab, se distinguen dos opciones:

- Inicialización de las variables en otro fichero diferente al generado mediante la herramienta Guide para que las almacene en el workspace.
- Utilización de la función "*assignin*" y "*evalin*" para almacenar y extraer datos respectivamente, del workspace principal. La función "*evalin*" extrae los datos

en formato *string*, por lo que si trabajamos con números será necesario pasarlo a formato *num* mediante la función "*str2num*".

Debido a la mayor optimización de código, y dado que al globalizar las variables en un archivo inicial, éstas se van almacenando de forma directa en el workspace principal con sus respectivos formatos, se ha optado por generar las variables de forma global en un fichero de inicialización.

- **gui_final.fig y gui_final.m**

Estos archivos desarrollados mediante la herramienta Guide, componen la interfaz principal de la aplicación.

En la figura siguiente se muestra la apariencia final de la interfaz desde la pantalla de edición de la herramienta Guide. En ésta se realizan las modificaciones oportunas para lograr la estética deseada para la interfaz, ciertas configuraciones, y la selección de aquellas funciones que se desean implementar en el archivo de extensión .m (callbacks)

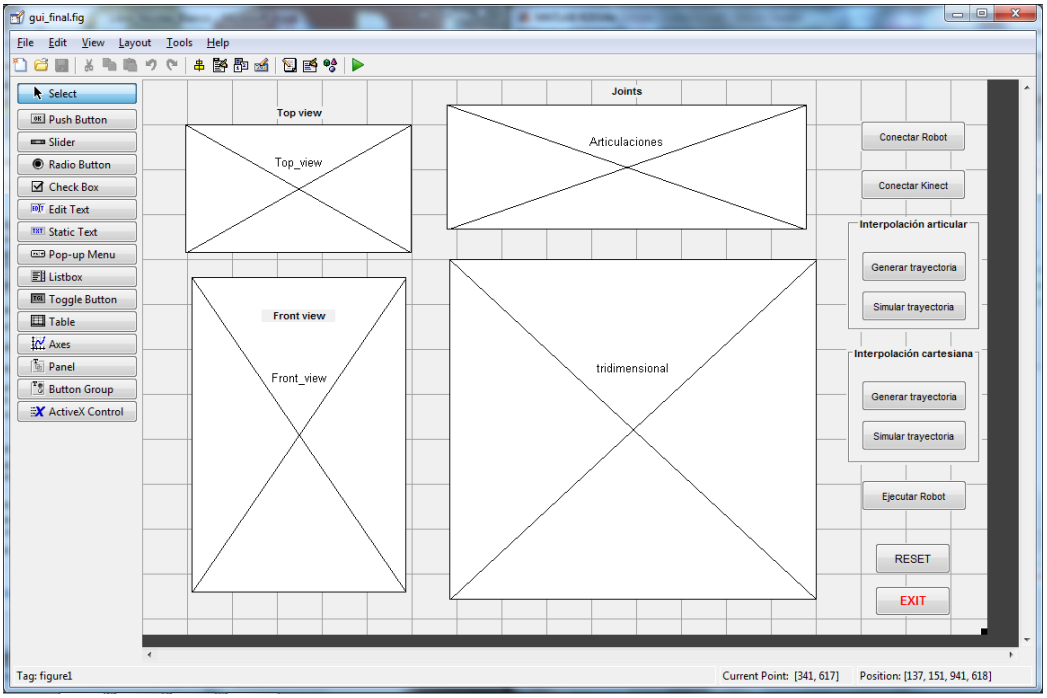


FIGURA 6.1 Interfaz gráfica principal en pantalla de edición GUIDE

A partir de la figura anterior, se explican los diferentes controles utilizados con sus respectivas funcionalidades:

- Figuras *Top_view* y *Front_view*: muestran una representación de la planta y el perfil del robot. El usuario puede definir en estas mediante el cursor los puntos de paso de la trayectoria del robot que desee. Cada figura posee dos funciones de inicialización, "*Top_view_CreateFcn*" y "*Front _view_CreateFcn*" encargadas de generar la figura y representar la vista del robot; y dos funciones "*Top_view_ButtonDownFcn*" y "*Front_view_ButtonDownFcn*" con las instrucciones necesarias tras pulsar cada una de las figuras, para almacenar los puntos seleccionados por el usuario en la figura. Estos son adquiridos mediante la función "*ginputax*" proporcionada por MathWorks, ya que la función de Matlab "ginput" produce errores en su uso en la GUI.

- Figura *Articulaciones*: muestra la ejecución de cada una de las articulaciones del robot que generan la trayectoria, en el transcurso del tiempo. Debido a que la herramienta Guide no permite el uso de la función específica de la Toolbox de Robótica "*qplot*", estas son representadas mediante la función "*plot*". Esta figura posee únicamente una función de inicialización de la figura denominada "*Articulaciones_CreateFcn*", puesto que la representación gráfica se lleva a cabo tras pulsar los botones de *Generar trayectoria* (articular o cartesiana) en sus respectivas funciones.

- Figura *tridimensional*: muestra una representación tridimensional de la trayectoria generada inicialmente por el usuario. En caso de emplear la Kinect para la evitación de obstáculos, esta figura también muestra la regeneración de la trayectoria. Posee una única función de inicialización denominada "tridimensional_CreateFcn", encargada definir las dimensiones de la figura y su estética.

- Botón *Conectar Robot*: este botón permite la conexión con el servidor para posibilitar su posterior ejecución mediante la función desarrollada "*conexión_robotstudio_Calback*". Tras pulsarlo se pregunta al usuario con qué estación desea conectar, la real o la simulación (RobotStudio), para conectar con la dirección IP adecuada a través del socket de comunicación. Una vez seleccionado se muestra una barra de progreso, desarrollada mediante la función "*waitbar*", mientras que se completa la conexión con el robot.

- Botón *Conectar Kinect*: este botón permite la conexión con el sensor Kinect una vez conectado correctamente al puerto USB. Posee una función denominada "*conectar_kinect_Callback*" donde se llama a la función "*open_kinect*" desarrollada mediante la herramienta Guide, se tratará posteriormente.

- Botón *Generar trayectoria* (Interpolación articular): tras pulsar el usuario este botón, se llama a la función "*generar_trayectoria_articular_Callback*", la cual aplica la cinemática inversa e interpola la trayectoria mediante el método articular, explicado previamente en el apartado 5.2, generando la matriz de

transformación T. Posteriormente, dibuja la ejecución de las articulaciones del robot en el transcurso del tiempo que generan la trayectoria en la figura "*Articulaciones*", y la trayectoria en un espacio tridimensional en la figura "tridimensional".

- Botón *Simular trayectoria* (Interpolación articular): tras pulsar el usuario este botón, se llama a la función "*simular_trayectoria_articular_Callback*", aplicando igualmente la cinemática inversa e interpolando la trayectoria mediante el método articular, para poder simularla en el caso de que no se hubiera generado previamente la matriz de transformación T. Posteriormente se llama a la función "*simulación_robot*" desarrollada mediante la herramienta Guide (explicada posteriormente), donde se llevará a cabo una simulación en Matlab de la trayectoria, mediante el modelo del IRB120 desarrollado.

- Botón *Generar trayectoria* (Interpolación cartesiana): tras pulsar el usuario este botón, se llama a la función "*generar_trayectoria_cartesiana_Callback*", la cual aplica la cinemática inversa e interpola la trayectoria mediante el método cartesiano, explicado previamente en el punto 5.3, generando la matriz de transformación T. Posteriormente, dibuja la ejecución de las articulaciones del robot en el transcurso del tiempo que generan la trayectoria en la figura "*Articulaciones*", y la trayectoria en un espacio tridimensional en la figura "tridimensional".

- Botón *Simular trayectoria* (Interpolación cartesiana): tras pulsar el usuario este botón, se llama a la función "*simular_trayectoria_cartesiana_Callback*", aplicando igualmente la cinemática inversa e interpolando la trayectoria mediante el método cartesiano, para poder simularla en el caso de que no se hubiera generado previamente la matriz de transformación T. Posteriormente se llama a la función "*simulación_robot*" desarrollada mediante la herramienta Guide (explicada posteriormente), donde se llevará a cabo una simulación en Matlab de la trayectoria, mediante el modelo del IRB120 desarrollado.

- Botón *Ejecutar Robot*: tras pulsar el usuario este botón, se llama a la función "*ejecutar_robot_Callback*". Esta comprueba inicialmente que la conexión con el robot se haya establecido, en tal caso, el botón posee dos estados, off para "Ejecutar Robot" y on para "Detener ejecución". También tiene en cuenta si la Kinect está conectada y adquiriendo datos, este caso estudia la presencia de obstáculos, y si es necesario regenera la trayectoria llamando a la función "*regenerar_trayectoria*" (explicada posteriormente) y la envía a la estación del robot a través del socket de comunicación desarrollado.
- Botón *RESET*: este botón se ha añadido para resetear la interfaz gráfica al estado inicial mediante la llamada a la función "*reset_Callback*". Este reseteo es cómo para el trabajo del usuario, aunque se puede trabajar directamente sobre la interfaz y los datos se irán sobrescribiendo.

- Botón *EXIT*: este botón permite cerrar la interfaz, su funcionamiento es idéntico a cerrarla mediante el aspa de la barra de menús, desarrollado en la función "*figure1_CloseRequestFcn*". Esta función cuestiona al usuario si está realmente seguro de salir de la aplicación, en caso de confirmación, desconecta el robot si está conectado, y sale de la aplicación.

- **simulacion_robot.fig y simulacion_robot.m**

Estos archivos implementados mediante la herramienta Guide, componen una interfaz auxiliar de la aplicación, que permite la simulación de la trayectoria generada, en la plataforma Matlab, empleando el modelo del IRB120 desarrollado. La figura siguiente muestra la apariencia de la interfaz vista por el usuario durante el proceso de simulación de una trayectoria:

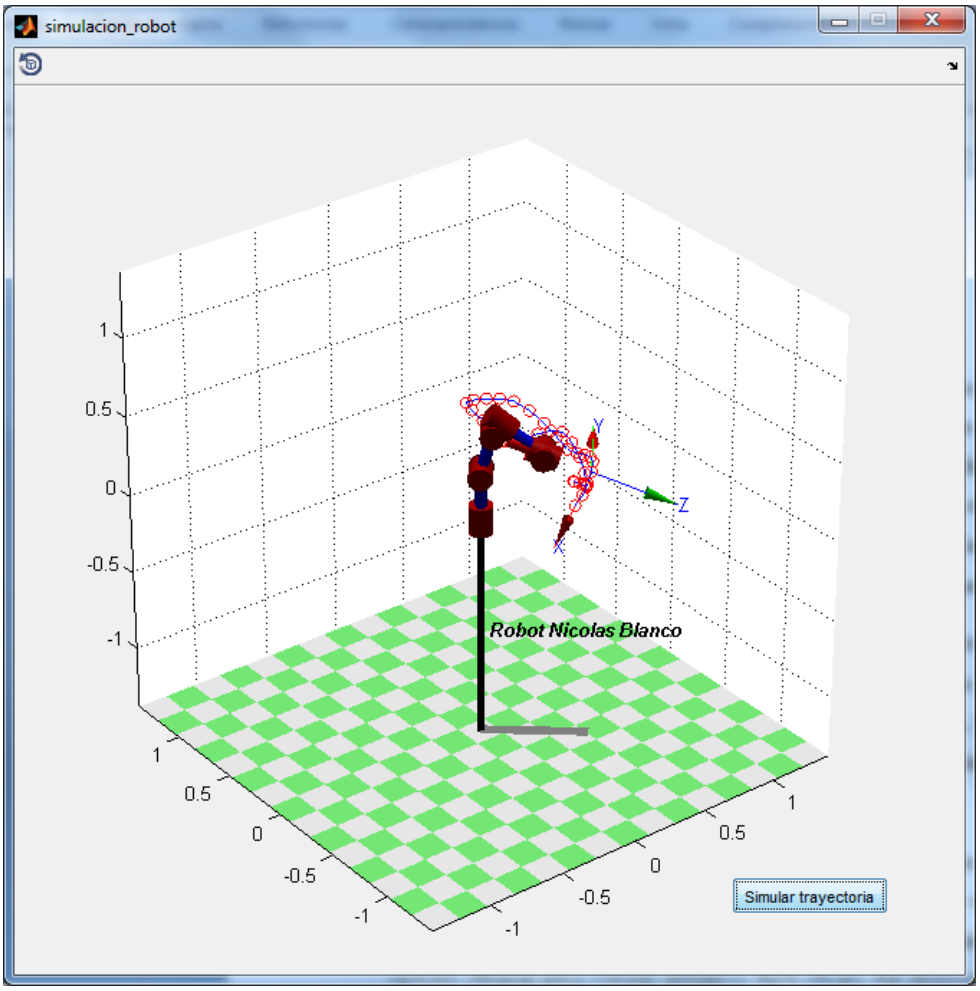


FIGURA 6.2 Interfaz gráfica simulación de trayectoria

Como se ha mostrado previamente, la llamada a esta interfaz se realiza presionando los botones de *Simular trayectoria* presentes en la interfaz principal.

Esta interfaz consta de una figura central, donde al abrir la aplicación se muestra el modelo del IRB120 desarrollado y la trayectoria creada por el usuario.

Por otro lado también presenta un botón denominado *Simular trayectoria* que permite la simulación del robot.

Adicionalmente se ha añadido mediante la opción Toolbar editor, la posibilidad de rotación tridimensional de la figura, para mejor apreciación de la simulación y comodidad del usuario a través de un botón de rotate en la barra de herramientas.

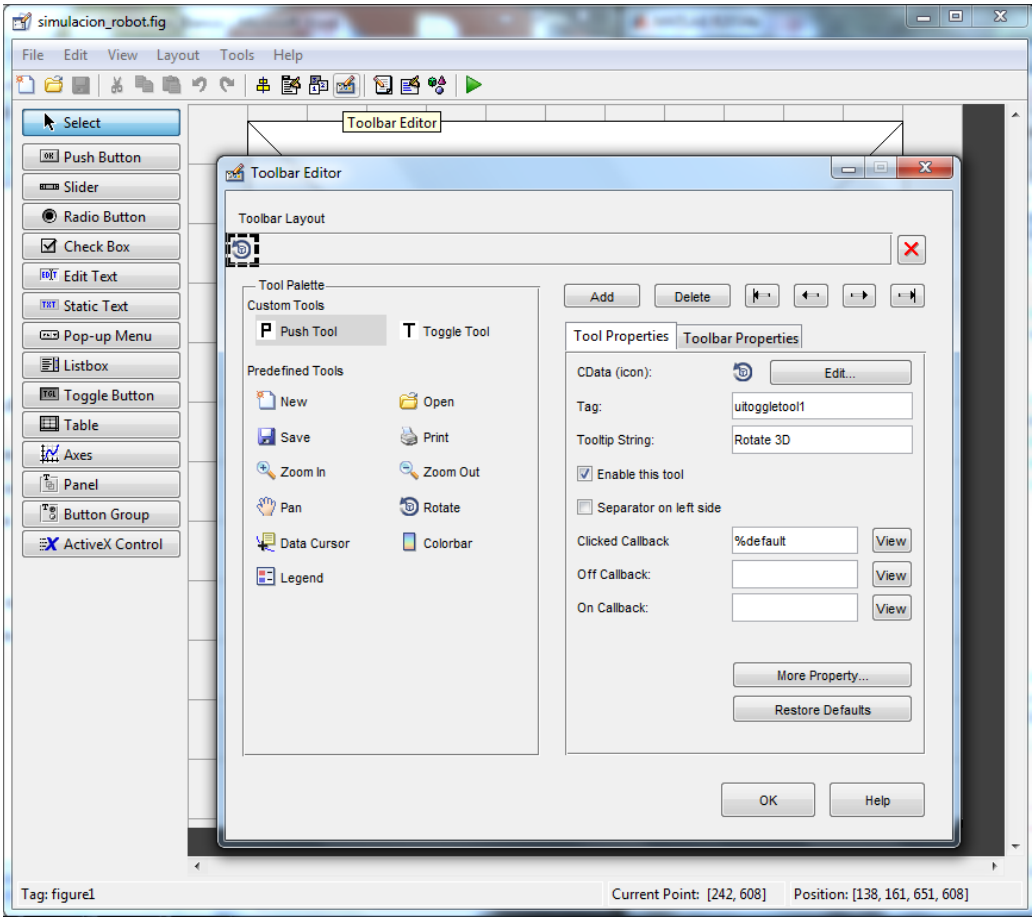


FIGURA 6.3 Rotate 3D, Toolbar Editor

- **open_kinect.fig y open_kinect.m**

Estos archivos implementados mediante la herramienta Guide, componen una interfaz auxiliar de la aplicación, que permite la conexión del sensor Kinect, su correcta

calibración con la zona de trabajo del brazo robot, y la adquisición y registro de datos del entorno.

La figura siguiente muestra la apariencia de la interfaz vista por el usuario durante el proceso de simulación de una trayectoria:

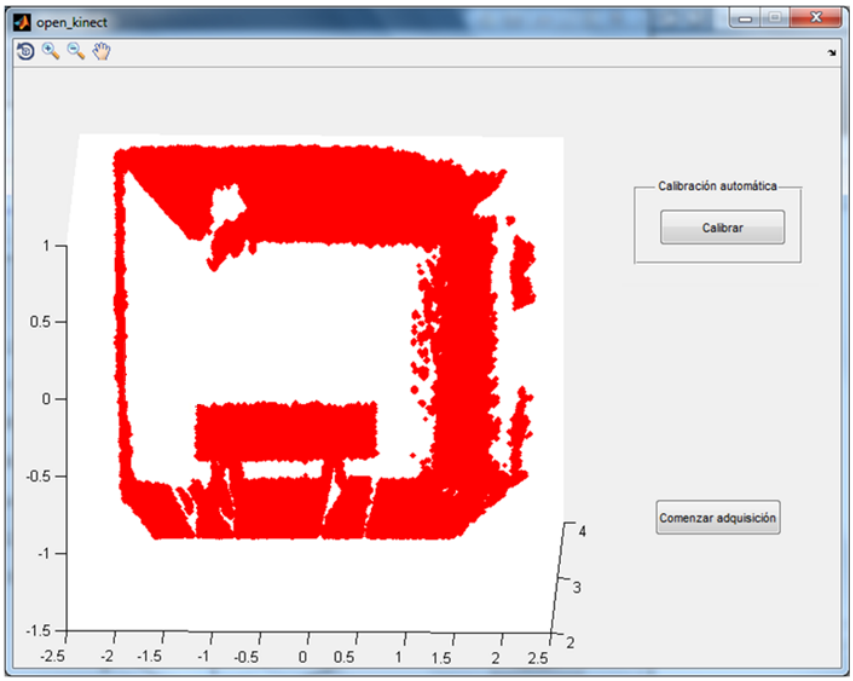


FIGURA 6.4 Interfaz gráfica sensor Kinect

Tras pulsar el botón *Conectar Kinect* de la interfaz principal, se hace una llamada a la función "*open_kinect*". Si el sensor Kinect está conectado adecuadamente, se abre esta segunda interfaz, mostrando una imagen tridimensional del campo de visión de la Kinect.

Esta imagen es ofrecida tras ejecutarse la función de inicialización de la figura "*axes1_CreateFcn*", donde se activa el sensor de profundidad de la Kinect, la adquisición de datos, y el registro de estos. Posteriormente calcula la nube de puntos en el espacio a partir de la imagen de profundidad, mediante la función creada "*depthtoxyz*", mostrando finalmente los puntos en la figura de la interfaz.

El sensor Kinect debe estar calibrado adecuadamente con la zona de trabajo del robot, por ello se ofrece al usuario un métodos de calibración automático de gran precisión y comodidad.

Este procedimiento de calibración se lleva a cabo pulsando el botón *Calibrar*, ejecutando de este modo la función "*calibración_automatica_Callback*". Para ello, debe haberse ubicado la diana correctamente en el Lay-out del robot (centrada y a 20cm de la base del robot). A continuación se muestra en la figura:

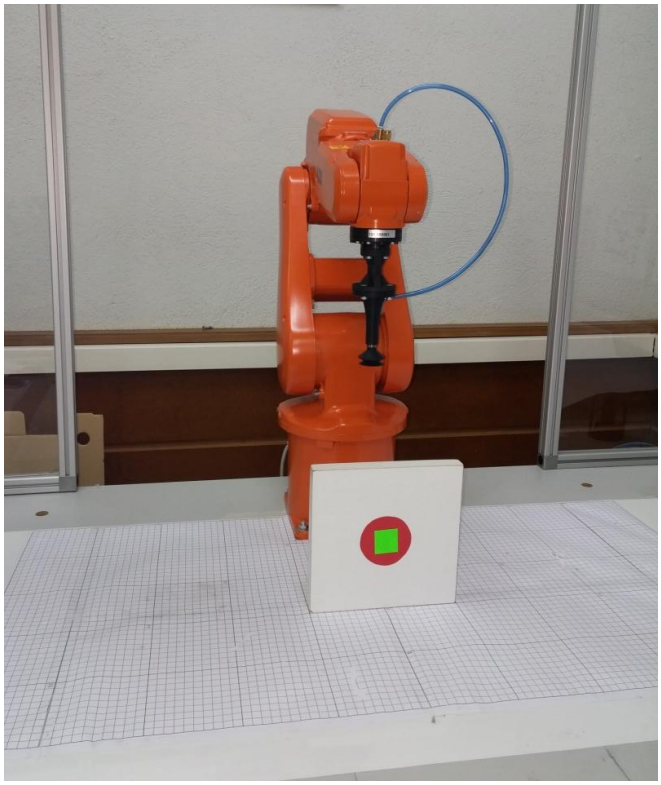


FIGURA 6.5 Ubicación diana en zona de trabajo del robot

La función desarrollada, adquiere inicialmente una imagen mediante la cámara RGB de la Kinect. Posteriormente se detectan los círculos con un radio similar al de la diana que aparezcan en la imagen mediante la función "*imfindcircles*", rechazando aquellos que no dispongan de color verde en el centro y rojo en el exterior. Este proceso se ha llevado a cabo mediante una comparación de los píxeles del círculo con un rango de colores rgb seleccionado tras numerosas pruebas. En caso de una detección, se pregunta al usuario si la localización de la diana realizada ha sido la correcta para evitar errores. En caso de ser incorrecta, continua el mismo proceso hasta detectar el blanco.

Cuando la diana ha sido detectada, se captura una imagen mediante el sensor de profundidad, y la función desarrollada ubica el pixel del centro del círculo de la imagen rgb, en la imagen de profundidad. Dado que la tabla que contiene la diana posee unas

dimensiones de 20x20, y se conoce su localización en el espacio de trabajo con respecto al robot, es sencillo ubicar el sensor Kinect con respecto al brazo robótico.

Finalmente, tras realizar el calibrado del sensor Kinect de forma correcta se puede proceder a la adquisición de datos. Para ello debe pulsarse el botón de tipo togglebutton *Comenzar adquisición*, ejecutándose de este modo la función desarrollada "*comenzar_adquisicion_Callback*". Esta lleva a cabo un proceso iterativo de detección y diferenciación de objetos mediante cálculos matriciales en cada una de las coordenadas x, y, z, con las nubes de puntos adquiridas, detectando aquellos obstáculos presentes sobre el área de trabajo del robot y rechazando el resto de objetos del entorno y el ruido que pueda generar la adquisición de puntos de la Kinect. En caso de detectar algún obstáculo que se interponga en la trayectoria, se llama a la función desarrollada "*regenerar_trayectoria*" para modificar los puntos de paso del brazo robot y enviar la nueva trayectoria al servidor a través del socket de comunicación TCP/IP. La nueva trayectoria es mostrada en la interfaz para visualización del usuario.

Si el botón es pulsado de nuevo, pasando a un estado de off, la adquisición de datos del sensor finaliza.

6.2 Limitaciones de GUI

En este apartado se tratan algunas limitaciones de la herramienta Guide encontradas en la implementación de la interfaz, con posibles soluciones que puedan ser útiles para el desarrollo de nuevas interfaces.

- La función "*ginput*" de Matlab permite identificar puntos sobre unos ejes, devolviendo sus coordenadas en vectores. Esta función podría haberse empleado para definir el usuario los puntos de paso de la trayectoria del robot mediante el cursor, en las figuras *Top view* y *Front view*. Aunque al emplear esta función, erróneamente los gráficos se van barriendo a medida que se desplaza el cursor a través de la interfaz creada. La siguiente figura refleja este comportamiento:

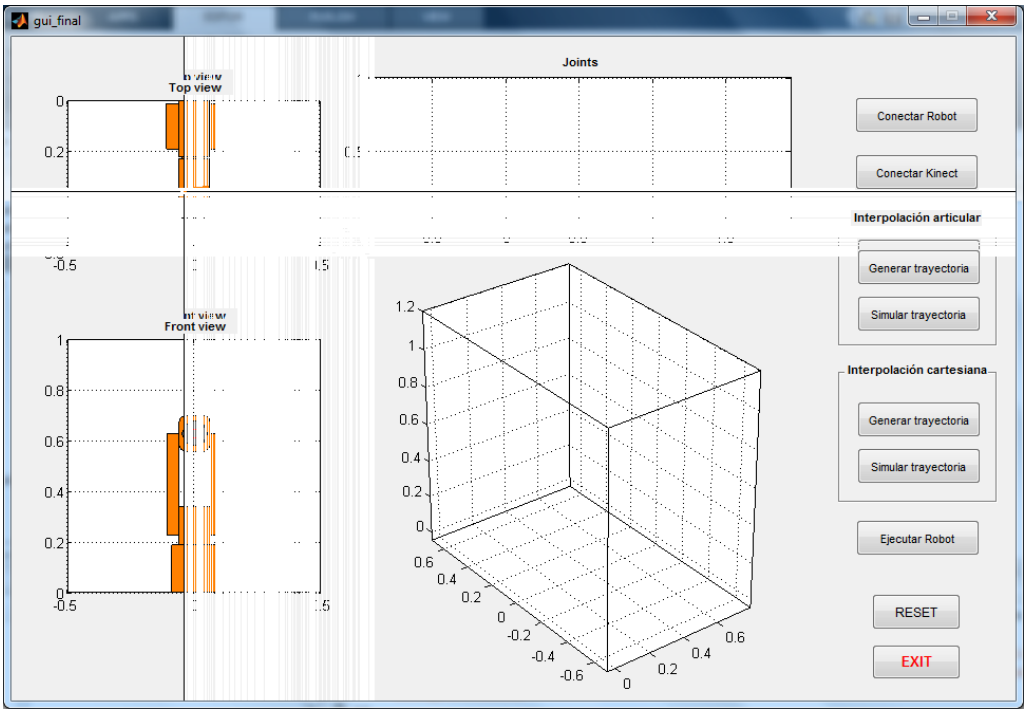


FIGURA 6.6 Error función *ginput* en la herramienta Guide

Por ello, ha surgido la necesidad de emplear una función facilitada por MathWorks denominada "*ginputax*", que posee una funcionalidad similar pero no presenta este error.

- La función "*qplot*" de la Toolbox de Robótica, permite representar las trayectorias articulares en función del tiempo para robots con 6 ejes, como es el caso del IRB120, indicando que curva representa cada articulación. La herramienta Guide impide la utilización de esta función, por lo que se ha optado por emplear simplemente la función "plot".

- En la herramienta Guide, cada componente en la interfaz posee un identificador denominado *tag*, para poder referirse a este, desde la programación del código mediante la instrucción "handles.*tag"*. En cambio, cuando se emplean diversos ejes y se plotea en uno de ellos, la estructura GUIHANDLES se modifica, y el *tag* de los ejes ya no figura. A continuación se ejemplifica este comportamiento. Se ha creado una GUI compuesta únicamente de una figura. La instrucción "*guihandles*" muestra la estructura handle de dicho componente. En la función

de creación de la figura, se muestra la estructura inicial y la posterior a un ploteo, el código implementado es el siguiente:

```
function tag_prueba_axes_CreateFcn(hObject, eventdata, handles)
      guihandles
      plot([1,2,3,2,1]);
      guihandles
```

Obteniendo los siguientes resultados en la ventana de comandos:

```
>> pruebaguide

ans =

            figure1: 231.0044
    tag_prueba_axes: 232.0049


ans =

    figure1: 231.0044
```

Para evitar este error se lleva a cabo un proceso de refresco del *tag* de la figura tras realizar el ploteado, para ello se ha desarrollado el siguiente código:

```
function tag_prueba_axes_CreateFcn(hObject, eventdata, handles)
      guihandles
      ejes=gca;
      refrescar_tag = get(ejes, 'Tag');
      plot([1,2,3,2,1]);
      set(ejes, 'Tag', refrescar_tag);
      guihandles
```

Obteniendo los siguientes resultados en la ventana de comandos:

```
>> pruebaguide

ans =

            figure1: 232.0052
    tag_prueba_axes: 233.0046


ans =

            figure1: 232.0052
    tag_prueba_axes: 233.0046
```

7 Conclusiones y trabajo futuros

En el presente proyecto se ha tratado el desarrollo de una aplicación que permita una comunicación con el brazo robótico IRB120 de ABB desde el entorno Matlab, posibilitando la generación de trayectorias definidas por el usuario a través de unos "puntos de paso" intermedios, así como la detección de nuevos obstáculos presentes en la trayectoria del robot y la planificación de nuevas trayectorias recalculadas para evitarlos.

Con todo ello, la detección del área de trabajo del robot se efectuará mediante una adquisición de datos del entorno empleando el sensor Kinect. Para ello se interactuará con el robot y el sensor Kinect en entorno Matlab, implementando el desarrollo de una interfaz gráfica a través de la herramienta GUIDE que permita enviar los comandos al robot a través de un socket de red.

El resultado ha sido ampliamente satisfactorio, debido a que se ha conseguido:

- Una comunicación fluida y eficaz desde el entorno Matlab que supera la limitación del tiempo, debido a las necesidades de ejecución de la aplicación en tiempo real. Esto posibilita la ejecución de una trayectoria de forma continuada, compuesta por diversos puntos de paso indicados por el usuario. El proceso seguido para ello, no ha sido otro que la creación de un nuevo datagrama que posibilite el envío de la trayectoria completa a través del socket de comunicación TCP/IP.
- La evitación de obstáculos interpuestos en la trayectoria del robot, permitiendo una posible actuación de regeneración de trayectoria en cualquier instante sobre la trayectoria indicada inicialmente. Esto se ha abordado mediante una escucha continuada desde el servidor a través del socket, esperando la recepción de nuevos datagramas y regenerando la trayectoria teniendo en cuenta la posición actual del robot.
- Una distinción de obstáculos en contacto con la superficie de trabajo del robot mediante el sensor Kinect, sin considerar aquellos que no se ubiquen en el entorno de trabajo. Esto se ha logrado a pesar de la imposibilidad de uso de

"Point Cloud Library" desde la plataforma Matlab debido a su escaso desarrollo hasta el momento[6].

Expuestas estas conclusiones y logros obtenidos en la elaboración de este trabajo, como todo proyecto caben mejoras, más aún teniendo en cuenta que no existen precedentes en el ámbito de la Universidad de Alcalá de Henares respecto a la utilización del sensor Kinect desde la plataforma Matlab. Con lo cual, considero que un área funcional que abordar en trabajos futuros, es la distinción de objetos en la adquisición de datos, así como la incorporación de nuevos sensores Kinect en el entorno del robot, que permitan un mayor control y detección del entorno de trabajo.

Otro posible trabajo futuro, podría ser la incorporación de sensores infrarrojos de profundidad en el propio robot, trabajando con ellos de forma dinámica para una detección de objetos extraños en el entorno, evitando así daños y colisiones, y posibilitando el trabajo sin resguardos de seguridad, incluso al lado de operarios humanos.

En cuanto a aplicaciones prácticas del presente proyecto, cabe destacar la incorporación del mismo al área industrial, lo cual gracias a la capacidad desarrollada de detección de obstáculos podemos relacionarlo con el campo de prevención de riesgos laborales dentro de áreas de trabajo y fabricación.

El robot IRB120 de ABB, utilizado en este proyecto, precisa de un resguardo de seguridad con el fin de evitar que objetos o incluso personas se interpongan en su área de trabajo, lo que podría ocasionar daños tanto en la integridad del propio robot, como en la producción y en aquellos operarios que por diversos motivos se encuentren en el entorno de trabajo del robot. Por ello, tras investigar en el mundo de los robots, llama la atención la innovadora modalidad de robots denominados "colaborativos y seguros", capaces de operar sin resguardos de seguridad, incluso al lado de operadores humanos. Sería interesante aplicar lo logrado en esto proyecto, con las mejoras necesarias que ello conllevara, a este ámbito de la robótica.

Otra de las aplicaciones a destacar, es la posibilidad de trabajar con el brazo robot, sin necesidad de acceder a la planta de forma física, debido a la conexión fluida y eficaz lograda a través del socket de comunicación basado en el protocolo TCP/IP.

[6] Julio 2015

Manual de usuario

En este capítulo se detalla un manual para el usuario, con el fin de obtener el máximo rendimiento de la aplicación. Se expondrán de forma detallada cada una de las interfaces creadas, las posibilidades en cuanto a su uso y funcionalidades, el modo de empleo, el flujo de programa, etc. También se indicará una guía de botones para que el usuario comprenda eficazmente el funcionamiento del sistema.

M.1 Guía de la interfaz gráfica a través de GUI

La interfaz GUI está desarrollada, de modo que al ser ejecutada la aplicación principal, o simplemente escribiendo en la ventana de comandos de Matlab el nombre del archivo *main_func*, nos aparezca la ventana de la interfaz lista para ser usada. Esta función principal ejecuta una serie de líneas de código necesarias para el desarrollo de la interfaz, llamando finalmente a la interfaz principal.

```
>> main_func
```

La figura mostrada a continuación compone la ventana principal desde la que se trabajará, para que el robot realice las aplicaciones desarrolladas, desplegando otras interfaces secundarias que incrementen las funcionalidades de la aplicación y enviando las instrucciones necesarias para que el robot ejecute los correspondientes movimientos.

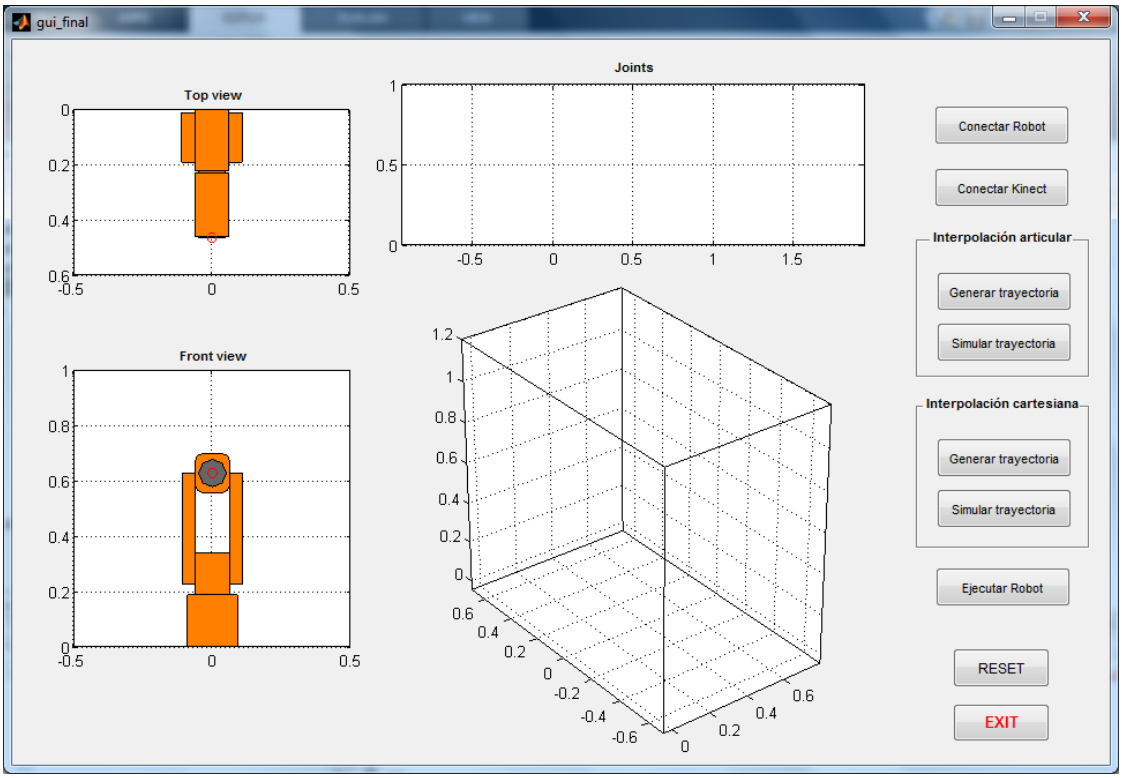


FIGURA M.1 Ventana de la interfaz gráfica principal (gui_final)

A continuación se detallan cada uno de los componentes de las diferentes interfaces, su significado y su funcionalidad:

M.1.1 Interfaz principal, "gui_final"

A continuación se detallan los distintos componentes que presenta la interfaz principal:

- Figura *Top view*

A la izquierda del panel, la figura superior muestra una vista de la planta del brazo robótico IRB120. En ella el usuario podrá seleccionar con el cursor los puntos de paso de la trayectoria tras pinchar en la figura.

- Figura *Front view*

A la izquierda del panel, la figura inferior muestra una vista frontal del brazo robótico IRB120. En ella el usuario podrá seleccionar con el cursor la altura de los puntos de paso de la trayectoria tras pinchar en la figura y haber seleccionado previamente los puntos de paso en la figura *Top view*. (Por defecto la altura de la trayectoria es de 20 cm).

- Figura *Joints*

La figura derecha superior, denominada *Joints*, indica las trayectorias articulares ejecutadas por el robot para generar la trayectoria en función del tiempo de cada uno de los 6 ejes que presenta el IRB120.

- Figura tridimensional

La figura inferior derecha, muestra al usuario una visualización tridimensional de la trayectoria generada por éste.

- Botón *Conectar Robot*

Este botón permite el establecimiento de la conexión con el brazo robot, tanto con la estación simulada de RobotStudio, como con la estación real.

- Botón *Conectar Kinect*

Este botón permite la conexión con la Kinect. Abre una segunda interfaz específica, que permite la calibración del sensor y la adquisición de datos de la zona de trabajo del robot.

- Botón *Generar trayectoria* (*Interpolación articular*)

Este botón permite la generación de la trayectoria mediante interpolación articular, tras haber indicado el usuario los puntos de paso de la trayectoria en las figuras *Top view* y *Front view*. Tras pulsarlo, se genera la visualización de las trayectorias articulares del robot en función del tiempo en la figura *Joints*, y la visualización tridimensional de la trayectoria creada a ejecutar por el robot.

- Botón *Simular trayectoria* (*Interpolación articular*)

Este botón permite la simulación de la trayectoria mediante interpolación articular. Abre una interfaz secundaria, empleando el modelo del IRB120 creado en Matlab para reproducirlo.

- Botón *Generar trayectoria* (*Interpolación cartesiana*)

Este botón permite la generación de la trayectoria mediante interpolación cartesiana, tras haber indicado el usuario los puntos de paso de la trayectoria en las figuras *Top view* y *Front view*. Tras pulsarlo, se genera la visualización de las trayectorias articulares del robot en función del tiempo en la figura *Joints*, y la visualización tridimensional de la trayectoria creada a ejecutar por el robot.

- Botón *Simular trayectoria* (*Interpolación cartesiana*)

Este botón permite la simulación de la trayectoria mediante interpolación cartesiana. Abre una interfaz secundaria, empleando el modelo del IRB120 creado en Matlab para reproducirlo.

- Botón *Ejecutar Robot*

Este botón, permite la ejecución de la trayectoria generada por el usuario, tras haber establecido conexión con una estación (real o simulada). La ejecución de la trayectoria se realiza de forma continuada. Si el sensor Kinect está conectado correctamente y detecta algún objeto que interfiera en la trayectoria, automáticamente envía una nueva trayectoria para evitar el obstáculo. Este botón tiene otro estado, *Detener Ejecución*, que permite parar el movimiento del robot una vez esté ejecutándose.

- Botón *RESET*

Este botón permite resetear los gráficos de la interfaz. Posibilita al usuario una mayor comodidad de trabajo, aunque se puede trabajar directamente sobre la interfaz y los datos se irán sobrescribiendo.

- Botón *EXIT*

Este botón permite salir y cerrar la interfaz.

M.1.2 Interfaz de simulación de trayectoria, "simulacion_robot"

Esta interfaz se abre tras pulsar uno de los botones de *Simular trayectoria* en la interfaz principal, ya sea mediante interpolación articular o cartesiana. Consta de los siguientes componentes:

- Figura tridimensional

Muestra una visión tridimensional del modelo IRB120 creado, con la trayectoria a simular indicada por el usuario.

- Botón *Simular trayectoria*

Este botón permite la reproducción de la trayectoria, empleando el modelo del IRB120 creado en Matlab para simularlo.

La ventana que aparece es la mostrada a continuación:

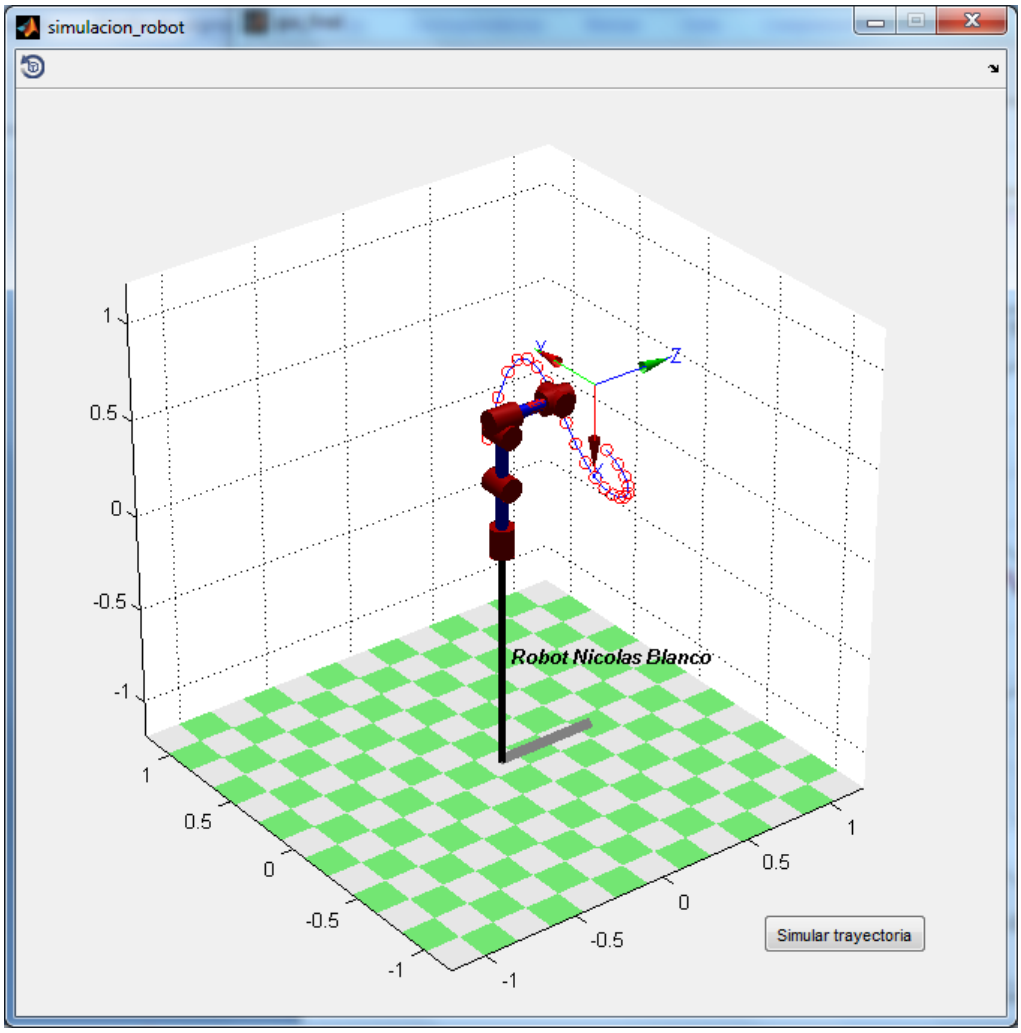


FIGURA M.2 Ventana de la interfaz gráfica de simulación (simulación_robot)

M.1.3 Interfaz de conexión sensor Kinect, "open_kinect"

Esta interfaz se abre tras pulsar el botón *Conectar Kinect* en la interfaz principal. Si el sensor está correctamente conectado. Consta de los siguientes componentes:

- Botón *Calibrar* (*Calibración automática*)

Este botón permite una calibración automática del sensor Kinect. Para ello debe ubicarse el tablero "diana" sobre la superficie de trabajo del brazo robot a 20 cm de su base, y la Kinect debe situarse frontal a este sobre una superficie, a una distancia entorno a 2 metros.

- Botón *Comenzar adquisición*

Este botón permite al usuario comenzar la adquisición de datos de la superficie de trabajo del robot, y regenerar la trayectoria en caso de que se interponga algún objeto sobre la trayectoria seleccionada por el usuario.

La ventana que aparece es la mostrada a continuación.

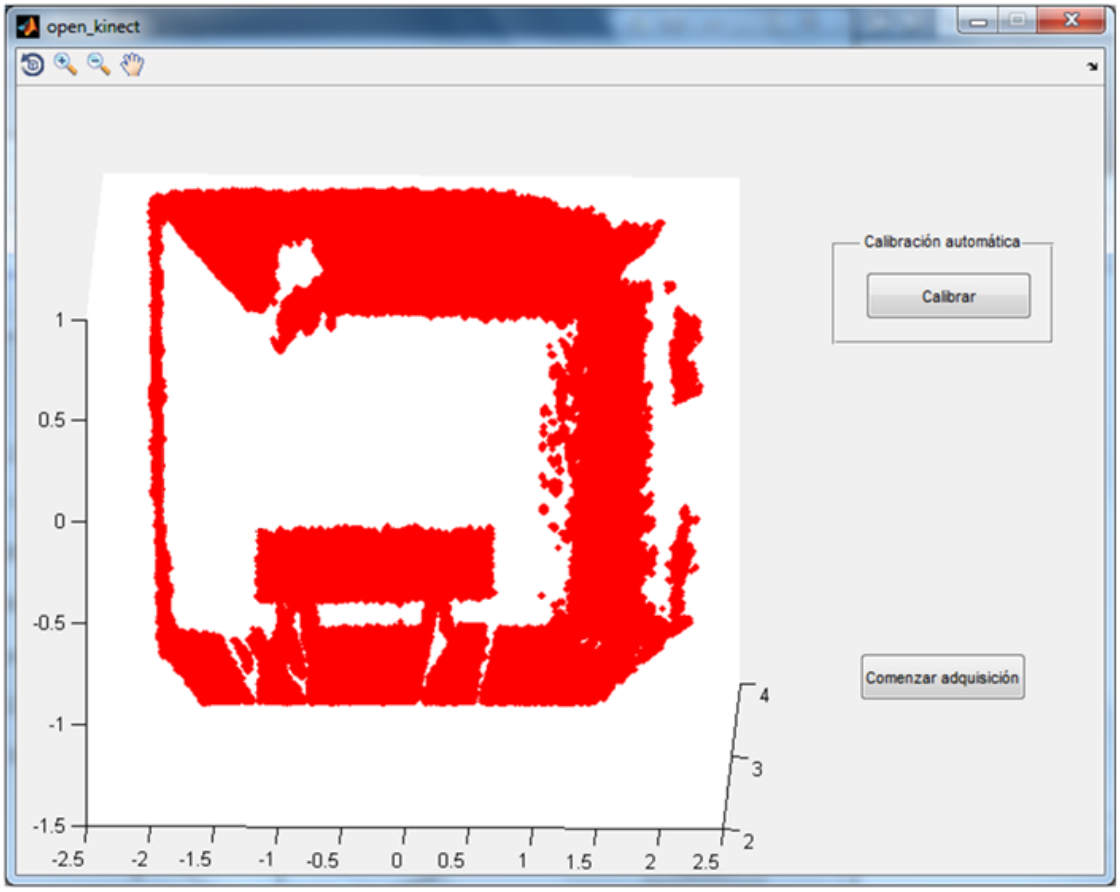


FIGURA M.3 Ventana de la interfaz gráfica de conexión Kinect (open_kinect)

M.2 Flujo de programa

El flujo de trabajo se inicia abriendo y ejecutando la aplicación "*main_func*", o directamente desde la ventana de comandos de Matlab, escribiendo "*main_func*". A continuación se abrirá la interfaz principal, debiendo seguir los siguientes pasos para su utilización:

1. Pinchamos sobre la figura *Top view* y comenzamos la **selección de puntos** mediante el cursor para determinar la trayectoria. Cuando finalicemos la selección, debemos pulsar el botón *Enter* del teclado. Se permite una selección máxima de 10 puntos.

2. Es opcional pinchar sobre la figura *Front view* para determinar la altura de los puntos seleccionados previamente. De igual forma que en el paso anterior, debemos pinchar con el cursor sobre la figura, y seleccionar por orden la altura de cada uno de los puntos. Por defecto, la altura de los puntos será de 20 cm con respecto a la superficie de trabajo del robot. A continuación se muestra una figura que detalla estos dos pasos:

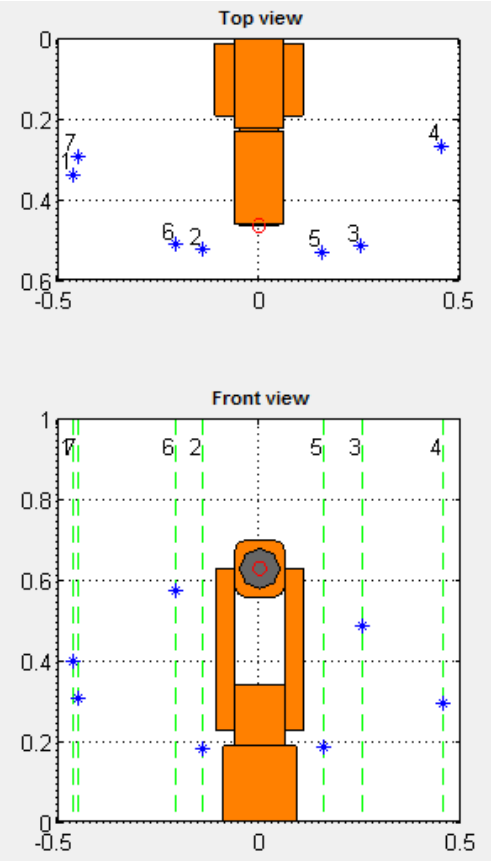


FIGURA M.4 Selección de puntos de la trayectoria

3. Una vez seleccionados los puntos, podemos **generar la trayectoria** pulsando el botón *Generar trayectoria* mediante los métodos de interpolación articular o cartesiana (según convenga al usuario). Así se mostrarán la representación de las trayectorias articulares en función del tiempo y la trayectoria generada de forma tridimensional.

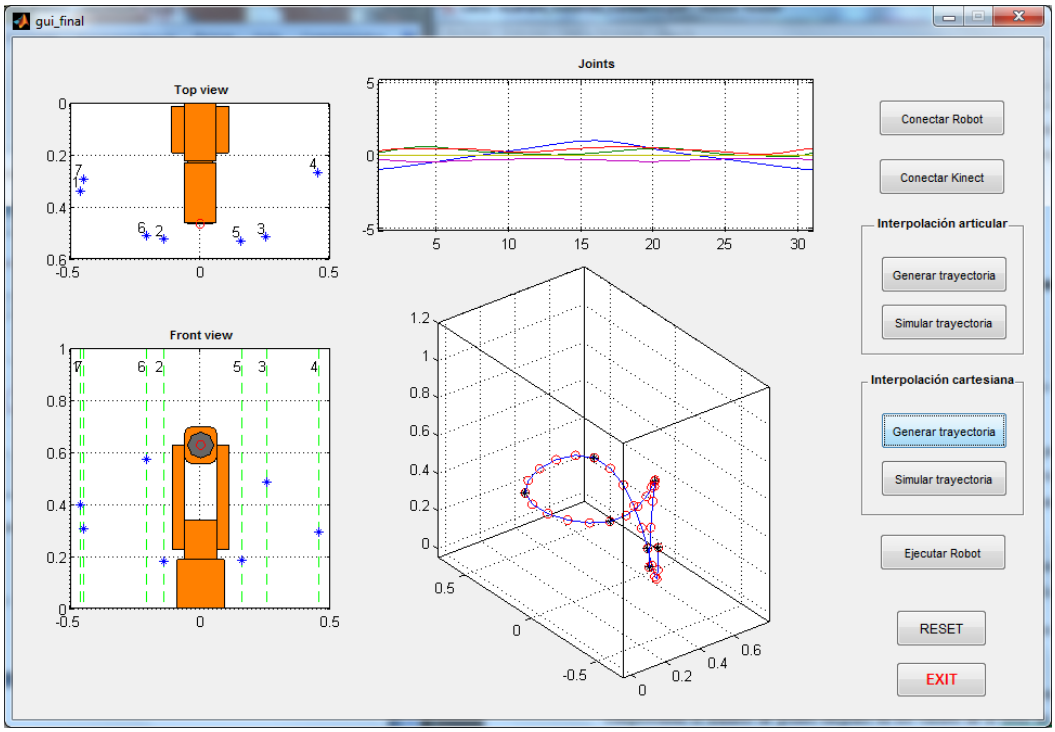


FIGURA M.5 Generación de trayectoria

4. Previamente a la simulación de la estación real, es recomendable **simular** la ejecución de **la trayectoria en Matlab** mediante un modelo creado del robot IRB120.

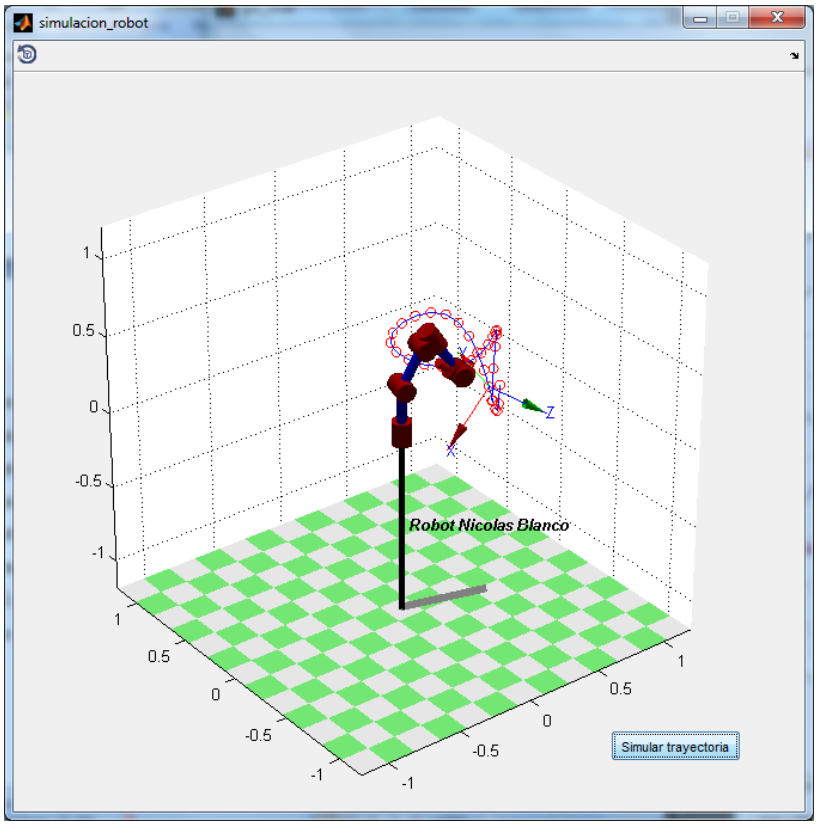


FIGURA M.6 Simulación de trayectoria en Matlab

Esto permite comprobar si el brazo robótico alcanza adecuadamente todos los puntos, previniendo así daños en su integridad. Para ello debe seleccionarse el botón *Simular trayectoria* (articular o cartesiana según convenga al usuario), de modo que se abrirá una nueva ventana con un botón de simulación. Al pulsarlo, el robot reproducirá la trayectoria. El usuario puede ayudarse del botón de rotación en la barra de herramientas, para una mejor visualización de la trayectoria.

5. En caso de detectar algún error, podemos seleccionar el botón *RESET* de la interfaz principal, para realizar de nuevo este procedimiento de generación de trayectoria, o trabajar directamente sobre la interfaz, de forma que los datos se irán sobrescribiendo.

6. Una vez seleccionada la trayectoria deseada de forma correcta, podemos proceder a la **conexión con la estación** del robot. Para ello debemos pinchar el botón *Conectar Robot* de la interfaz principal, apareciendo el siguiente cuadro de diálogo:

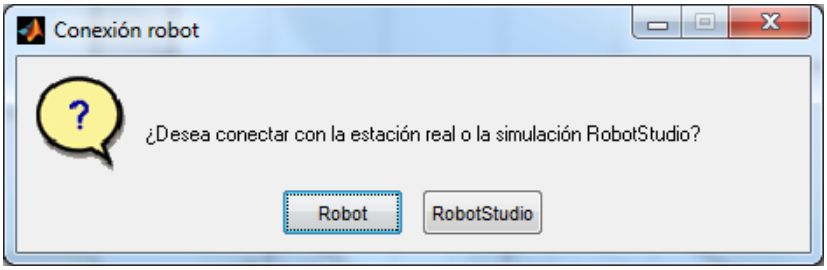


FIGURA M.7 Cuadro de diálogo de conexión con la estación del robot

A pesar de haber realizado una simulación de la trayectoria desde la plataforma Matlab, es aconsejable realizar otra simulación desde el software RobotStudio. Para ello debe desempaquetarse la estación *RS_PositionControlSocket_nicolas_ventosa* desde este software y reproducirse desde la pestaña de simulación. Una vez esto, podemos seleccionar la pestaña "*RobotStudio*" del cuadro de diálogo, de modo que nos aparecerá la siguiente barra de progreso mientras se lleva a cabo la conexión.

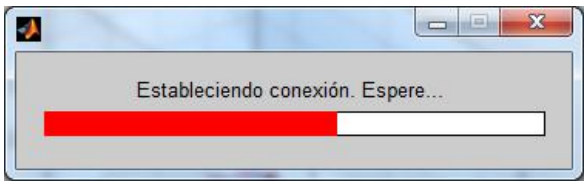


FIGURA M.8 Barra de progreso de establecimiento de conexión

Una vez establecida la conexión, es aconsejable ejecutar la trayectoria, pinchando en el botón "*Ejecutar Robot*" de la interfaz principal, este deberá cambiar al estado "*Detener ejecución*", y el robot comenzará a ejecutar la trayectoria. Tras confirmar este proceso, podemos detener la ejecución.

7. Para llevar a cabo la evitación de obstáculos, procedemos a la **conexión del sensor Kinect**. Una vez ubicado a una distancia entorno a 2 metros de la base del robot, sobre una superficie plana, debemos pulsar el botón "*Conectar Kinect*". Si nuestro ordenador ha instalado de forma correcta el dispositivo[7], aparecerá una nueva ventana, expuesta a continuación, mostrando una imagen tridimensional del entorno que percibe el campo de visión del sensor.

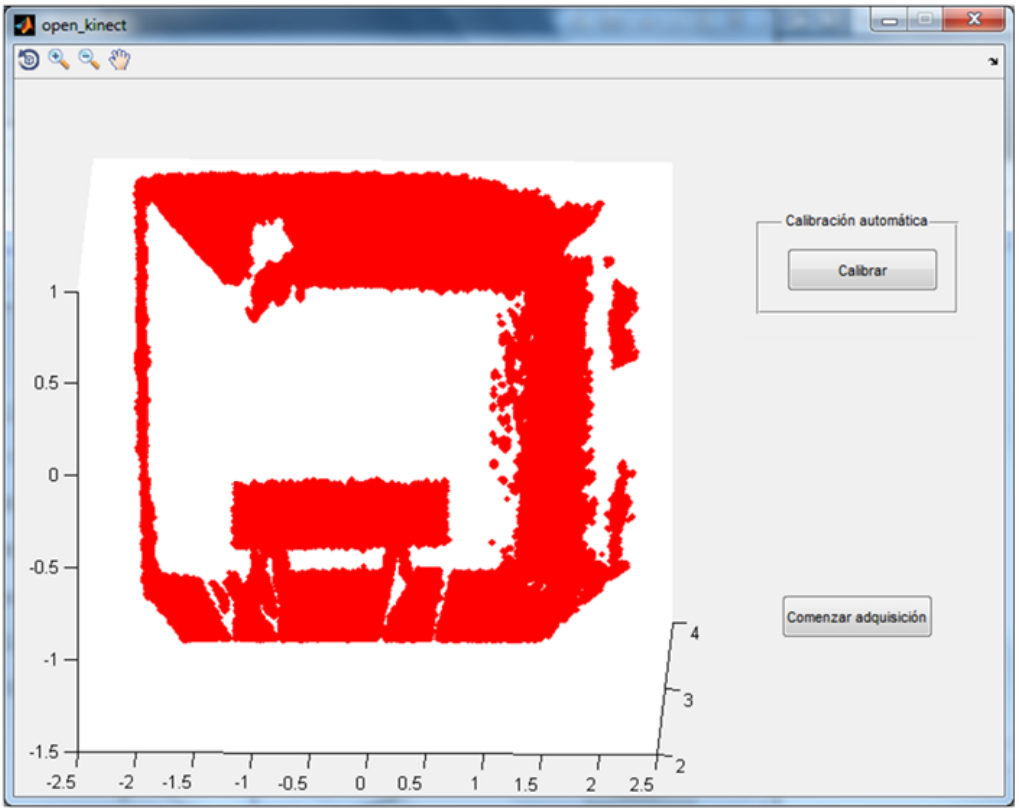


FIGURA M.9 Conexión y calibración del sensor Kinect

8. Para llevar a cabo la **calibración de la Kinect** con respecto al Lay-out del robot llevaremos a cabo un proceso de calibración automática empleando el tablero diana construido. Este debe ubicarse centrado, a 20 cm de la base del IRB120, entre sensor y

[7] Si se produce algún error en el flujo de programa, o no se muestra la imagen tridimensional del entorno en la nueva interfaz, consultar apartado *2.5.4 Interactuación con Kinect.*

robot. La aplicación realiza una búsqueda del blanco, y pregunta al usuario si la detección ha sido correcta.

FIGURA M.10 Proceso de calibración automática del sensor Kinect

Si la detección y calibración finalizan adecuadamente, aparecerá un cuadro de diálogo confirmándolo. En otro caso, se mostrará una ventana indicando un error en la calibración. A continuación queda ilustrado:

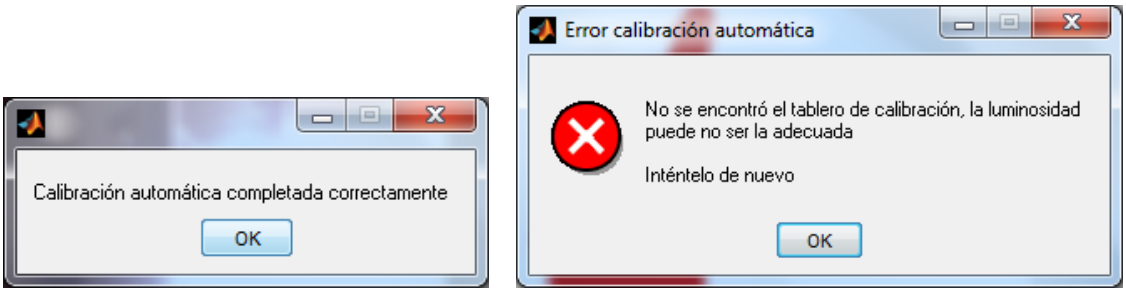


FIGURA M.11 Cuadros de diálogo de calibración automática del sensor Kinect

9. Tras concluir el proceso de calibración adecuadamente, se inicia la **adquisición de datos** mediante el sensor Kinect. Para ello el usuario debe pulsar el botón *Comenzar adquisición*. En este instante, el sensor comienza a obtener datos del área de trabajo del robot, y ejecuta un proceso de regeneración de trayectoria, en caso de detectar un obstáculo que interfiera en la trayectoria seleccionada por el usuario.

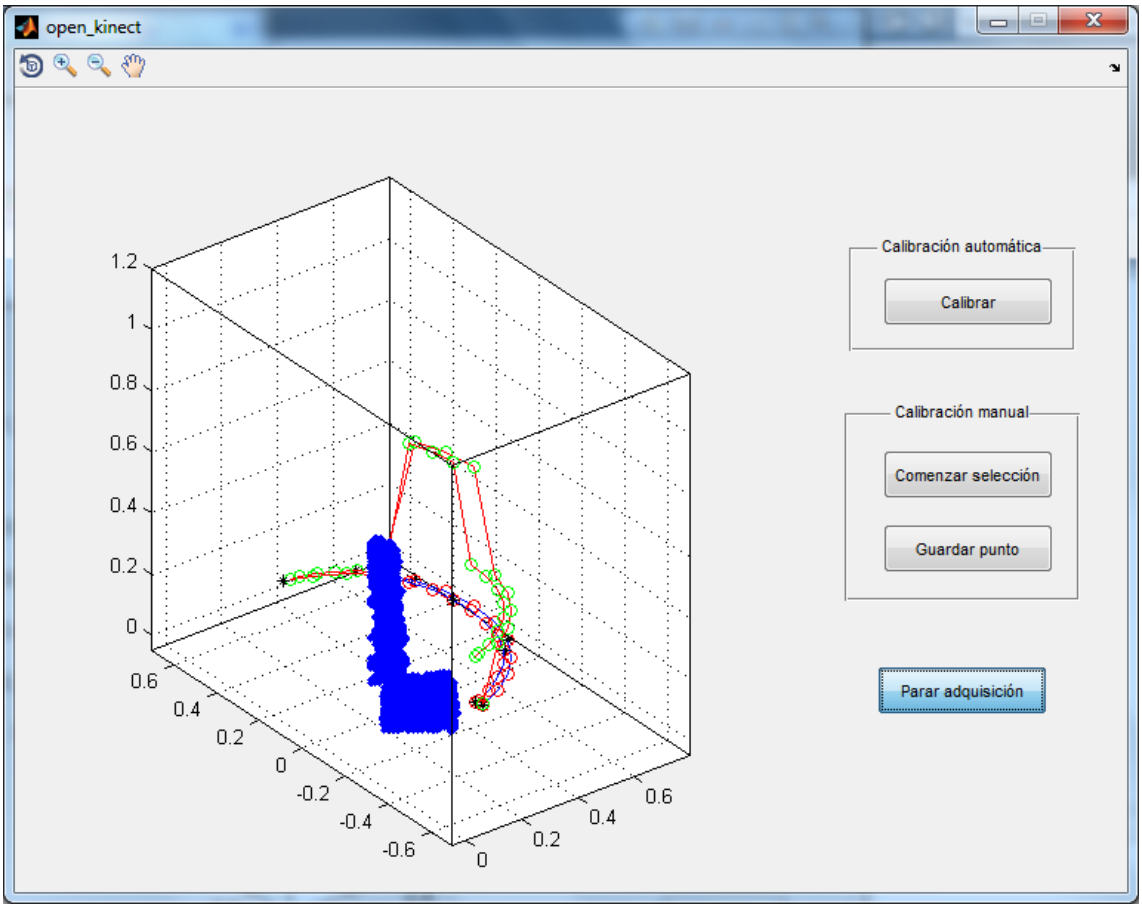


FIGURA M.12 Adquisición de datos y evitación de obstáculos mediante Kinect

10. Por último, podemos proceder a la **ejecución normal de la aplicación**. Se debe pulsar el botón *Ejecutar Robot* de la interfaz principal, pasando el botón a un estado de *Detener ejecución*, de modo que la estación reproduzca la trayectoria generada por el usuario de forma continuada. En caso de detectar el sensor Kinect un obstáculo que interfiera en esta, regenerará automáticamente la trayectoria y la enviará a la estación del robot. Si se desea conectar con la estación real del robot, simplemente se deberá volver a realizar la conexión mediante el botón *Conectar Robot*, escogiendo la opción *Robot* del cuadro de diálogo[8], y reproducir la trayectoria mediante el botón *Ejecutar Robot*.

11. Para **salir** del programa, el usuario deberá detener previamente la ejecución del robot pulsando el botón *Detener ejecución* de la interfaz principal y la adquisición de datos pulsando el botón *Parar adquisición* de la interfaz del sensor Kinect. Para cerrar el programa, simplemente debe pulsarse el botón *EXIT*, o el aspa de la barra de menús

[8] Ver figura *9.6 Cuadro de diálogo de conexión con la estación del Robot*

Cerrar. Aparecerá un cuadro de diálogo, donde el usuario deberá confirmar su deseo de finalizar la aplicación.

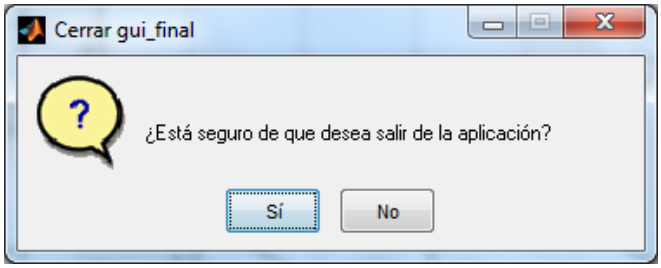


FIGURA M.13 Cuadro de diálogo de cierre de aplicación

Pliego de condiciones

En este capítulo se detallan las características de las herramientas utilizadas.

PL.1 Hardware

1. **Portátil Samsung NP530U3B**

- Procesador: Intel® Core™ i5
- RAM: 4 GB
- Tarjeta gráfica : Intel GMA HD Graphics 3000

2. **Robot Industrial ABB-IRB120**

- Peso: 25kg
- Altura: 580 mm
- Carga soportada: 3kg (hasta 4 Kg con muñeca vertical)
- Controlador: IRC5 Compact

3. **Microsoft Sensor Kinect**

PL.2 Software

- Windows 7 © Microsoft Corporation.
- ABB RobotStudio.
- Matlab R2014a (Con Toolbox de Robótica)
- Microsoft Office Home and Student 2010

Planos

En este capítulo se muestra el código de los archivos realizados tanto para las interfaces como para la aplicación final.

Debido a la extensión de los códigos desarrollados, sólo se indicarán los archivos de mayor transcendencia en este documento, el resto puede consultarse en el CD adjunto a este proyecto.

Función principal (main_func.m) Matlab

Esta es la función principal que permite definir todas las variables globales a emplear en el resto de funciones y además es la que se ejecuta para abrir el entorno de simulación desarrollado en este proyecto.

```
addpath('C:\Program Files\MATLAB\R2014a\toolbox\rvctools');
startup_rvc;

global q;   %Variable con las q
global puntos;  %Variable con los puntos de paso fijados por el
usuario
global mi_robot;
global top_points;
global front_lines;
global front_points;
global top_numeracion;
global front_numeracion;
global var_conexion_robotstudio;    %Variable que indica conexion con
RobotStudio
global var_aux_1; %Generar el borrado de los puntos en front_view si
fuese necesario
global trayectorias;    %Representacion grafica de trayectorias
global traslacion;  %Representacion tridimensional de la traslacion
del robot
global representacion;
global representacion_puntos;
global representacion_objetos;
global var_lim_aux; %Empleada en función regenerar trayectoria
global y_lim_aux;
```

```
global traslacion_aux;
global ejecutando_robot;
global kinect_esta_adquiriendo;
global objeto;

var_conexion_robotstudio=0;
var_aux_1=0;
var_lim_aux=0;
ejecutando_robot=0;
kinect_esta_adquiriendo=0;
objeto=0;

% Creamos el robot
l1=0.290; l2=0.270; l3=0.070; l4=0.302; l5=0.072;
%Formacion del "linkado" de cada uno de los eslabones del robot
L(1)=Link([0 l1 0 -pi/2]);
L(2)=Link([0 0 l2 0]);
L(3)=Link([0 0 l3 -pi/2]);
L(4)=Link([0 l4 0 pi/2]);
L(5)=Link([0 0 0 -pi/2]);
L(6)=Link([0 l5 0 0]);

irb_120=SerialLink(L,'name','Robot Nicolas Blanco');

irb_120.offset=[0 -pi/2 0 0 0 pi];
irb_120.tool=transl(0,0,0.16);

global irb_120;


gui_final
```

Interfaz principal (gui_final.m) Matlab

Esta función compone cada una de las funciones de la interfaz principal de la aplicación, desarrollada mediante la herramienta Guide. Entre sus funciones principales se encuentra la generación de trayectorias y el envío de estas a la estación del robot.

```
function varargout = gui_final(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @gui_final_OpeningFcn, ...
                   'gui_OutputFcn',  @gui_final_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
```

```
end


%Función que fija las dimensiones de la mesa y la zona de trabajo del
robot
function [Largo,Ancho,Alto]=dimensiones

Largo=1;
Ancho=0.6;
Alto=1;


function gui_final_OpeningFcn(hObject, eventdata, handles, varargin)
handles.output = hObject;
guidata(hObject, handles);


function varargout = gui_final_OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;


function reset_Callback(hObject, eventdata, handles)
global top_points;
global front_lines;
global front_points;
global top_numeracion;
global front_numeracion;
global trayectorias;
global representacion;
global az;
global el;
global representacion_puntos;
global representacion_objetos;

set (top_points,'visible','off');
set (top_numeracion,'visible','off');

set (front_lines,'visible','off');
set (front_points,'visible','off');
set (front_numeracion,'visible','off');

set (trayectorias,'visible','off');

rotate3d(handles.tridimensional,'off');
axes(handles.tridimensional);
hold off;
plot3(0,0,0);
axis equal;axis([-0.05 0.8 -0.7 0.7 -0.05 1.2]);
box on; grid on;


function generar_trayectoria_articular_Callback(hObject, eventdata,
handles)

global puntos;
global q;
global trayectorias;
```

```
global irb_120;
global traslacion;
global representacion;
global representacion_puntos;

clear T;
for i=1:length(puntos(:,1))
    T(:,:,i) = Calcula_T(puntos(i,1),puntos(i,2),puntos(i,3));
end

%Aplicamos la cinemática inversa
q_puntos_via=irb120_ikine(T,[0 0 0 0 0 0]);

duracion_segmento=2;
puntos_por_segmento=4;

%Interpolamos esta trayectoria
q=mstraj(q_puntos_via,[],duracion_segmento*ones(1,length(puntos(:,1)))
,q_puntos_via(1,:),duracion_segmento/puntos_por_segmento,1);

Tseguida=irb_120.fkine(q);
traslacion=transl(Tseguida);
traslacion=traslacion';

axes(handles.Articulaciones);
trayectorias=plot(handles.Articulaciones,q);
axis equal;
box on;grid on;

axes(handles.tridimensional);
axis equal;
box on;grid on
hold off;
representacion=plot3(traslacion(1,:),traslacion(2,:),traslacion(3,:),'
ro', ...
      traslacion(1,:),traslacion(2,:),traslacion(3,:),'b');
hold on
representacion_puntos=plot3(puntos(:,1),puntos(:,2),puntos(:,3),'k*');
axis equal;
axis([-0.05 0.8 -0.7 0.7 -0.05 1.2]);
box on; grid on;
hold off;


function simular_trayectoria_articular_Callback(hObject, eventdata,
handles)

global q;
global puntos;
global irb_120;
global traslacion;

clear T;
for i=1:length(puntos(:,1))
    T(:,:,i) = Calcula_T(puntos(i,1),puntos(i,2),puntos(i,3));
end

%Aplicamos la cinemática inversa
q_puntos_via=irb120_ikine(T,[0 0 0 0 0 0]);
```

```
duracion_segmento=2;
puntos_por_segmento=4;

%Interpolamos esta trayectoria
q=mstraj(q_puntos_via,[],duracion_segmento*ones(1,length(puntos(:,1)))
,q_puntos_via(1,:),duracion_segmento/puntos_por_segmento,1);

Tseguida=irb_120.fkine(q);
traslacion=transl(Tseguida);
traslacion=traslacion';

simulacion_robot;


function ejecutar_robot_Callback(hObject, eventdata, handles)

global q;
global mi_robot;
global var_conexion_robotstudio;
global ejecutando_robot;
global var_lim_aux;
global kinect_esta_adquiriendo;
global objeto;

if var_conexion_robotstudio==0
    set(hObject,'Value',0);%Mantenemos el botón sin pulsar
    msgbox('Debe establecer la conexión con la estación del robot
previamente');
    return;
end
estado_boton=get(hObject,'Value');
%Modificamos el string del boton dependiendo de si está pulsado o no.
Por
%defecto en la guide está definido como 'Comenzar adquisición'
if estado_boton
    set(hObject,'String','Detener
ejecución','BackgroundColor','r','ForegroundColor','w');
    ejecutando_robot=1;
    var_lim_aux=0;
else
    set(hObject,'String','Ejecutar
Robot','BackgroundColor','default','ForegroundColor','k');
    ejecutando_robot=0;
    mi_robot.stoprobot;
    pause(1);

    return;
end

%Ejecutamos robot
q_deg=q.*180/pi; %Lo pasamos de radianes a grados para que lo ejecute
el robot

%Lo ordenamos en un solo vector
contador=1;
for i=1:length(q_deg)
    q_deg_vector(contador:(contador+5))=q_deg(i,:);
    contador=(i*6)+1;
```

```
    a=i;
end

if kinect_esta_adquiriendo==1
    if objeto==0
        disp('enviaaaausuario11')
        mi_robot.JointTrajectory(q_deg_vector);
    else
        var_lim_aux=0;
        regenerar_trayectoria;
    end
else
    disp('enviaaaausuario')
    mi_robot.JointTrajectory(q_deg_vector);
end



function Top_view_CreateFcn(hObject, eventdata, handles)

[Largo,Ancho,Alto]=dimensiones;
set(gca,'Ydir','reverse');%Para concordancia puntos con Front view
invertimos los ejes
axis equal;axis manual;grid on; box on;
axis([-Largo/2 Largo/2 0 Ancho]);hold on;
rectangle('Position',[-0.062,0,0.124,0.22],'FaceColor',[1,0.5,0]);
rectangle('Position',[-
0.1095,0.01,0.0475,0.180],'FaceColor',[1,0.5,0]);
rectangle('Position',[0.062,0.01,0.0475,0.180],'FaceColor',[1,0.5,0]);
rectangle('Position',[-0.05,0.22,0.1,0.008],'FaceColor',[1,0.5,0]);
rectangle('Position',[-0.062,0.228,0.124,0.23],'FaceColor',[1,0.5,0]);
rectangle('Position',[-
0.05,0.458,0.1,0.006],'FaceColor',[0.4,0.4,0.4]);

plot(0,0.464,'ro');

dibujo=guidata(hObject);



function Front_view_CreateFcn(hObject, eventdata, handles)

[Largo,Ancho,Alto]=dimensiones;
axis equal;axis manual;grid on;
axis([-Largo/2 Largo/2 0 Alto]);hold on; box on;
rectangle('Position',[-0.090,0,0.180,0.190],'FaceColor',[1,0.5,0]);
rectangle('Position',[-
0.062,0.190,0.124,0.150],'FaceColor',[1,0.5,0]);
rectangle('Position',[-
0.1095,0.230,0.0475,0.4],'FaceColor',[1,0.5,0]);
rectangle('Position',[0.062,0.230,0.0475,0.4],'FaceColor',[1,0.5,0]);
rectangle('Position',[-
0.062,0.56,0.124,0.14],'Curvature',[0.4],'FaceColor',[1,0.5,0]);
rectangle('Position',[-0.05 0.58 0.1
0.1],'Curvature',[1,1],'FaceColor',[0.4,0.4,0.4]);
plot(0,0.630,'ro');



function Top_view_ButtonDownFcn(hObject, eventdata, handles)

global puntos;
```

```
global top_points;
global front_lines;
global front_points;
global top_numeracion;
global var_aux_1;
global front_numeracion;

set (top_points,'visible','off');
set (top_numeracion,'visible','off');

set (front_lines,'visible','off');
set (front_points,'visible','off');
set (front_numeracion,'visible','off');

[x,y]=ginputax(handles.Top_view,10);
%Sleccionamos los puntos pertenencientes al interior del grafico
[Largo,Ancho,Alto]=dimensiones;
contador=1;
for i=1:length(x)
    if x(i)>-Largo/2 && x(i)<Largo/2 && y(i)>0 && y(i)<Ancho
        Y(contador)=x(i,1);
        X(contador)=y(i,1);
        contador=contador+1;
    end
end
numeracion=cellstr(num2str([1:length(X)]'));%Convierte el array en
array de celulas de cadenas de caracteres
top_points=plot(Y,X,'b*');
top_numeracion=text(Y,X, numeracion, 'VerticalAlignment','bottom', ...
                            'HorizontalAlignment','right');

for i=1:length(Y)
    axes(handles.Front_view)
    front_lines(i)=plot([Y(i) Y(i)],[0 1],'--g');
    front_numeracion(i)=text(Y(i),0.9, numeracion(i),
'VerticalAlignment','bottom', ...
                            'HorizontalAlignment','right');
end
Z=0.2*ones(size(X)); %Por defecto z=0.2

var_aux_1=0;
puntos=[X' Y' Z'];


function Front_view_ButtonDownFcn(hObject, eventdata, handles)

global puntos;
global front_points;
global var_aux_1;

if var_aux_1==1
    set (front_points,'visible','off');
end

[x,y]=ginputax(handles.Front_view,10);
%Sleccionamos los puntos pertenencientes al interior del grafico
[Largo,Ancho,Alto]=dimensiones;
contador=1;
for i=1:length(x)
    if x(i)>-Largo/2 && x(i)<Largo/2 && y(i)>0 && y(i)<Alto
```

```
        X(contador)=x(i);
        Z(contador)=y(i);
        contador=contador+1;
    end
end
front_points=plot(puntos(:,2),Z,'b*');
var_aux_1=1;
puntos(:,3)=Z;


function Articulaciones_CreateFcn(hObject, eventdata, handles)

axis equal;
box on;grid on;


function text4_CreateFcn(hObject, eventdata, handles)


function exit_Callback(hObject, eventdata, handles)

close(gui_final);%Al generar una confirmacion para cerrar la gui, no
es necesario volver a realizarla ahora
close all;


function conexion_robotstudio_Callback(hObject, eventdata, handles)
% hObject    handle to conexion_robotstudio (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global mi_robot
global var_conexion_robotstudio;
var_conexion_robotstudio=1;

set(handles.ejecutar_robot,'Value',0);%En caso de que estuviese el
boton de ejecución pulsado
set(handles.ejecutar_robot,'String','Ejecutar Robot');

confirmacion = questdlg('¿Desea conectar con la estación real o la
simulación RobotStudio?','Conexión
robot','Robot','RobotStudio','Robot');
switch confirmacion
    case 'Robot'
        mi_robot=irb120('172.29.28.185',1024);
    case 'RobotStudio'
        mi_robot=irb120('127.0.0.1',1024);
end

mi_robot.connect;
ventana = waitbar(0,'Estableciendo conexión. Espere...');
steps = 5000;
for step = 1:steps
    waitbar(step / steps)
end
close(ventana)


% --- Executes on button press in conexion_robot.
function conexion_robot_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to conexion_robot (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global mi_robot

mi_robot.stoprobot;




% --- Executes when user attempts to close figure1.
function figure1_CloseRequestFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global mi_robot;
global var_conexion_robotstudio;

confirmacion = questdlg('¿Está seguro de que desea salir de la
aplicación?','Cerrar gui_final','Sí','No','Sí');
switch confirmacion
    case 'Sí'
        if var_conexion_robotstudio==1
            mi_robot.disconnect;
        end
        delete(hObject);
        close all;
    case 'No'
        return;
end



function generar_trayectoria_cartesiana_Callback(hObject, eventdata,
handles)

global puntos;
global q;
global trayectorias;
global irb_120;
global traslacion;
global representacion;
global representacion_puntos;

puntos=puntos';
tam=size (puntos);
F=spline((1:tam(1,2)),puntos);

% Trajectory
step=0.2;
t=[1:step:tam(1,2)];
traslacion=ppval(F,t);

clear T;
for i=1:length(traslacion)
    T(:,:,i) =
Calcula_T(traslacion(1,i),traslacion(2,i),traslacion(3,i));
end

q=irb120_ikine(T,[0 0 0 0 0 0]);
```

```
axes(handles.Articulaciones);

trayectorias=plot(handles.Articulaciones,q);
axis equal;
box on;grid on

axes(handles.tridimensional);
axis equal;
box on;grid on
hold off;
representacion=plot3(traslacion(1,:),traslacion(2,:),traslacion(3,:),'
ro', ...
      traslacion(1,:),traslacion(2,:),traslacion(3,:),'b');
hold on;
representacion_puntos=plot3(puntos(1,:),puntos(2,:),puntos(3,:),'k*');
axis equal;
axis([-0.05 0.8 -0.7 0.7 -0.05 1.2]);
box on; grid on;
hold off;

puntos=puntos';



function simular_trayectoria_cartesiana_Callback(hObject, eventdata,
handles)

global puntos;
global q;
global trayectorias;
global traslacion;
global irb_120;

puntos=puntos';
tam=size (puntos)
F=spline((1:tam(1,2)),puntos);

% Trajectory
step=0.2;
t=[1:step:tam(1,2)];
traslacion=ppval(F,t);

for i=1:length(traslacion)
    T(:,:,i) =
Calcula_T(traslacion(1,i),traslacion(2,i),traslacion(3,i));
end

q=irb120_ikine(T,[0 0 0 0 0 0]);

puntos=puntos';
simulacion_robot;



function tridimensional_CreateFcn(hObject, eventdata, handles)

global az;
global el;
```

```
axis equal;
axis([-0.05 0.8 -0.7 0.7 -0.05 1.2]);

hold on;
set(gca,'XLim',[-0.05 0.8]);
set(gca,'YLim',[-0.7 0.7]);
set(gca,'ZLim',[-0.05 1.2]);
box on; grid on;


function conectar_kinect_Callback(hObject, eventdata, handles)

open_kinect
```

Interfaz de simulación de trayectoria (simulación_robot.m) Matlab

Esta función compone una interfaz auxiliar de la aplicación que permite la simulación de la trayectoria generada en la plataforma Matlab, empleando el modelo del IRB120 desarrollado.

```
function varargout = simulacion_robot(varargin)

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @simulacion_robot_OpeningFcn, ...
                   'gui_OutputFcn',  @simulacion_robot_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end


function simulacion_robot_OpeningFcn(hObject, eventdata, handles,
varargin)

handles.output = hObject;
guidata(hObject, handles);


function varargout = simulacion_robot_OutputFcn(hObject, eventdata,
handles)

varargout{1} = handles.output;


function simular_Callback(hObject, eventdata, handles)
```

```
global q;
global irb_120

hold on;
irb_120.plot(q)


function simulacion_tridimensional_CreateFcn(hObject, eventdata,
handles)

global irb_120;
global traslacion;
global q;

% Representación trayectoria
plot3(traslacion(1,:),traslacion(2,:),traslacion(3,:),'ro', ...
      traslacion(1,:),traslacion(2,:),traslacion(3,:),'b');

irb_120.plot([0 0 0 0 0 0]);
```

Interfaz conexión y adquisición de datos con Kinect (open_kinect.m) Matlab

Esta función permite la conexión con el sensor Kinect, su calibración, y la adquisición de las nubes de puntos, así como la detección de aquellos objetos presentes en la superficie de trabajo que se interpongan en la trayectoria del robot.

```
function varargout = open_kinect(varargin)

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @open_kinect_OpeningFcn, ...
                   'gui_OutputFcn',  @open_kinect_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end



function open_kinect_OpeningFcn(hObject, eventdata, handles, varargin)
% Choose default command line output for open_kinect
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
```

```
function varargout = open_kinect_OutputFcn(hObject, eventdata,
handles)

varargout{1} = handles.output;

function axes1_CreateFcn(hObject, eventdata, handles)

handles=guidata(hObject)
eje_axis1=gca;
refrescar_tag = get(eje_axis1, 'Tag');

depthVid=videoinput('kinect',2,'Depth_640x480');%Acepta
'Depth_320x240' 'Depth_640x480' 'Depth_80x60'
depthImage=getsnapshot(depthVid);

%Control ángulo
srcDepth = getselectedsource(depthVid);
set(srcDepth, 'CameraElevationAngle', 0);%Puede variar de -27 a 27
grados

%Ajustar modo de disparo a "manual", el disparo se produce cuando
ejecutemos la función
triggerconfig(depthVid,'manual');

%Ajustar la adquisición de frames por disparo suficientes para
realizar el seguimiento
depthVid.FramesPerTrigger=1;

%Iniciar dispositivo de profundidad. Esto comienza la adquisición pero
no registra los datos adquiridos
start(depthVid);

%Comenzamos el registro de los datos adquiridos
trigger(depthVid);

%Obtenemos los datos adquiridos
[depthFrameData, depthTimeData, depthMetaData]=getdata(depthVid);

%Detenemos los dispositivos
stop(depthVid)

%Cálculo nube de puntos
[X Y Z]=depthtoxyz(depthFrameData);

plot3(X,Y,Z,'r.')
set(eje_axis1, 'Tag', refrescar_tag);


function comenzar_adquisicion_Callback(hObject, eventdata, handles)

global X;
global Y;
global Z;
global posicion;
global abc;
global xlim;
global zlim;
global x_lim;
global y_lim;
```

```
global z_lim;
global puntos_objetos;
global traslacion;
global puntos;
global representacion;
global representacion_puntos;
global representacion_objetos;
global y_lim_aux;
global var_lim_aux;
global ejecutando_robot;
global kinect_esta_adquiriendo;
global objeto;
global x_min;
global x_max;
global y_min;
global y_max;
global z_min;
global estado_boton_adquisicion;
global depthVid;

enviada_trayectoria_usuario=1;

estado_boton_adquisicion=get(hObject,'Value');

%Modificamos el string del boton dependiendo de si está pulsado o no.
Por defecto en la guide está definido como 'Comenzar adquisición'
if estado_boton_adquisicion
    set(hObject,'String','Parar adquisición');
    var_lim_aux=0;
    kinect_esta_adquiriendo=1;
else
    set(hObject,'String','Comenzar adquisición');
    kinect_esta_adquiriendo=0;
end

if estado_boton_adquisicion==get(hObject,'Max')
depthVid=videoinput('kinect',2,'Depth_640x480');

%Control ángulo
srcDepth = getselectedsource(depthVid);
set(srcDepth, 'CameraElevationAngle', 0);%Puede variar de -27 a 27
grados

%Ajustar modo de disparo a "manual", el disparo se produce cuando
ejecutemos la función
triggerconfig(depthVid,'manual');

%Ajustar la adquisición de frames por disparo suficientes para
realizar el seguimiento
depthVid.FramesPerTrigger=1;
depthVid.TriggerRepeat=inf;

%Iniciar dispositivo de color y profundidad. Esto comienza la
adquisición pero no registra los datos adquiridos
start(depthVid);
end

representacion=plot3(traslacion(1,:),traslacion(2,:),traslacion(3,:),'
ro', ...
    traslacion(1,:),traslacion(2,:),traslacion(3,:),'b');
```

```
hold on;
representacion_puntos=plot3(puntos(:,1),puntos(:,2),puntos(:,3),'k*');
axis equal;
axis([-0.05 0.8 -0.7 0.7 -0.05 1.2]);
box on; grid on;
hold off;
while estado_boton_adquisicion==get(hObject,'Max')

%Comenzamos el registro de los datos adquiridos
trigger(depthVid);

%Obtenemos los datos adquiridos
[depthFrameData, ~, ~]=getdata(depthVid);

%Cálculo nube de puntos
[X Y Z]=depthtoxyz(depthFrameData);

%Delimitamos el reconocimiento de objetos en el área de trabajo del
robot
X(X<x_min)=nan;
X(X>x_max)=nan;
Y(Y<y_min)=nan;
Y(Y>y_max)=nan;
Z(Z<(z_min+0.02))=nan;%Damos cierto margen para no tener en cuenta la
superficie de la mesa

%A continuación se realiza el cambio de coordenadas de la Kinect a las
del brazo robot. Para realizar la transformación de coordenadas, es
más eficiente transformar las matrices X Y Z a un solo vector y
aplicar homtrans que realizarlo con un bucle
a=X';A=a(:)';
b=Y';B=b(:)';
c=Z';C=c(:)';
matriz=[A;B;C];
T=transl(0,y_max,0)*trotz(180,'deg')*transl(-((x_max-
x_min)/2+x_min),0,0);
abc=homtrans(T,matriz);

objeto=0;

[tamx tamy]=size(abc);
xlim=[];
puntos_objetos=[nan;nan;nan];

for j=1:tamy

    if abc(3,j)>(z_min+0.02) && abc(3,j)<(z_min+0.03)
        objeto=1;
        var=1;
        matriz_x_lim=[];
        matriz_x_ordenada=[];
        x_lim=[];
        y_lim=[];
        z_lim=[];
        abc_ordenada_x=[];
        x_ordenada=[];

        %Calculamos nube de puntos sobre superfice de trabajo
        for k=1:length(abc)
```

```
            if abc(3,k)>(z_min+0.03) && abc(3,k)<(z_min+0.05)
                matriz_x_lim(:,var)=abc(:,k);
                var=var+1;
            end
        end
%En caso de que detecte algún objeto entre la kinect y el lay-out del
robot, sin detectar objeto sobre la superficie de trabajo:
        %Salimos del bucle
        if length(matriz_x_lim)==0
            break;
        end
%Ordenamos la matriz y obtenemos los puntos límite entre objetos en x
e y
        matriz_x_ordenada=(sortrows(matriz_x_lim',1))';
        %En caso de que la longitud de los puntos sea menor que 5 lo
        %consideramos como ruido y salimos del bucle
        if length(matriz_x_ordenada)<=5
            break;
        end

        var=1;
        var_2=1;
        ruido=0;
        x_lim(:,var)=matriz_x_ordenada(:,1);
        for k=2:length(matriz_x_ordenada)
            if matriz_x_ordenada(1,k)>(matriz_x_ordenada(1,k-1)+0.05)
                %Eliminamos los puntos de ruido en caso que aparezcan
                if (k-var_2)==1
                    ruido=1;
                else
                    if k==length(matriz_x_ordenada)
                        ruido=1;
                        break
                    end
                    var=var+2;
                    x_lim(:,var-1)=matriz_x_ordenada(:,k-1);
                    x_lim(:,var)=matriz_x_ordenada(:,k);
                    y_lim(var-1)=min(matriz_x_ordenada(2,var_2:k-1));
                    y_lim(var-2)=max(matriz_x_ordenada(2,var_2:k-1));
                    var_2=k;
                    ruido=0;
                end
            end
        end
%Eliminamos ruido en caso de serlo el último punto de la matriz
        if ruido==0
            x_lim(:,var+1)=matriz_x_ordenada(:,end);
            y_lim(var+1)=min(matriz_x_ordenada(2,var_2:end));
            y_lim(var)=max(matriz_x_ordenada(2,var_2:end));
        elseif ruido==1
            x_lim(:,var+1)=matriz_x_ordenada(:,end-1);
            y_lim(var+1)=min(matriz_x_ordenada(2,var_2:end-1));
            y_lim(var)=max(matriz_x_ordenada(2,var_2:end-1));
        end

%Almacenamos únicamente los puntos que se encuentran dentro de los
límites de x de los objetos (obtenidos anteriormente)
        abc_ordenada_x=(sortrows(abc',1))';
        x_ordenada=abc_ordenada_x(1,:);
        abc_ordenada_x(1,:)=x_ordenada;
        var=1;
```

```
        var_2=1;
        tam_puntos_x=[];
        tam_puntos_y=[];
        puntos_x=[];
        puntos_y=[];

        for k=1:length(x_lim)/2
            for l=1:length(abc)
                if(abc_ordenada_x(1,l)>=x_lim(1,var) &&
abc_ordenada_x(1,l)<=x_lim(1,var+1))
                    puntos_x(:,var_2)=abc_ordenada_x(:,l);
                    var_2=var_2+1;
                end
            end
%Número de puntos de cada objeto será tam_puntos_x(i)=tam_puntos_x(i)-
tam_puntos_x(i-1)
            tam_puntos_x(k)=length(puntos_x);
            if tam_puntos_x(k)==0
                if k==1
                    tam_puntos_x(k)=1;
                else
                    tam_puntos_x(k)=tam_puntos_x(k-1);
                end
            end
            var=var+2;
        end

        %Almacenamos sobre los anteriores únicamente los puntos que se
        encuentran dentro de los límites de y.
        var=1;
        var_2=1;
        for k=1:length(tam_puntos_x)
            if k==1
                for l=1:tam_puntos_x(k)
                    if (puntos_x(2,l)<=y_lim(var_2) &&
puntos_x(2,l)>=y_lim(var_2+1))
                        puntos_y(:,var) = puntos_x(:,l);
                        var=var+1;
                     end
                end
                var_2=var_2+2;
                tam_puntos_y(k)=length(puntos_y);
            elseif k>1
                for l=tam_puntos_x(k-1):tam_puntos_x(k)
                    if (puntos_x(2,l)<=y_lim(var_2) &&
puntos_x(2,l)>=y_lim((var_2)+1))
                        puntos_y(:,var) = puntos_x(:,l);
                        var=var+1;
                    end
                end
                var_2=var_2+2;
                tam_puntos_y(k)=length(puntos_y);
            end
        end

        %Comprobamos si hay algún objeto sobre el cuerpo situado en la
        superficie de trabajo para desecharle

        var=1;
        puntos_objetos=[];
        for k=1:length(tam_puntos_y)
```

```
            puntos_z_aux=[];
            z_ordenada=[];
            if k==1
                puntos_z_aux=puntos_y(:,1:tam_puntos_y(k));
            elseif k>1
                puntos_z_aux=puntos_y(:,tam_puntos_y(k-
1):tam_puntos_y(k));
            end
%En caso de que haya un único punto, tomamos su z como altura
            if length(1:tam_puntos_y(k))==0
                z_lim(k)=x_lim(3,k*2);
                if k==1
                    tam_puntos_y(k)=1;
                else
                    tam_puntos_y(k)=tam_puntos_y(k-1);
                end
            else
                z_ordenada=(sortrows(puntos_z_aux',3))';
                for l=1:(length(z_ordenada)-1)
                    [z_row z_col]=size(z_ordenada);
%En caso de que haya algún ruido, añadimos esta condición para evitar
que dé error el programa
                    if z_col==1
                        z_lim(k)=z_ordenada(3,1);
                        var=var+l;
                        break;
                    end
                    if (abs((z_ordenada(3,l))-z_ordenada(3,l+1))>0.02
|| l==length(z_ordenada)-1)
                        z_lim(k)=z_ordenada(3,l);
                        puntos_objetos(:,var:(l-
1+var))=z_ordenada(:,1:l);
                        var=var+l;
                        break
                    end
                end
            end
        end
        T=transl(0,0,-z_min);
        puntos_objetos=homtrans(T,puntos_objetos);
        %Pasamos a coordenadas del robot; ya que la coordenada y es la
        coordenada x empleada por nosotros y viceversa
        puntos_objetos=[puntos_objetos(2,:);-
puntos_objetos(1,:);puntos_objetos(3,:)];
        %También pasamos a coordenadas del robot los puntos límites de
      los objetos obtenidos anteriormente
        x_lim_aux=y_lim;%Para no machacar la variable x_lim
        y_lim=-x_lim(1,:);
        x_lim=x_lim_aux;
        z_lim=z_lim-z_min;

        enviada_trayectoria_usuario=0;
        trayectoria_enviada=1;
        regenerar_trayectoria;
        break;

    end
    trayectoria_enviada=0;
end

if objeto==0
```

```
        axes(handles.axes1);
        hold off;

representacion=plot3(traslacion(1,:),traslacion(2,:),traslacion(3,:),'
ro', ...
        traslacion(1,:),traslacion(2,:),traslacion(3,:),'b');
        hold on;

representacion_puntos=plot3(puntos(:,1),puntos(:,2),puntos(:,3),'k*');
        axis equal;
        axis([-0.05 0.8 -0.7 0.7 -0.05 1.2]);
        set(gca,'XLim',[-0.05 0.8]);
        set(gca,'YLim',[-0.7 0.7]);
        set(gca,'ZLim',[-0.05 1.2]);
        box on; grid on;
end



if trayectoria_enviada==0 && enviada_trayectoria_usuario==0 &&
ejecutando_robot==1
    q_deg=q.*180/pi; %Lo pasamos de radianes a grados para que lo
ejecute RobotStudio

    %Lo ordenamos en un solo vector
    contador=1;
    for i=1:length(q_deg)
        q_deg_vector(contador:(contador+5))=q_deg(i,:);
        contador=(i*6)+1;
        a=i;
    end
    disp('enviando usuario')
    mi_robot.JointTrajectory(q_deg_vector);
    enviada_trayectoria_usuario=1;
    var_lim_aux=0;
end

estado_boton_adquisicion=get(hObject,'Value');
if estado_boton_adquisicion==get(hObject,'Min')
    %Detenemos los dispositivos
    stop(depthVid)
end

end
disp('Programa finalizado con exito')



function calibracion_automatica_Callback(hObject, eventdata, handles)

global x_min;
global x_max;
global y_min;
global y_max;
global z_min;
%Create videoinput object for color stream
colorVid=videoinput('kinect',1,'RGB_640x480');
f=figure(1);
set(f,'name','Proceso de calibración automática','numbertitle','off')
colorImage=getsnapshot(colorVid);
imshow(colorImage,[0 4000]);
```

```
%d = imdistline;%Para determinar el radio del circulo
[centers, radii] = imfindcircles(colorImage,[7
13],'ObjectPolarity','dark', ...
    'Sensitivity',0.92);%Fijamos la sensibilidad a 0.92, esta varía
entre 0 y 1

[length_row length_column rgb]=size(colorImage);%Tamaño imagen

centers_integer=int32(centers);%Convertimos el centro obtenido a
entero

[num_centers length_vector]=size(centers);

%Eliminamos círculos en bordes de la imagen en caso de que aparezca
alguno, pues estos nos pueden dar errores de ejecución y el buscado
estará centrado

centers_aux=[];
j=0;
for i=1:num_centers
    if centers(i,1)<16 || centers(i,1)>(length_column-16) ||
centers(i,2)<16 || centers(i,2)>(length_row-16)

    else
        j=j+1;
        centers_aux(j,:)=centers(i,:);
    end
end
centers_integer=int32(centers_aux);

pix_verde=0;%Variable para confirmar que se han detectado varios
pixeles verdes dentro del circulo
pix_rojo=0;
diana=0;
[num_centers_integer length_vector]=size(centers_integer);

%Buscamos el círculo deseado mediante búsquedas de colores rgb
for i=1:num_centers_integer
    for j=centers_integer(i,1)-10:centers_integer(i,1)+10
        for k=centers_integer(i,2)-10:centers_integer(i,2)+10
            if colorImage(k,j,1)>=81 & colorImage(k,j,1)<=210 &
colorImage(k,j,2)>=150 & colorImage(k,j,2)<=260 &
colorImage(k,j,3)>=78 & colorImage(k,j,3)<=240
                pix_verde=pix_verde+1;
                if pix_verde>=10
                    for m=centers_integer(i,1)-
15:centers_integer(i,1)+15
                        for n=centers_integer(i,2)-
15:centers_integer(i,2)+15
                            if colorImage(n,m,1)>=100 &
colorImage(n,m,1)<=230 & colorImage(n,m,2)>=20 &
colorImage(n,m,2)<=110 & colorImage(n,m,3)>=130 &
colorImage(n,m,3)<=230
                                pix_rojo=pix_rojo+1;
                                if pix_rojo==8

diana_circulo=viscircles(centers(i,:), radii(i,:),'EdgeColor','r');
                                    hold on;
```

```
diana_asterisco=plot(centers(i,1),centers(i,2),'b*');
                                    hold off
                                    confirmacion = questdlg('¿Se ha
realizado la calibración correctamente?','Confirmación calibración
Kinect','Sí','No','Sí');
                                    switch confirmacion
                                        case 'Sí'
                                            diana=1;

center=centers_integer(i,:);

                                        case 'No'
                                            diana=2;
                                            set
(diana_circulo,'visible','off');
                                            set
(diana_asterisco,'visible','off');
                                    end
                                end
                            end
                            if diana>0
                                break;
                            end
                        end
                        if diana>0
                            break;
                        end
                    end
                end
            end
            if diana>0
                break;
            end
        end
        if diana>0
            break;
        end
    end
    if diana==1
        break;
    elseif diana==2
        diana=0;
        pix_verde=0;
        pix_rojo=0;
    end
end

if diana==0
    waitfor(msgbox({'No se encontró el tablero de calibración, la
luminosidad' 'puede no ser la adecuada' '' 'Inténtelo de nuevo'
''},'Error calibración automática','error'));
    close(f)
    return
end
%Una vez obtenido el centro en la imagen de color, obtenemos los datos
de dicho pixel en la imagen de profundidad
depthVid=videoinput('kinect',2,'Depth_640x480');
depthImage=getsnapshot(depthVid);
```

```
%Ajustar modo de disparo a "manual", el disparo se produce cuando
ejecutemos la función
triggerconfig(depthVid,'manual');

%Ajustar la adquisición de frames por disparo suficientes para
realizar el seguimiento
depthVid.FramesPerTrigger=1;

%Iniciar dispositivo de color y profundidad. Esto comienza la
adquisición pero no registra los datos adquiridos
start(depthVid);

%Comenzamos el registro de los datos adquiridos
trigger(depthVid);

%Obtenemos los datos adquiridos
[depthFrameData, depthTimeData, depthMetaData]=getdata(depthVid);

%Detenemos los dispositivos
stop(depthVid)

%Cálculo nube de puntos
[X Y Z]=depthtoxyz(depthFrameData);

center_depth=[X(center(2),center(1)), Y(center(2),center(1)),
Z(center(2),center(1))];

a=X';A=a(:)';
b=Y';B=b(:)';
c=Z';C=c(:)';
matriz=[A;B;C];

%En una franja horizontal y vertical en z e i de +-1 con respecto al
centro obtenido, obtenemos los límites de la tabla de calibración en x
y en z respectivamente

%Calculamos las x límite de la tabla de calibración
%Calculamos franja en z de +-0.1 respecto al centro de la diana
z_ordenada=sortrows(matriz',3)';
j=0;
z_diana=[];
for i=1:length(z_ordenada)
    if z_ordenada(3,i)>center_depth(3)-0.01 &&
z_ordenada(3,i)<center_depth(3)+0.01 &&
z_ordenada(1,i)>center_depth(1)
        j=j+1;
        z_diana(:,j)=z_ordenada(:,i);
    end
end

%Eliminamos los puntos que no se encuentren en la misma coordenada y
con un margen de 0.03
z_diana_aux=[];
j=0;
for i=1:length(z_diana)
    if z_diana(2,i)>(center_depth(2)-0.03) &&
z_diana(2,i)<center_depth(2)+0.03
        j=j+1;
        z_diana_aux(:,j)=z_diana(:,i);
```

```
    end
end
x_diana_ordenada=sortrows(z_diana_aux',1)';
for i=2:length(x_diana_ordenada)
    if x_diana_ordenada(1,i)>x_diana_ordenada(1,i-1)+0.04
        limite_x_der=x_diana_ordenada(:,i-1);
        break;
    elseif i==length(x_diana_ordenada)
        limite_x_der=x_diana_ordenada(:,i);
    end
end

%Cálculo limite izquierdo tabla de calibración (mismo proceso)
z_ordenada=sortrows(matriz',3)';
j=0;
z_diana=[];
for i=1:length(z_ordenada)
    if z_ordenada(3,i)>center_depth(3)-0.01 &&
z_ordenada(3,i)<center_depth(3)+0.01 &&
z_ordenada(1,i)<center_depth(1)
        j=j+1;
        z_diana(:,j)=z_ordenada(:,i);
    end
end

z_diana_aux=[];
j=0;
for i=1:length(z_diana)
    if z_diana(2,i)>(center_depth(2)-0.03) &&
z_diana(2,i)<center_depth(2)+0.03
        j=j+1;
        z_diana_aux(:,j)=z_diana(:,i);
    end
end

x_diana_ordenada=sortrows(z_diana_aux',1)';
for i=length(x_diana_ordenada):-1:2
    if x_diana_ordenada(1,i)>x_diana_ordenada(1,i-1)+0.04
        limite_x_izq=x_diana_ordenada(:,i);
        break;
    elseif i==2
        limite_x_izq=x_diana_ordenada(:,i);
    end
end

%Calculamos la z límite de la tabla de calibración (mismo proceso)
x_ordenada=sortrows(matriz',1)';
j=0;
x_diana=[];
for i=1:length(z_ordenada)
    if x_ordenada(1,i)>center_depth(1)-0.01 &&
x_ordenada(1,i)<center_depth(1)+0.01 &&
x_ordenada(3,i)>center_depth(3)
        j=j+1;
        x_diana(:,j)=x_ordenada(:,i);
    end
end

x_diana_aux=[];
j=0;
```

```
for i=1:length(x_diana)
    if x_diana(2,i)>(center_depth(2)-0.03) &&
x_diana(2,i)<center_depth(2)+0.03
        j=j+1;
        x_diana_aux(:,j)=x_diana(:,i);
    end
end

z_diana_ordenada=sortrows(x_diana_aux',3)';
for i=2:length(z_diana_ordenada)
    if z_diana_ordenada(3,i)>z_diana_ordenada(3,i-1)+0.04
        limite_z_sup=z_diana_ordenada(:,i-1);
        break;
    elseif i==length(z_diana_ordenada)
        limite_z_sup=z_diana_ordenada(:,i);
    end
end

if (limite_x_der(1)-limite_x_izq(1))>0.19 && (limite_x_der(1)-
limite_x_izq(1))>0.19 && limite_z_sup(3)-limite_x_der(3)<20 &&
limite_z_sup(3)-limite_x_der(3)>0
    waitfor(msgbox('Calibración automática completada
correctamente'));
else
    waitfor(msgbox('Error de calibración!','Error calibración
automática','error'));
end

lim_tabla=[limite_x_der(1) limite_x_izq(1); limite_x_der(2)
limite_x_izq(2);
    limite_z_sup(3) limite_z_sup(3)-0.2];

 %Delimitamos el lay out del robot
x_min= limite_x_izq(1)-0.3;
x_max= limite_x_der(1)+0.3;
y_min= min(limite_x_izq(2)-0.3,limite_x_der(2)-0.3);
y_max= max(limite_x_izq(2)+0.2,limite_x_der(2)+0.2);
z_min= limite_z_sup(3)-0.2;

close(f)



function figure1_CloseRequestFcn(hObject, eventdata, handles)
global estado_boton_adquisicion;
global depthVid;
if estado_boton_adquisicion==1
    stop(depthVid)
end

delete(hObject);
```

Imagen de profundidad a coordenadas 3D (depthtoxyz.m) Matlab

Esta función permite obtener la conversión de la imagen de profundidad 2D obtenida mediante el sensor Kinect a una nube de puntos en el espacio tridimensional.

```
function [x, y, z] = depthtoxyz(depth)

depth=double(depth);%Para poder realizar operación matricial elemento
a elemento
depth(depth==0)=nan;     %Tomar como not a number para valores nulos
para eliminarlos de la representación

%Datos camara IR
fx=580; %Distancia focal en el eje x en pixeles
fy=580; %Distancia focal en el eje y en pixeles
[v u]=size(depth);    %Numero de pixeles en eje y, eje x
respectivamente
centro=[u/2 v/2];    %Definimos centro en pixeles

matriz_pixeles_x=ones(v,1)*(1:u)-centro(1);    %Definimos valor pixel
matriz x
matriz_pixeles_y=(1:v)'*ones(1,u)-centro(2);  %Definimos valor pixel
matriz y

mm_to_m=10^-3;  %Unidades metros a mm

%Aplicamos ecuaciones de proyecciones de perspectiva
x_kinect=matriz_pixeles_x.*depth/fx*mm_to_m;
y_kinect=matriz_pixeles_y.*depth/fy*mm_to_m;
z_kinect=depth*mm_to_m;

%Cambio coordenadas kinect a reales
x=x_kinect;
y=z_kinect;
z=-y_kinect;
```

Orientación efector final brazo robot (Calcula_T.m)

Esta función permite obtener las matrices de transformación homogénea para cada uno de los puntos seleccionados por el usuario que conforman la trayectoria del robot.

```
function T = Calcula_T(xp,yp,zp)

%Determinación del punto FOCO
foco=[0 0 0.56];

%Cálculo del vector de aproximación
a=[(xp-foco(1)) (yp-foco(2)) (zp-foco(3))];

%Normalización del vector de aproximación
a=a./norm(a);

%Cálculo de un vector perpendicular a z y a  (perpendicular al plano
que
%contiene a ambos)
v_plano=cross(a,[0 0 1]);

%Cálculo del vector n perpendicular al anterior y a "a"
n=cross(a,v_plano);
```

```
%Por último, el vector o es ortonormal con los otros dos
o=cross(a,n);

%Construyo la submatriz de rotación noa
R=[n' o' a'];

%Construyo la matriz de transformación homogenea
rotacion=r2t(R);
T=transl(xp,yp,zp)*rotacion;

end
```

Regeneración de trayectorias (regenerar_trayectoria.m) Matlab

Esta función permite la regeneración de trayectorias en caso de que la presencia de un obstáculo se interponga en la trayectoria normal del robot.

```
global x_lim;
global y_lim;
global z_lim;
global puntos_objetos;
global puntos;
global q;
global trayectorias;
global irb_120;
global traslacion;
global q_aux;
global traslacion_aux;
global mi_robot;

global var_lim_aux;%Utilizaremos esta variable para ejecutar lo
explicado a continuación

global y_lim_aux;
global ejecutando_robot;

%La adquisición de puntos que realiza la Kinect no es uniforme, por lo
que creamos estas variables auxiliares sensiblemente superiores a los
límites obtenidos anteriormente de x_lim, y_lim, z_lim. En caso de que
sean superados se generará una nueva trayectoria, pues en caso
contrario se generarían nuevas trayectorias en cada iteración de la
adquisición.
x_lim_aux=x_lim;
z_lim_aux=z_lim;

if var_lim_aux==1
    disp('sin cambio');

    for i=1:(length(y_lim)/2)
        if y_lim_aux(2*i)>y_lim(2*i)||y_lim_aux((2*i)-1)<y_lim((2*i)-
1)||length(y_lim)~=length(y_lim_aux)
            var_lim_aux=0;
            y_lim_aux=[];%Inicializamos el vector para sobrescribirle
posteriormente
            break;
        end
```

```
    end
end


if var_lim_aux==0
    traslacion_aux=traslacion;
    disp('modificacion');
for i=1:(length(y_lim)/2)
    y_lim_aux(2*i)=y_lim(2*i)-0.03; %Definimos el valor de y_lim_aux
como el de y_lim +-un rango de seguridad
    y_lim_aux((2*i)-1)=y_lim((2*i)-1)+0.03;
end
for i=1:length(z_lim);
    for j=1:length(traslacion)
        if traslacion(3,j)<(z_lim(i)+0.15)%Tomamos el 0.1 como margen

            if traslacion(2,j)>(y_lim(i*2)-
0.15)&&traslacion(2,j)<(y_lim((i*2)-1)+0.15)%y_lim=[y1_max y1_min
y2_max y2_min...];0.1 como margen

                if traslacion(1,j)>(x_lim((i*2)-1)+0.15)%x_lim=[x1_max
x1_min x2_max x2_min...]; tomamos el 0.1 como margen
                    traslacion_aux(3,j)=z_lim(i)+0.15;
                    if j>1
                        if traslacion_aux(3,j-1)<((z_lim(i)-
traslacion_aux)/2)
                            traslacion_aux(3,j-1)=(z_lim(i)-
traslacion_aux)/2;
                            disp('hola1')
                        end
                    end
                    if j<length(traslacion)
                        if traslacion_aux(3,j+1)<((z_lim(i)-
traslacion_aux)/2)
                            traslacion_aux(3,j+1)=(z_lim(i)-
traslacion_aux)/2;
                            disp('hola1')
                        end
                    end
                else
                    traslacion_aux(3,j)=z_lim(i)+0.15;

                    if j>1
                        if traslacion_aux(3,j-1)<z_lim(i)
                            traslacion_aux(3,j-1)=z_lim(i);
                            disp('hola2')
                        end
                    end
                    if j<length(traslacion)
                        if traslacion_aux(3,j+1)<z_lim(i)
                            traslacion_aux(3,j+1)=z_lim(i);
                            disp('hola1')
                        end
                    end
                end

            end
        end
    end
end
```

```
clear T;
for i=1:length(traslacion_aux(1,:))
    T(:,:,i) =
Calcula_T(traslacion_aux(1,i),traslacion_aux(2,i),traslacion_aux(3,i))
;
end

%Aplicamos la cinemática inversa
q_puntos_via=irb120_ikine(T,[0 0 0 0 0 0]);

duracion_segmento=1;
puntos_por_segmento=1;

%Interpolamos esta trayectoria
q_aux=mstraj(q_puntos_via,[],duracion_segmento*ones(1,length(traslacio
n_aux(1,:))),q_puntos_via(1,:),duracion_segmento/puntos_por_segmento,1
);

Tseguida=irb_120.fkine(q_aux);

var_lim_aux=1;

%Ejecutamos RobotStudio
q_deg=q_aux.*180/pi; %Lo pasamos de radianes a grados para que lo
ejecute RobotStudio

%Lo ordenamos en un solo vector
contador=1;
for i=1:(length(q_deg)-1)
    q_deg_vector(contador:(contador+5))=q_deg(i,:);
    contador=(i*6)+1;
    a=i;
end

disp('envio trayectoria');
if ejecutando_robot==1
    mi_robot.JointTrajectory(q_deg_vector);
end

%Dibujamos tras enviar la trayectoria al robot para mayor efectividad
de programa. Dibujamos una sola vez el objeto en caso de que se
produzca una modificación para agilizar la ejecución de programa
        axes(handles.axes1);
        hold off;

representacion=plot3(traslacion(1,:),traslacion(2,:),traslacion(3,:),'
ro', ...
        traslacion(1,:),traslacion(2,:),traslacion(3,:),'b');
        hold on;

plot3(traslacion_aux(1,:),traslacion_aux(2,:),traslacion_aux(3,:),'go'
, ...

traslacion_aux(1,:),traslacion_aux(2,:),traslacion_aux(3,:),'r');

representacion_puntos=plot3(puntos(:,1),puntos(:,2),puntos(:,3),'k*');
```

```
representacion_objetos=plot3(puntos_objetos(1,:),puntos_objetos(2,:),p
untos_objetos(3,:),'b.');%hold on;
        axis equal;
        axis([-0.05 0.8 -0.7 0.7 -0.05 1.2]);
        set(gca,'XLim',[-0.05 0.8]);
        set(gca,'YLim',[-0.7 0.7]);
        set(gca,'ZLim',[-0.05 1.2]);
        box on; grid on;
end
```

Control IRB120 y recepción de datagramas (IRB120_Control) RobotStudio

Este módulo desarrollado en RobotStudio compuesto por diversos procedimientos, permite la conexión con el cliente, así como la recepción y ejecución de trayectorias recibidas mediante datagramas desde Matlab.

```
MODULE IRB120_Control

VAR num      receivedData{1024};

VAR jointtarget axis_pos := [ [0, 0, 0, 0, 0, 0], [ 0, 9E9, 9E9, 9E9,
9E9, 9E9] ];
VAR jointtarget jtar;
VAR orient effector_orient;
VAR robtarget Target_xyz;
VAR robjoint ax;
VAR num axMatrix{6};
VAR num rotMatrix{3};
VAR num posMatrix{3};
VAR num limitMatrix{2};
VAR num velocityMatrix{2};
VAR num n;
VAR bool confFirstError;
VAR num cnfNum;
VAR num variable:=0;


VAR  num contador:=0;
VAR num contador_j:=1;
VAR num contador_j_aux;

    PROC DataProcessingAndMovement()
        WHILE connectionStatus DO
            IF variable = 3 THEN
                stringHandler(receivedData);
            ELSE
            SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= WAIT_MAX;   !Receive data from the socket
            stringHandler(receivedData);  !Handle the received data
string
            ENDIF
        ENDWHILE
      ERROR
        IF ERRNO=ERR_SOCK_CLOSED THEN
            connectionStatus := FALSE;
```

```
            SocketClose socketServer;
            WaitTime 2;
            SocketClose socketClient;
            WaitTime 2;
            main;
        ENDIF
    ENDPROC
    PROC stringHandler (num Data2Process{*})

        n := 0;
        Target_xyz := CRobT(\Tool:=tool \WObj:=wobj0);
        TEST Data2Process{1}
      CASE 6:

        IF contador_j<>1 THEN
        contador:=0;
        contador_j_aux:=contador_j;
        FOR j FROM contador_j_aux TO Data2Process{2} DO
            contador_j:=j;
            FOR i FROM 3+contador+((contador_j_aux-1)*12) TO
13+contador+((contador_j_aux-1)*12) STEP 2 DO
                Incr n;
                IF Data2Process{i} = 0 THEN
                    axMatrix{n} := -Data2Process{i+1};
                ELSEIF Data2Process{i} = 1 THEN
                    axMatrix{n} := Data2Process{i+1};
                ENDIF
            ENDFOR

            ax.rax_1 := axMatrix{1};
            ax.rax_2 := axMatrix{2};
            ax.rax_3 := axMatrix{3};
            ax.rax_4 := axMatrix{4};
            ax.rax_5 := axMatrix{5};
            ax.rax_6 := axMatrix{6};
            axis_pos := [[ax.rax_1, ax.rax_2, ax.rax_3, ax.rax_4,
ax.rax_5, ax.rax_6] , [ 0, 9E9, 9E9, 9E9, 9E9, 9E9]];

            !MoveAbsJ axis_pos, vel, z30, tool;

            !====== Security procedure (reachability) ======
            CheckAxesLimitations axMatrix;
            !================================================

            IF statusAVAILABLE THEN
                ConfJ \Off;
                MoveAbsJ axis_pos, vel, z30, tool;

            ELSEIF statusAVAILABLE = FALSE THEN
                SendDataProcedure 2;
            ENDIF

            contador:=contador+12;
            n:=0;

            variable:=1;
            SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= 0.001;

            IF variable = 1 THEN
                variable :=3;
```

```
                RETURN;
            ENDIF
        ENDFOR
        ENDIF

        contador:=0;
        FOR j FROM 1 TO Data2Process{2} DO
            contador_j:=j;
            FOR i FROM 3+contador TO 13+contador STEP 2 DO
                Incr n;
                IF Data2Process{i} = 0 THEN
                    axMatrix{n} := -Data2Process{i+1};
                ELSEIF Data2Process{i} = 1 THEN
                    axMatrix{n} := Data2Process{i+1};
                ENDIF
            ENDFOR

            ax.rax_1 := axMatrix{1};
            ax.rax_2 := axMatrix{2};
            ax.rax_3 := axMatrix{3};
            ax.rax_4 := axMatrix{4};
            ax.rax_5 := axMatrix{5};
            ax.rax_6 := axMatrix{6};
            axis_pos := [[ax.rax_1, ax.rax_2, ax.rax_3, ax.rax_4,
ax.rax_5, ax.rax_6] , [ 0, 9E9, 9E9, 9E9, 9E9, 9E9]];

            !MoveAbsJ axis_pos, vel, z30, tool;

            !====== Security procedure (reachability) ======
            CheckAxesLimitations axMatrix;
            !================================================

            IF statusAVAILABLE THEN
                ConfJ \Off;
                MoveAbsJ axis_pos, vel, z30, tool
            ELSEIF statusAVAILABLE = FALSE THEN
                SendDataProcedure 2;
            ENDIF

            contador:=contador+12;
            n:=0;

            variable:=1;
            SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= 0.001;

            IF variable = 1 THEN
                variable :=3;
                RETURN;
            ENDIF
        ENDFOR
        variable:=3;

        CASE 7:
        StopMove;
        SocketReceive socketClient \Data:=receivedData
\ReadNoOfBytes:=1024 \Time:= WAIT_MAX;   !Receive data from the socket
        StartMove;

   ENDTEST
```

```
        ERROR
            IF ERRNO=ERR_SOCK_TIMEOUT THEN
                variable:=0;
                TRYNEXT;!Ejecutamos siguiente instrucción
            ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
                variable:=0;
                TRYNEXT;!Ejecutamos siguiente instrucción
             ELSE
               !No error recovery handling
            ENDIF
```

Presupuesto

Este capítulo proporciona información detallada acerca de los costes teóricos del desarrollo del proyecto, incluyendo los costes de materiales y las tasas profesionales.

PR.1 Costes materiales

Objeto		***Cantidad***	***Precio unidad (€)***	***Precio total (€)***
Hardware	Portátil Samsung	1	650	650
	ABB-IRB120	1	10954,00	10954,00
	Sensor Kinect	1	130	130
Software	Microsoft Windows 7	1	0,00	0,00
	RobotStudio	1	0,00	0,00
	Matlab R2014a	1	0,00	0,00
	Microsoft Office Home and Student 2010	1	100,00	100,00
TOTAL				11834,00

Tabla PR.1: Costes materiales (hardware y software) sin IVA.

PR.2 Costes totales

Los costes totales del Proyecto han sido obtenidos sumando los costes materiales y profesionales aplicando el IVA. Además hay que tener en cuenta los costes de impresión y encuadernación.

Objeto		***Costes totales***
Costes materiales		11734,00 €
Costes profesionales		6000,00 €
Proyecto	Impresión	60 €
	Encuadernación	40 €
Subtotal		17934 €
IVA (21%)		3766 €
TOTAL		21700 €

Tabla PR.2: Costes totales con IVA.

Bibliografía

[1] Trabajo Fin de Grado: "Socket based communication in RobotStudio for controlling ABB-IRB120 robot. Design and development of a palletizing station", Marek Jerzy Frydrysiak

[2] Trabajo Fin de Grado: "Desarrollo de una interfaz para el control del robot IRB120 desde Matlab", Azahara Gutiérrez Corbacho

[3] Fundamentos de Robótica, Antonio Barrientos; McGraw Hill

[4] Manual de referencia técnica. Instrucciones, funciones y tipos de datos de RAPID

[5] Robotics, Vision and Control, Peter Corke

[6] Robotics Toolbox. Peter Corke. User's Guide

[7] Image Processing Toolbox.User´s Guide

[8] Image Acqusition Toolbox.User´s Guide

[9] Spline Toolbox.User´s Guide

[10] http://www.mathworks.es/

[11] https://msdn.microsoft.com/en-us/library/hh855355.aspx

[12] http://new.abb.com/products/robotics/es/robots-industriales/irb-120