Universidad Zaragoza
1542

Trabajo Fin de Grado en Ingeniería Informática

Desarrollo de un demostrador práctico en ROS para el robot manipulador UR10

Autor

IGNACIO HERRERA SEARA

Director

GONZALO LÓPEZ NICOLÁS

Codirector

IGNACIO CUIRAL ZUECO

Escuela de Ingeniería y Arquitectura

2021

Desarrollo de un demostrador práctico en ROS para el robot manipulador UR10

RESUMEN

La robótica es una rama multidisciplinar muy importante hoy en día, sobre todo en aplicaciones industriales, y que cobra cada vez más relevancia. Dentro de la robótica nos encontramos con los robots manipuladores pensados para sustituir a los humanos en operaciones que requerían gran esfuerzo, en entornos peligrosos o en la realización de tareas muy repetitivas, entre otras cosas. Con la evolución de la industria surgieron necesidades que requerían la colaboración estrecha entre estos robots y los humanos, y es aquí donde aparecen los robots colaborativos. Además, en los últimos años ha crecido el uso de software robótico colaborativo, dentro de este campo nos encontramos con el entorno ROS, que cuenta con una comunidad de desarrolladores muy activa.

En este trabajo se ha utilizado el robot colaborativo UR10 junto con el entorno ROS para el desarrollo de diversas aplicaciones prácticas, que han sido probadas tanto en simulación como en un entorno real. El control de los movimientos del robot se ha desarrollado a través del *framework* MoveIt. Las aplicaciones implementadas han sido las siguientes: realización de dibujos en un lienzo por el robot, extracción y dibujo de contornos de objetos reales y, por último, teleoperación del robot. Además, las dos últimas aplicaciones hacen uso de la cámara RGB-D Realsense D435 para su funcionamiento.

Este proyecto ha sido el primero en utilizar el robot UR10 en el entorno ROS en el Departamento de Informática e Ingeniería de Sistemas. Por lo tanto, en la memoria se recoge toda la información necesaria para poder configurar y ejecutar los programas para controlar el robot, tanto en simulación como en un entorno real. Además, se han agregado diversos tutoriales y fuentes de información útiles sobre el robot y las diversas herramientas utilizadas. Todo esto con el objetivo de que sirva de punto de partida y de referencia para futuros trabajos que hagan uso de estas herramientas.

Índice

Capítulo 1

Introducción y objetivos

1.1. Introducción

La robótica es una rama multidisciplinar muy importante hoy en día, sobre todo en aplicaciones industriales, y que va cobrando más relevancia, puesto que cada vez se necesita que más procesos sean robotizdos y dichos procesos son cada vez más complejos. Dentro de la robótica nos encontramos con los robots manipuladores, que fueron pensados para sustituir a los humanos en operaciones que requerían gran esfuerzo, en entornos peligrosos o en la realización de tareas muy repetitivas, entre otras cosas.

Los robots manipuladores no fueron concebidos para trabajar compartiendo el espacio de trabajo con los humanos debido a que poseen una gran fuerza y velocidad de movimiento y, por lo tanto, pueden poner en peligro la integridad física de los humanos que se encuentren dentro de su espacio de trabajo. Con el paso del tiempo surgieron necesidades que requerían la colaboración estrecha entre los humanos y los robots manipuladores, y es aquí donde surgieron los llamados robots colaborativos o cobots, diseñados especialmente para trabajar con los humanos. Los cobots surgieron hace poco más de una década y es uno de los segmentos de la robótica que mas crece anualmente, no solo a nivel comercial sino también en innovación.

En este trabajo se va a utilizar el robot colaborativo UR10 [1], de la empresa danesa Universal Robots, junto al entorno ROS (Robot Operating System) [2] con el objetivo de desarrollar los siguientes demostradores/aplicaciones:

1. Pizarra Virtual: el usuario dibuja sobre una pizarra en el ordenador y el robot replica el dibujo en papel.

2. Dibujo Contornos: mediante el uso de una cámara y de marcas ArUco [3] se reconocen los objetos que hay sobre una mesa de trabajo y el robot debe dibujar sus contornos.

3. Teleoperación: uso de una cámara y de una marca ArUco para la teleoperación del robot.

Este trabajo se enmarca dentro de las actividades del proyecto COMMANDIA [4], un proyecto de investigación en Robótica móvil colaborativa de objetos deformables cofinanciado por el Programa Interreg Sudoe y por el Fondo Europeo de Desarrollo Regional (FEDER).

1.2. Objetivos

El objetivo de este trabajo no es solo el desarrollo de las tres aplicaciones nombradas anteriormente, sino también el aprendizaje del entorno ROS junto con el *framework* MoveIt para el desarrollo de aplicaciones de robots manipuladores. Es la primera vez que se pone en funcionamiento un robot UR10 junto con ROS en el Departamento de Informática e Ingeniería de Sistemas. Este trabajo sirve, por lo tanto, de punto de partida y de referencia para futuros trabajos que hagan uso de estas herramientas.

A continuación se enumeran las distintas tareas seguidas para alcanzar los objetivos propuestos:

1. Instalación del sistema operativo Ubuntu 16.04 LTS junto al entorno ROS y el *framework* MoveIt.

2. Aprendizaje del entorno ROS, sus conceptos teóricos y sus distintas herramientas, siguiendo la documentación oficial.

3. Estudio del *framework* MoveIt, los elementos que lo componen y sus distintas funcionalidades.

4. Configuración del entorno de trabajo en ROS: definición del modelo del robot, creación de la escena y generación del nuevo paquete de MoveIt.

5. Configuración del robot UR10 real para que pueda ser utilizado en ROS.

6. Desarrollo de la aplicación 'Pizarra Virtual' y realización de pruebas en un entorno real.

7. Estudio de la utilización de la cámara Intel® RealSense™ Depth Camera D435 junto a las marcas ArUco en ROS para el procesamiento de imágenes.

8. Desarrollo de la aplicación 'Dibujo Contornos' y realización de pruebas para comprobar su correcto funcionamiento.

9. Desarrollo de la aplicación 'Teleoperación' y ejecución en el robot real.

1.3. Entorno de trabajo

Se han utilizado las siguientes herramientas de *hardware* y de *software*:

- **Robot manipulador colaborativo:** Universal Robots UR10 [1].
- **Cámara RGB-D:** Intel® RealSense™ Depth Camera D435 [5].
- **Software principal:**
  - ROS (Robot Operating System [2]), en su versión *Kinetic*.
  - MoveIt [6]: en su versión para ROS *Kinetic*.
  - Librería de visión por computador OpenCV [7] (versión 2.4.9.1).
  - Adaptación para ROS de la librería Intel® RealSense™ SDK 2.0 (System Development Kit) [8].
  - Adaptación para ROS de la librería ArUco de la empresa PAL Robotics [9].
  - Python 2.7 y C++ 11.

1.4. Estructura de la memoria

Esta memoria se estructura en 9 capítulos cuyo contenido es:

- **Capítulo 1:** Introducción y contexto del presente trabajo, planteamiento del problema y objetivos establecidos para su resolución.
- **Capítulo 2:** Introducción al entorno ROS y MoveIt y, explicación del concepto de robot manipulador colaborativo.
- **Capítulo 3:** Configuración del modelo del robot en ROS, creación del paquete de MoveIt y definición del entorno de trabajo del robot.
- **Capítulo 4:** Explicación de los pasos necesarios para ejecutar ROS y MoveIt en el robot real.
- **Capítulo 5:** Explicación de la metodología seguida para calibrar la posición de la punta del rotulador a utilizar con respecto al robot.
- **Capítulo 6:** Explicación en alto nivel de la solución implementada para las diferentes aplicaciones.
- **Capítulo 7:** explicación y desarrollo de la primera aplicación a implementar, Pizarra Virtual, así como los resultados obtenidos.

- **Capítulo 8:** descripción del demostrador Dibujo Contornos y explicación de su desarrollo y de los resultados conseguidos.
- **Capítulo 9:** explicación de la tercera, y última, aplicación a desarrollar, Teleoperación. Además de mostrar los resultados obtenidos durante su ejecución.

Se incluyen diversos anexos que complementan a los capítulos con información considerada relevante. Además, hay disponible una lista de reproducción [10] con diversos vídeos en los que se muestran los resultados obtenidos en las pruebas realizadas con los demostradores.

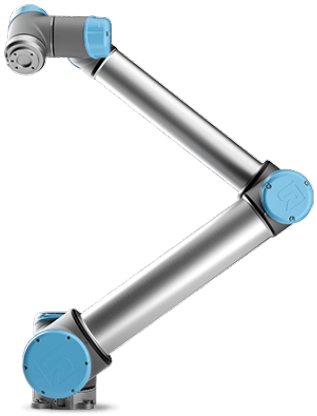
Capítulo 2

Iniciación

En este capítulo se exponen los principales conceptos de los robots colaborativos, y en concreto del UR10, además del entorno ROS y del *framework* MoveIt.

2.1. Robot colaborativo UR10

El robot colaborativo UR10 utilizado en este trabajo se encuentra sobre la superficie del robot móvil Campero (figura 2.1), que es el prototipo no comercial que posteriormente se ha comercializado como el modelo RB-EKEN 10 de la empresa Robotnik. El robot Campero se adquirió expresamente para su uso en el proyecto COMMANDIA [4].



(a) Robot UR10.

(b) Robot Campero.

Figura 2.1: Robots UR10 y Campero.

Los robots colaborativos ofrecen ciertas ventajas sobre los robots manipuladores tradicionales. La más importante es que pueden compartir espacio de trabajo con los humanos, por lo que no es necesario instalar pantallas protectoras. Otra característica importante es que su base de sujeción (*footprint*) es más reducida, lo cual facilita su instalación en espacios de trabajo reducidos. Otra ventaja respecto a los industriales es que su precio es mucho menor, por lo que están al alcance de pequeñas y medianas empresas. Sin embargo, para reducir

el riesgo de provocar daños a los trabajadores, su velocidad de movimiento suele ser más reducida y su capacidad de carga es menor.

Según las especificaciones de la propia empresa [1] cuenta con 15 funcionalidades de seguridad avanzada con certificación TÜV NORD que han sido probadas de acuerdo a las normas *EN ISO 13849:2008 PL.d*, lo que implica que su probabilidad de fallos por hora es del 0.00001 % a 0.0001 % [11]. Entre las funcionalidades de seguridad destaca el frenado de emergencia ante colisiones así como la reducción de velocidades ante la aproximación de operarios u objetos del entorno, mediante el uso de sensores especiales.

A continuación se introduce la clasificación en dos grupos de los elementos que conforman un brazo robótico

- Manipulador (*manipulator*): es el componente principal y está formado por una serie de eslabones (*links*), elementos estructurales sólidos, unidos mediante articulaciones (*joints*) que permiten el movimiento relativo entre dos eslabones consecutivos. Cada *joint* otorga al robot un grado de libertad.

- Actuador final (*end-effector*): el elemento que permite realizar la acción correspondiente a la tarea que se desea cumplimentar con el robot. Por ejemplo, una pinza para el agarre y transporte, un soldador para la unión de piezas, etc.

El robot UR10 del trabajo, figura 2.2, consta de 6 grados de libertad, además de un sensor de torque y presión (Robotiq FT-300S) y una pinza (Robotiq 2F-85) como *end-effector* que, aunque no se van a utilizar, no se van retirar. Se ha acoplado un soporte para un rotulador al *end-effector* paralelo al sensor y la pinza.

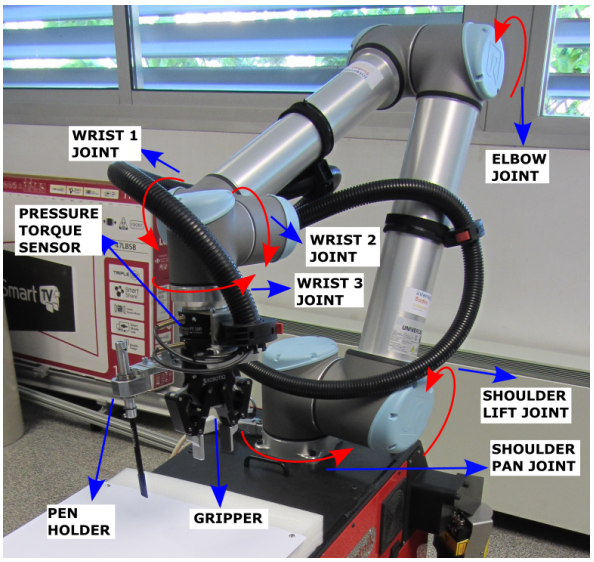


Figura 2.2: Robot UR10 utilizado en el trabajo. Las articulaciones están señaladas, con una flecha roja, junto con su dirección de giro. También se marca la pinza, el soporte del rotulador y el sensor de torque y presión.

2.2. Iniciación a ROS

Para la iniciación a ROS (Robot Operating System) se ha utilizado principalmente el libro 'ROS Robot Programming' [12], que explica todos los conceptos de manera clara y sencilla. Además, se han seguido los ejemplos que aparecen en la página web oficial de ROS [13].

ROS es un "meta-sistema operativo" orientado a robots, es decir, no es un sistema operativo tradicional como Windows o Android, sino que corre sobre un sistema operativo. La diferencia está en que ROS se nutre de las funcionalidades de los sistemas tradicionales (como pueden ser el manejo de procesos, sistemas de ficheros, compiladores, interfaces de usuario...) para funcionar, por tanto, es un sistema de soporte para controlar robots y sensores con abstracción de hardware y desarrollo de aplicaciones robóticas basadas en sistemas operativos convencionales. Entre sus funcionalidades se encuentran: control de errores, paso de mensajes entre procesos, transmisión/recepción de datos entre hardware heterogéneo, control de paquetes... además de ofrecer librerías y herramientas para facilitar el desarrollo de código.

Un paquete es la unidad básica de ROS, las aplicaciones se desarrollan en base a éstos, y contienen toda la información necesaria para ejecutar otros paquetes o nodos. Los paquetes pueden estar formados por nodos, librerías, archivos de configuración, otros paquetes, mensajes, *topics*, etc. El objetivo es que cada paquete esté destinado a una funcionalidad concreta, por ejemplo, mover las articulaciones de un robot manipulador de manera independiente. Los paquetes hacen las veces de módulos que pueden ser reutilizados por otros desarrolladores, ahorrando así tiempo de desarrollo y generando una comunidad colaborativa donde además se invita a publicar el código de manera libre (como el propio ROS).

A continuación se explican los conceptos más importantes para el desarrollo de aplicaciones en ROS:

- **Nodos**: es la unidad mínima de procesamiento de ROS, similar a un programa ejecutable. Se recomienda crear un nodo para cada propósito y que pueda ser reutilizable de manera sencilla. Hay dos formas de ejecutar un nodo mediante comandos:
    - *rosrun*: comando básico de ejecución de ROS, lanza únicamente un nodo del paquete al que pertenece.
    - *roslaunch*: usa archivos '.launch' con sintaxis basada en XML. Permiten lanzar múltiples nodos (no tienen porque ser del mismo paquete) y tiene funcionalidades extra como cambiar parámetros de los nodos, variables de entorno, etc.
- ***Master***: actúa como un servidor de nombres para las conexiones nodo-a-nodo y el intercambio de mensajes, sin el *master* la comunicación sería imposible. Los nodos solo

utilizan el servidor para registrarse o darse de baja y solicitar información sobre otros nodos. Una vez localizado el nodo con el que quieren comunicarse se establece una conexión *peer-to-peer* entre ellos. Utilizando el comando *roscore* se lanza el *master*.

- **Mensajes**: un mensaje contiene la información que se envían los nodos entre sí. Pueden contener tipos básicos como pueden ser enteros, decimales, booleanos o cadenas de texto, pero también se permiten mensajes anidados y vectores de mensajes.

- ***Topics***: es un tipo de comunicación unidireccional asíncrona entre nodos. Cada *topic* solo puede contener un tipo de mensaje. Un nodo registra un *topic* en el *master* y, después, cualquier nodo puede publicar mensajes en dicho *topic* o bien suscribirse al *topic* para recibir mensajes.

- ***Services***: tipo de comunicación que se utiliza para intercambiar mensajes de manera síncrona y bidireccional entre un nodo cliente, que solicita un servicio respecto a una tarea en particular, y un nodo servidor, que responde a las solicitudes. El nodo cliente espera a recibir la respuesta y, una vez recibida la conexión entre ambos se cierra, es decir, es una comunicación "de una sola vez".

- ***Actions***: método de comunicación bidireccional asíncrono, parecido a los *services*, que se utiliza en aquellas situaciones donde el tiempo de respuesta es alto y se necesitan respuestas intermedias (*feedback*) hasta que el servidor devuelva el resultado final.

- ***Parameters***: cada nodo puede tener una serie de parámetros que se registran en el servidor de parámetros almacenado en el *master* y que pueden ser leídos y modificados por otros nodos. Esto es útil, por ejemplo, para cambiar la velocidad máxima y mínima de un robot móvil en cualquier momento.

2.3. URDF

Para trabajar con ROS es necesario describir los robots y sus distintos componentes, para ello se utiliza un formato de fichero XML llamado URDF (*Unified Robot Description Format*) [14], que permite describir completamente el modelo de un robot indicando sus partes, articulaciones, sensores, propiedades físicas, etc. También existen los ficheros Xacro (XML Macros) [15], en los que se utiliza un lenguaje de macro que nos permite definir archivos URDF de una manera más sencilla y reutilizable, permitiendo crear expresiones matemáticas, definir propiedades que se repiten a lo largo de un fichero, parametrizar el archivo, importar modelos de otros archivos, etc. Para más información sobre la sintaxis o ejemplos se pueden visitar las páginas web de URDF [16] y Xacro [17] en la wiki de ROS.

2.4. Iniciación a MoveIt

MoveIt es un *framework* de ROS para el desarrollo de aplicaciones orientadas a robots manipuladores. Este *framework* permite la planificación y ejecución de trayectorias, manipulación, percepción 3D, detección de colisiones, etc. La instalación se realiza de manera muy sencilla en forma de paquete para ROS, como se explica en su página web [18]. Para el aprendizaje de MoveIt se han seguido los tutoriales oficiales de la versión Kinetic [19], además, se puede encontrar una explicación de sus principales conceptos en el siguiente enlace [20].

Moveit es un *framework* complejo que abarca una gran variedad de conceptos, entre ellos, los más relevantes para este trabajo son:

- **move_group:** se trata del nodo principal de MoveIt. Es el encargado de recibir las operaciones de planificación de trayectorias, ejecución de movimientos, configuración de parámetros, comunicación con el robot, etc.

- **Interfaz de usuario:** el usuario puede acceder a las funcionalidades del nodo *move_group* a través de diferentes interfaces. En este trabajo se van a utilizar la interfaz de C++ y el plugin de Rviz.

- **Planificación de movimientos:** en MoveIt se pueden usar distintos tipos de planificadores de movimientos, por ejemplo, CHOMP y OMPL. Se pueden utilizar dos tipos de planificación en el *end-effector*:

  - *Posición objetivo*: el usuario define una posición y orientación objetivo para el *end-effector*, a partir de los cuales el planificador devuelve una trayectoria.

  - *Camino cartesiano*: el usuario define una serie de puntos de guiado, *waypoints*, por donde debe pasar el *end-effector*. El planificador devolverá la trayectoria que pase por el mayor número de puntos definidos.

- **Restricciones:** se pueden añadir restricciones cinemáticas a los movimientos del robot, estas restricciones las tiene en cuenta el planificador de movimientos a la hora de buscar una solución. Las restricciones con las que cuenta MoveIt son: restricción de orientación y posición de *links*, restricciones de visibilidad (requiere del uso de sensores) y restricciones de *joints*, que permiten establecer un valor máximo y mínimo para cada *joint*. Además, los usuarios pueden definir sus propias restricciones.

2.5. Rviz

Rviz (ROS visualization) es una herramienta de visualización de datos 3D para ROS. Es una herramienta muy útil puesto que nos permite ver los movimientos de los robots, su espacio de trabajo, la interacción de estos con el entorno, los datos que se reciben de los sensores, imágenes en tiempo real de una cámara, etc. Los desarrolladores además pueden crear menús específicos dentro de Rviz para sus paquetes, como es el caso de MoveIt: tiene una interfaz muy simple, figura 2.3, desde donde se pueden planificar y ejecutar movimientos, cargar escenas, desactivar objetos del entorno, activar/desactivar articulaciones, etc.

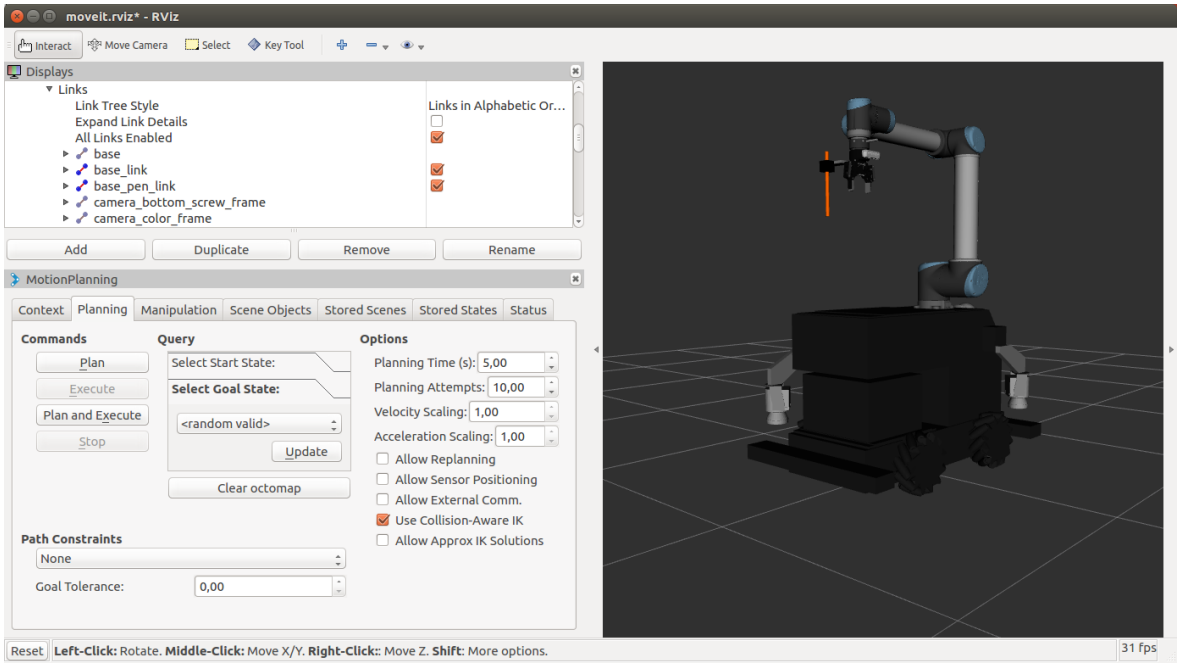


Figura 2.3: GUI del menu de MoveIt en Rviz.

En este trabajo se utilizará Rviz para la visualización de los movimientos del robot UR10 tanto en simulación como en el robot real, la visualización del entorno y de las imágenes de la cámara en tiempo real. Añadir que, como en este trabajo no se requieren simulaciones físicas de las dinámicas del sistema ni simulación de sus sensores, no ha sido necesario hacer uso de simuladores, como Gazebo.

Capítulo 3

Configuración del proyecto en ROS

Antes de empezar con el desarrollo de los demostradores es necesario realizar una serie de tareas previas que nos permitan utilizar el robot con ROS y MoveIt:

1. Creación del modelo del robot Campero en URDF (apartado 3.2).
2. Creación del paquete de MoveIt para el robot UR10 (apartado 3.3).
3. Definición del entorno de trabajo del robot (apartado 3.4).

3.1. Estructura del proyecto

Para proporcionar una visión general de la estructura del proyecto se muestra en la figura 3.1 un diagrama de los módulos que lo componen y sus correspondientes paquetes de ROS asociados.

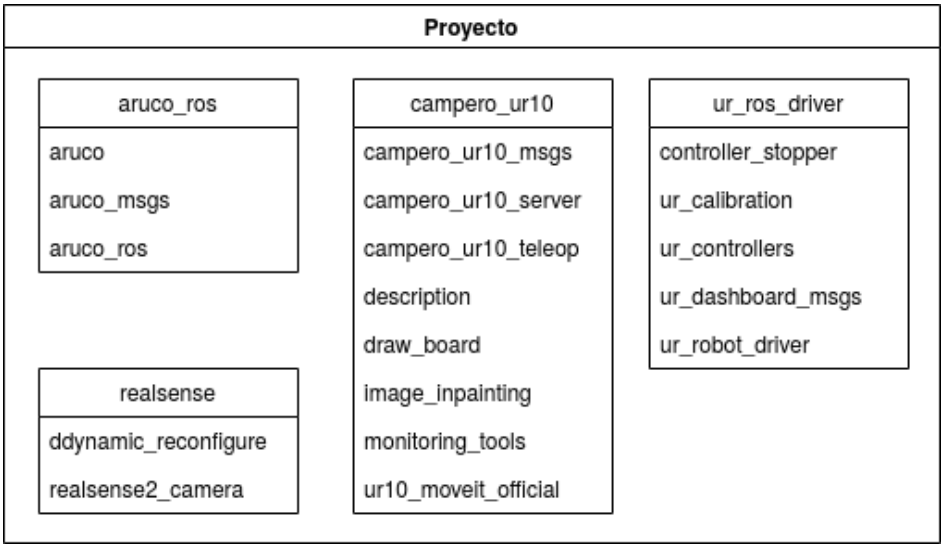


Figura 3.1: Estructura del Proyecto Desarrollado. Se compone de 4 módulos, cada uno de los cuales está formado por una serie de paquetes de ROS.

El proyecto se divide en 4 módulos principales, cada uno con un propósito distinto:

- **aruco_ros:** wrapper de la librería ArUco para ROS de la empresa PAL Robotics [9], su objetivo es detectar marcas ArUco y publicar su información, su id y la posición y orientación de las mismas respecto de la cámara. Las marcas ArUco, desarrolladas en la Universidad de Córdoba [3], son unos marcadores visuales de forma cuadrada con patrones en blanco y negro fácilmente identificables en imágenes con técnicas de visión por computador. Al paquete *aruco_ros* se le ha añadido un nuevo nodo que permite enviar la imagen obtenida de la cámara donde se encuentran las marcas reconocidas y la posición de las esquinas de cada marca en dicha imagen, además de la información nombrada anteriormente. En el Anexo D se puede encontrar una explicación más detallada del nodo.

- **realsense:** Adaptación para ROS de la librería Intel® RealSense™ SDK 2.0 [8]. Nos permite utilizar la cámara RealSense™ Depth Camera D435.

- **ur_ros_driver:** módulo de Universal Robots que contiene los paquetes necesarios para utilizar el hardware de sus robots con ROS [21].

- **campero_ur10:** módulo principal del proyecto que contiene los siguientes paquetes:

    - *campero_ur10_msgs*: contiene los mensajes que se intercambian los nodos para el funcionamiento de los demostradores implementados.
    - *campero_ur10_server*: contiene el nodo que recibe las operaciones de los demostradores y se comunica con MoveIt.
    - *campero_ur10_teleop*: paquete del demostrador "Teleoperación".
    - *description*: contiene los paquetes necesarios para crear el modelo del robot, la escena donde trabaja el robot y distintos archivos de configuración para que funcione sobre el robot real.
    - *image_inpainting*: paquete del demostrador "Dibujo Contornos".
    - *monitoring_tools*: distintas herramientas desarrolladas para testear.
    - *ur10_moveit_official*: paquete de MoveIt del robot UR10 a utilizar.

3.2. Descripción del robot

En la figura 3.2 se encuentran marcados los distintos componentes que hay que añadir al archivo de descripción URDF del robot Campero. También se ve la mesa de trabajo, de la cual se hablará en el apartado 3.4, y el lienzo sobre el que pinta el robot.

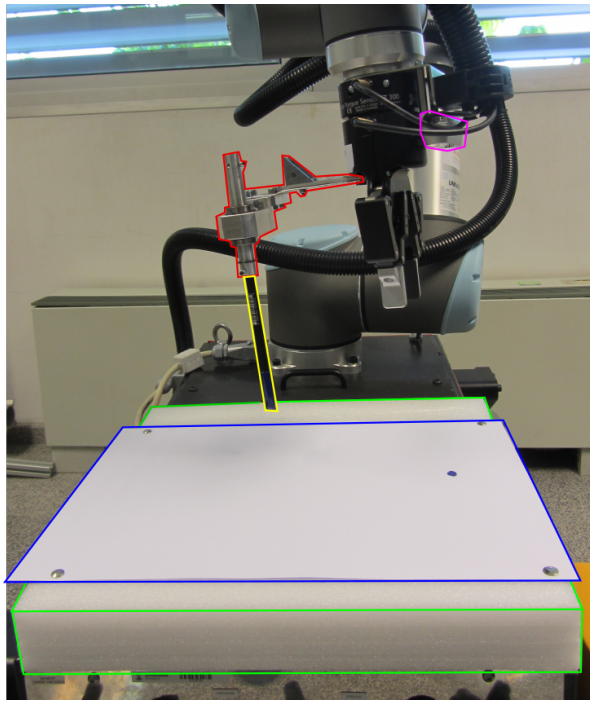


Figura 3.2: Los componentes a añadir al archivo de descripción del robot Campero son: el rotulador, marcado en amarillo; el soporte del rotulador, marcado en rojo; y la cámara, marcada en violeta. También se ve marcado en azul el lienzo y en verde la mesa de trabajo, la cual hay que añadir al entorno de trabajo.

Partiendo del archivo de descripción del robot Campero que venía en el paquete proporcionado por Robotnik, *campero_mecanum.urdf.xacro*, se procedió a crear uno nuevo llamado *campero_ur10.urdf.xacro* (localizado en *campero_ur10_description/robots*). Este archivo difiere con respecto al de Robotnik en la manera de instanciar al robot UR10. En este nuevo archivo el modelo del robot UR10 recibe como argumento un fichero de calibración (explicación en el apartado 4.2.1 paso 3), se ha eliminado el uso de un prefijo para los nombres de las articulaciones del robot y se han definido límites articulares para cada una de las articulaciones. En la figura 3.3 se ve la instanciación del robot UR10 en el archivo URDF.

```
<!-- UR-10 ARM -->
<xacro:arg name="kinematics_config" default="$(find campero_ur10_description)/config/campero_ur10_calibration.yaml"/>
<xacro:ur10_robot prefix="" joint_limited="true"
    shoulder_pan_lower_limit="${-PI}" shoulder_pan_upper_limit="${PI}"
    shoulder_lift_lower_limit="${-PI}" shoulder_lift_upper_limit="${PI}"
    elbow_joint_lower_limit="${-PI}" elbow_joint_upper_limit="${PI}"
    wrist_1_lower_limit="${-PI}" wrist_1_upper_limit="${PI}"
    wrist_2_lower_limit="${-PI}" wrist_2_upper_limit="${PI}"
    wrist_3_lower_limit="${-PI}" wrist_3_upper_limit="${PI}"
    kinematics_file="${load_yaml('$(arg kinematics_config)')}">
</xacro:ur10_robot>
```

Figura 3.3: Instanciación del modelo URDF del UR10 en el archivo de descripción del robot Campero, junto con el parámetro de calibración (*kinematics_config*) y los límites articulares, uno inferior y otro superior por cada articulación.

Otra diferencia reside en la incorporación de la cámara RealSense™ Depth Camera D435 en el modelo, puesto que en el archivo original no estaba definida. Para ello se importa el archivo URDF de la cámara del paquete *realsense2_camera* y se configura como elemento solidario al *end-effector*, como se ve en la figura 3.4. También, hay que especificar una posición relativa de la cámara respecto al *end-effector*. Esto se realizó con mediciones manuales.

```
<!-- Real sense camara D435 -->
<xacro:sensor_d435 parent="ee_link">
    <origin xyz="0.055 0 -0.045" rpy="${PI} 0 0"/>
</xacro:sensor_d435>
```

Figura 3.4: Instanciación del modelo URDF de la cámara D435 en el archivo de descripción del Campero, donde el *ee_link* hace referencia al *end-effector*. La posición origen de la cámara se define con respecto a dicha referencia.

Por último es necesario añadir el soporte del rotulador sujeto al brazo robótico (figura 3.5). Al no contar con un modelo 3D ya creado se procedió a crear un nuevo *link*, llamado *base_pen_link*, compuesto por tres objetos: dos cubos y un cilindro, este último representa el rotulador y la parte del soporte por donde se introduce. Los tamaños definidos para estos objetos son una aproximación pesimista del volumen real del objeto, puesto que dicho volumen será el que se tenga en cuenta para evitar colisiones durante la planificación de movimientos. También es necesario crear una articulación, *base_pen_joint*, que una el nuevo *link* con el *end-effector*. El procedimiento seguido para calcular la posición relativa del soporte se puede encontrar en el Capítulo 5.

```
<!--Soporte rotulador-->
<joint name="base_pen_joint" type="fixed">
    <parent link ="ee_link" />
    <child link = "base_pen_link" />
    <origin xyz="0.264 0.0435 0.129" rpy="0.0 0.0 0.0" />
</joint>

<link name = "base_pen_link" type="fixed">
    <visual> <!--cilindro soporte-->
        <origin xyz="-0.11 0 0" rpy="0 ${PI/2} 0" />
        <geometry>
            <cylinder length="0.24" radius="0.0075"/>
        </geometry>
        <material name="yellow">
            <color rgba="1 0.4 0 1"/>
        </material>
    </visual>
    <visual> <!--box-->
        <origin xyz="-0.175 0 0" rpy="0 0 0" />
        <geometry>
            <box size="0.04 0.04 0.04 "/>
        </geometry>
        <material name="black">
            <color rgba="0 0 0 1"/>
        </material>
    </visual>
    <visual> <!--conector wrist3-->
        <origin xyz="-0.19 -0.04 -0.035" rpy="0 0 0" />
        <geometry>
            <box size="0.01 0.045 0.11"/>
        </geometry>
        <material name="grey">
            <color rgba="0.5 0.5 0.5 1"/>
        </material>
    </visual>
</link>
```

Figura 3.5: Soporte Rotulador en el archivo de descripción del Campero.

Se muestra el resultado final del modelo del soporte del rotulador en la figura 3.6 y el modelo completo del robot Campero en la figura 3.7.

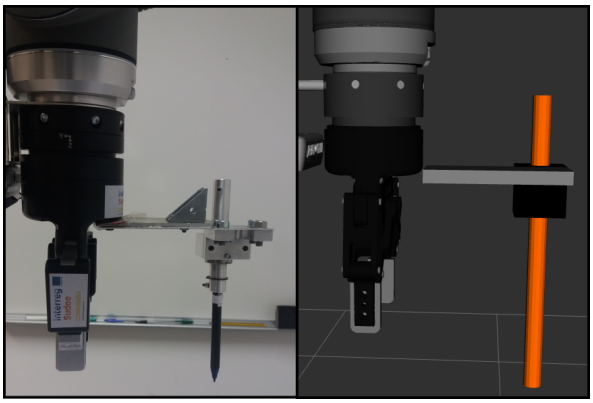


Figura 3.6: Soporte del rotulador montado en el robot real, imagen de la izquierda, y el modelo creado en URDF visto desde Rviz, imagen derecha.

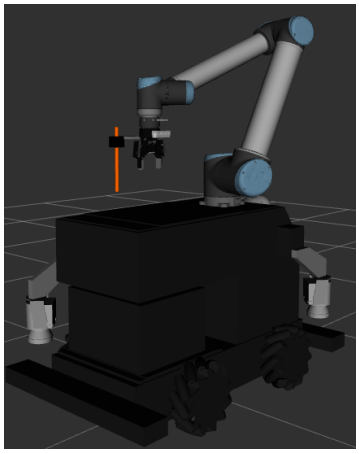


Figura 3.7: Resultado modelo URDF del robot Campero visto en Rviz.

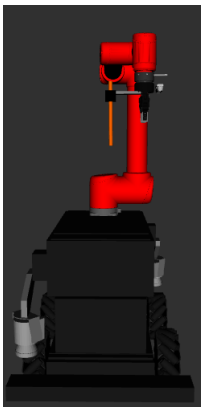
3.3. Creación del paquete de MoveIt

Una vez definido el modelo del robot en URDF es necesario crear un nuevo paquete de configuración de MoveIt, que permita controlar el UR10 con este *framework*. Para crear el paquete se ha utilizado la herramienta MoveIt Setup Assistant, disponible en Moveit, que facilita esta labor. Se ha seguido el tutorial oficial disponible en su web [22] para su creación, que explica de manera detallada los pasos a seguir. El paquete generado se llama *ur10_moveit_official*. Entre las partes más importantes del proceso de creación se encuentra la definición de grupos de planificación, en este caso fueron dos:

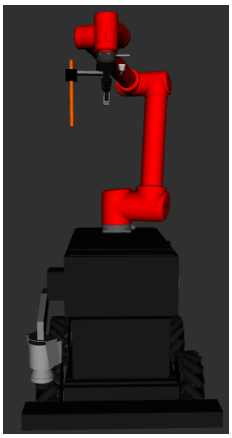
- **manipulator:** comprende los *links* del UR10 desde su base (*base_link*) a la última parte de la muñeca (*wrist_3_link*). Este grupo tiene como *solver* de cinemáticas el *KDLKinematicsPlugin* y como planificador el que viene por defecto, *RTTConnect*.

- **endeffector**: se corresponde con el actuador final que incluye los *links* de la pinza y del sensor de torque y presión. Este grupo no se va a usar activamente, pero es necesario definirlo para evitar colisiones en el proceso de planificación de trayectorias.

También se pueden establecer posiciones para el robot, en este caso se han definido dos posiciones: “up”, figura 3.8a; y “see_aruco”, figura 3.8b.



(a) Posición inicial por defecto en simulación, llamada “up”.



(b) Posición para facilitar la percepción con la cámara de las marcas ArUco situadas sobre la superficie de trabajo, llamada “see_aruco”.

Figura 3.8: Posiciones del robot UR10 definidas.

Para configurar una posición inicial por defecto a un grupo de planificación en simulación hay que añadir la posición deseada en el archivo de configuración *fake_controllers.yaml* del paquete de MoveIt generado, figura 3.9.

```
initial:
  - group: manipulator
    pose: up
```

Figura 3.9: Posición “up” establecida como la inicial por defecto en simulación al grupo manipulator.

El robot Campero también consta de ruedas y de una cámara delantera que son articulaciones activas (se pueden mover), pero que no son útiles para el control del UR10 y, por lo tanto, hay que definirlas como pasivas. De esta forma se evita publicar los estados de dichas articulaciones y, por consiguiente, la reducción de errores.

Para lanzar el robot UR10 en simulación se ejecuta el siguiente comando, el cual también lanza la interfaz de Rviz:

```
roslaunch ur10_moveit_official demo.launch
```

3.4. Definición del entorno de trabajo

Nuestro entorno de trabajo va a estar formado por dos objetos:

- "Mesa" de trabajo: como se va a dibujar sobre la superficie del robot Campero y con el objetivo de evitar que se pueda dañar al realizar algún movimiento incorrecto, se ha colocado una caja sobre el robot que hace de superficie de apoyo para el dibujo, como se aprecia en la figura 3.2.

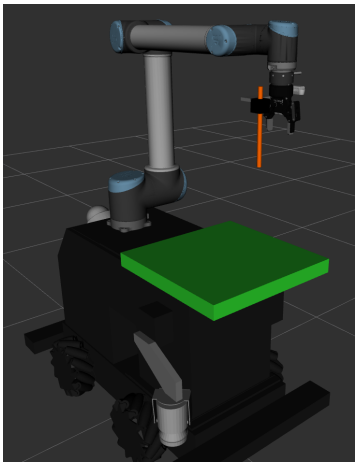
- Pantalla protectora invisible: se trata de una pantalla virtual que se coloca alrededor del robot. Tiene como objetivo evitar planificaciones que generen movimientos amplios y bruscos en el robot, así como limitar el espacio de trabajo del robot a un volumen alejado de singularidades. Indirectamente, permite proteger el material y a los trabajadores que se encuentren en sus proximidades gracias a esa limitación de movimiento.

A través de MoveIt podemos definir entornos de trabajo, llamados escenas, de manera sencilla mediante archivos '.scene', formados por objetos no móviles. Además, MoveIt también tiene en cuenta las colisiones con los objetos de las escenas a la hora de planificar y ejecutar movimientos. Por tanto nuestra escena, llamada *ws_walls.scene* y ubicada en del paquete *campero_ur10_description*, contiene los siguientes objetos:

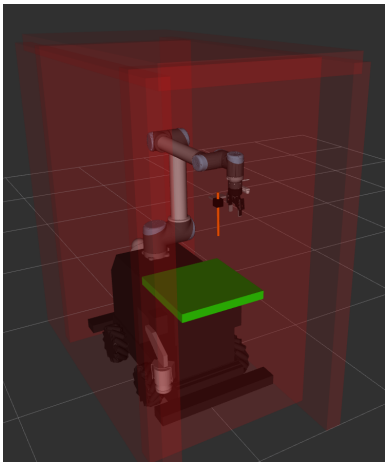
- "Mesa" de trabajo: se ha definido mediante una caja de color verde de 50 cm de ancho, 50 cm de largo y 5 cm de grosor.

- Pantalla protectora: se ha definido mediante 5 planos invisibles formando una caja abierta por abajo en torno al robot Campero.

La escena definida para el trabajo junto con el formato de los archivos '.scene' se puede ver en el Anexo A. En la figura 3.10 se pude visualizar la escena creada junto con el robot Campero.



(a) Escena resultante.



(b) Escena con la pantalla protectora formada por cajas (semi-transparentes para facilitar su visualización).

Figura 3.10: Escena *ws_walls.scene* vista en Rviz.

Capítulo 4

Ejecución en entorno real

En este capítulo se va a hablar sobre la configuración previa necesaria que hay que realizar, tanto en el robot UR10 como en el ordenador donde se aloja el proyecto, para ejecutar los demostradores en un entorno real. También se explicarán los pasos a seguir para iniciar y ejecutar paquetes de ROS en el robot.

4.1. Medidas de seguridad

Lo primero de todo es necesario hablar sobre medidas de seguridad que posee el robot. En simulación no hay ningún problema si el robot choca, realiza movimientos inesperados, etc. Por contra, en el entorno real hay que tener más cuidado y evitar todos los posibles problemas.

Puesto que el UR10 es un robot colaborativo, está pensado para trabajar de manera estrecha con los humanos y cuenta con más medidas de seguridad que un robot manipulador tradicional (como ya se explicó en el apartado 2.1). Al tratarse de un robot colaborativo y estar sobre un robot móvil, no cuenta con una pantalla protectora y, para evitar posibles incidencias en caso de producirse algún fallo, se definió una caja de seguridad invisible en ROS que rodea al robot (ya mencionada en el apartado 3.4). Además, durante la ejecución de cualquier experimento se deja una distancia de seguridad entre las personas y el robot de al menos 1.3 metros, este valor se corresponde con el alcance del UR10 [1].

Otro sistema de seguridad muy importante es el "Mando de seguridad" (figura 4.1), que permite detener la ejecución del robot en cualquier momento pulsando la seta de emergencia (botón rojo). Una vez pulsada la seta de emergencia para poder volver a mover el robot será necesario pulsar el botón verde y reiniciar el programa. Por este motivo es muy importante que, siempre que se vaya a mover el robot, un trabajador tenga sujeto el mando y esté atento en todo momento para detener el robot en caso de que se vaya a producir un choque.

Figura 4.1: Mando de Seguridad.

El proceso de puesta en marcha del robot tras la activación de la seta de emergencia es largo y tedioso. Para poder parar el robot en situaciones no peligrosas sin hacer uso de la seta de emergencia se desarrolló el programa *stop_robot.py*, disponible en el paquete *monitoring_tools*, que permite detener la ejecución del robot a través de MoveIt. La utilización de este programa no exime del uso del mando de seguridad, puesto que el programa podría fallar o podría darse el caso de que el ordenador en un momento determinado no respondiese. La explicación del programa se puede encontrar en el Anexo B.

En simulación, las articulaciones del robot tienen unas velocidad máximas muy altas por lo que en el entorno real habrá que reducirlas. Esto nos permitirá tener un mayor margen de tiempo para detenerlo en caso de que se vaya a producir un choque. La configuración de las velocidades máximas se detallará en el apartado 4.2.1.

4.2. Configuración y ejecución

Como en la mayoría de robots comerciales el UR10 funciona bajo un software desarrollado por el mismo fabricante, concretamente el software *PolyScope v3.1*. Por tanto, para que ROS pueda comunicarse con el robot es necesario realizar una serie de configuraciones tanto en el robot como en el ordenador que contiene el proyecto. Para ello se ha seguido la guía de Universal Robots [21], que explica de forma detallada todos los pasos necesarios.

4.2.1. Configuración

- Configuración del hardware:

    1. UR10: lo primero es configurar el robot, para ello es necesario instalar la extensión *externalcontrol-1.0.5.urcap* como se explica en la sección *Prepare the Robot*. En la guía pone que es necesario tener como mínimo la versión 3.7 de *PolyScope*, pero se probó a instalarla sobre la versión disponible en nuestro UR10,

versión 3.1 del software, y funcionó adecuadamente. Posteriormente hay que crear un programa vacío y cargar la extensión, como se explica en esta otra guía [23].

2. Ordenador: instalar los paquetes y sus dependencias como se explica en la sección *Building*.
3. Calibración: por último hay que extraer la calibración del robot, que nos da información sobre los parámetros de calibración del modelo cinemático del robot. Aunque este paso es optativo, se recomienda hacerlo puesto que mejora la precisión. La calibración se extrae corriendo el programa del primer paso y ejecutando el siguiente comando en el ordenador:

   roslaunch ur_calibration calibration_correction.launch robot_ip:=192.168.0.210/ target_filename:="${HOME}/campero_ur10_calibration.yaml"

- Configuración del proyecto:

  - Fichero velocidades reales: el robot en simulación se mueve a una velocidad muy alta por lo que necesitamos reducirla. Para ello creamos un nuevo archivo llamado *joint_limits_real.yaml*, en el paquete *ur10_moveit_official*, con los mismos valores que el original, *joint_limits.yaml*, pero estableciendo en 0.05rad/s la velocidad máxima de las articulaciones *shoulder_pan*, *shoulder_lift* y *elbow*. Las velocidades de las articulaciones de la muñeca (*wrist*) se mantienen, puesto que no suponen un peligro. Posteriormente, el archivo original de velocidades se renombró a *joint_limits_sim.yaml*. Para poder elegir entre los dos ficheros de velocidades se ha añadido el parámetro *sim* al archivo *move_group.launch*. El parámetro *sim* se establece a True para utilizar los valores de simulación y a False para establecer los valores del robot real.
  - Fichero controladores: es necesario cambiar en el fichero *controllers.yaml*, del paquete *ur10_moveit_official*, el valor del campo *action_ns* a "scaled_pos_joint_traj_controller/follow_joint_trajectory", de lo contrario MoveIt no será capaz de controlar el robot.
  - Fichero bringup: creación del fichero *campero_ur10_bringup.launch* en el paquete *campero_ur10_description*. Este fichero nos permite establecer la comunicación entre el *driver* del robot y ROS. Se trata de una simplificación de los archivos originales *ur10_bringup.launch* y *ur_common.launch*, incluidos en el paquete *ur_robot_driver*.

Entre los parámetros más importantes del nuevo archivo se encuentran los siguientes:

- robot_ip: dirección IP del robot, cuyo valor por defecto es la IP del UR10 del Campero, '*192.168.0.210*'.
- controller_config_file: archivo de configuración *yaml*, donde se encuentran los controladores y parámetros que usará la interfaz hardware. El fichero por defecto es *config/ur10_controllers.yaml*, ubicado en el paquete *campero_ur10_description*. Simplemente se trata de una copia del mismo archivo que se puede encontrar en el paquete *ur_robot_driver*, pero de esta forma está todo más organizado.
- kinematics_config: fichero de calibración, en este caso el extraído en la sección anterior.
- robot_description_file: archivo de descripción URDF del robot, en este caso *campero_ur10.urdf.xacro*

4.2.2. Ejecución

A continuación se enumeran los pasos a seguir para establecer correctamente la comunicación entre el robot real y ROS.

1. Lanzar nodo *master* de ROS:

   ```
   roscore
   ```

2. Lanzar *driver*:

   ```
   roslaunch campero_ur10_description /
       campero_ur10_bringup.launch
   ```

3. Ejecutar el programa urcap: desde el ordenador del UR10 lanzar el programa creado en el paso 1 de configuración del hardware, sección 4.2.1. Si todo funciona correctamente en la terminal que corre el *driver* aparecerá el mensaje: "Robot ready to receive control commands". Si se detiene el *driver* o si se pulsa la seta de emergencia habrá que parar el programa y volverlo a ejecutar.

4. Lanzar MoveIt:

   ```
   roslaunch ur10_moveit_official /
       ur10_moveit_planning_execution.launch
   ```

   El launch llama al archivo *move_group.launch* con *sim:=false* para utilizar el archivo de velocidades reales.

5. Lanzar Rviz:

```
roslaunch ur10_moveit_official /
    moveit_rviz.launch config:=true
```

Con el parámetro *config* a True, se cargará nuestro fichero de configuración de Rviz. Si no se configura el parámetro a True se generará un archivo de configuración Rviz por defecto y no se visualizarán los elementos correctamente.

Capítulo 5

Calibración de la posición del soporte

La posición de la punta del rotulador con respecto al *end-effector* es desconocida y propensa a pequeñas variaciones entre usos. Por esta razón es necesario desarrollar un método de calibración que devuelva las coordenadas de posición O_x, O_y y O_z de la punta del rotulador con respecto al *end-effector* en metros. En la figura 5.1 se pueden visualizar los parámetros de calibración sobre el robot UR10.

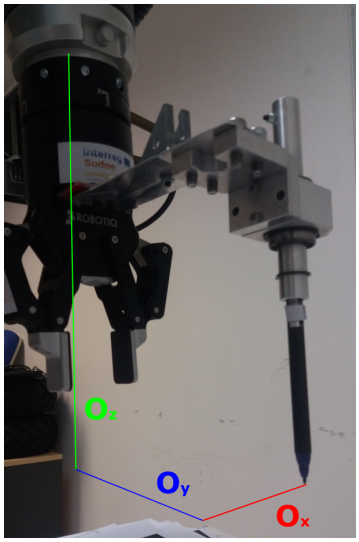


Figura 5.1: Parámetros de calibración O_x, O_y y O_z de la punta del rotulador con respecto al *end-effector* vistos en el robot UR10. Cada parámetro está marcado con un color, correspondiente a su eje, y una línea que indica la distancia desde la punta del rotulador al *end-effector* en el eje del parámetro.

Estos parámetros se colocan después en el archivo de configuración *ur10_draw.config* del paquete *campero_ur10_server* y en la posición de la articulación *base_pen_joint* del archivo de descripción del Campero. Para realizar esta tarea se desarrollaron, de manera secuencial, dos métodos distintos:

- **Medición aproximada**: este primer método se utilizó para el primer demostrador, y solo se hace uso de los parámetros X e Y, como se explica en el capítulo de dicho

demostrador (capítulo 7). Consiste en medir de manera manual la distancia que separa la articulación *wrist_3*, que coincide con el origen del *end-effector*, y la punta del rotulador en los ejes X e Y, de esta forma se obtienen unas medidas próximas a la realidad. Los resultados obtenidos fueron los siguientes:

$$\vec{O} = (O_x, O_y) = (0.03, 0.13)m \tag{5.1}$$

- **Utilización marcas Aruco**: en el segundo demostrador se requiere que el rotulador repase los contornos capturados con una cámara, por tanto, es necesario que la precisión de la calibración sea mayor que en el caso anterior. Aprovechando que se hace uso de las marcas ArUco, éstas se pueden emplear para calibrar la herramienta. Para ello se hace uso de una marca de tamaño $58 \times 58mm$. El proceso de calibración es el siguiente:

  1. Acercar la cámara lo máximo posible a la marca para que esta ocupe lo máximo posible en la imagen. Esta parte se hace utilizando el programa *moveit_commander*, incluido en la librería MoveIt, que permite mover el robot de una manera precisa y sencilla. Una vez posicionado el robot, se guarda la posición de la marca respecto al mundo $Aruco_w$. Como la posición de las marcas ArUco presenta valores con ruido, la posición resultante se coge de una media de mil muestras. Para ello se creó el programa *aruco_distance.py*, localizado en el paquete *monitoring_tools*, que nos devuelve la media de la posición de las marcas ArUco que le especifiquemos tomando un número de muestras preestablecido.
  2. Utilizando el mismo programa de control que en el paso anterior, movemos el robot hasta que la punta del rotulador esté apoyado en el centro de la marca ArUco y se guarda la posición del *ee_link* respecto al mundo EE_w.
  3. Por último, se calcula el vector que va desde la posición EE_w a la posición $Aruco_w$. Las componentes resultantes se corresponderán con los parámetros de calibración O_x, O_y y O_z, respectivamente. Esto es:

     $$\vec{O} = (\vec{Aruco_w} - \vec{EE_w}) = (O_x, O_y, O_z) = (0.0435, 0.129, 0.264)m \tag{5.2}$$

     Donde $\vec{Aruco_w}$ y $\vec{EE_w}$ son los vectores de posición de los puntos $Aruco_w$ y EE_w, respectivamente.

También añadir que, una vez terminada la parte práctica del trabajo, se encontró un paquete desarrollado por Intel para la calibración de herramientas mediante el uso de una cámara y de marcas ArUco [24] que puede ser de gran ayuda para futuros proyectos.

Capítulo 6

Arquitectura de la solución

6.1. Introducción

Las aplicaciones implementadas se pueden clasificar en dos grupos: aplicaciones de dibujo, que son "Pizarra Virtual" y "Dibujo Contornos"; y aplicaciones de teleoperación, que incluye a "Teleoperación". Además, para mantener el rotulador perpendicular al área de trabajo, todas las aplicaciones contarán con una restricción de perpendicularidad del *end-effector*. Por tanto, se puede apreciar que los demostradores comparten una serie de características comunes. Con el objetivo de no repetir código y abstraer la comunicación con el *framework* de MoveIt, se optó por la creación de un "Servidor". El "Servidor" es el encargado de recibir las operaciones de los demostradores, transformar estas operaciones en movimientos válidos del robot y enviar dichos movimientos a MoveIt con las restricciones necesarias.

Las aplicaciones "Dibujo Contornos" y "Teleoperación" hacen uso de las marcas ArUco para su funcionamiento, concretamente de la posición de éstas respecto a la cámara. Además de la posición de las marcas, la aplicación "Dibujo Contornos" también necesita la imagen capturada por la cámara, donde se encuentran las marcas, y los puntos de las esquinas de cada marca en dicha imagen. Por tanto, es necesario un programa, "ArucoDetector", que reciba las imágenes en tiempo real de la cámara RealSense™ Depth Camera D435 y las procese para obtener la posición de las marcas y los puntos de las esquinas de cada una. Por último, tanto el detector de marcas como la aplicación "Dibujo Contornos" requieren de la siguiente información del sensor de la cámara: las dimensiones de las imágenes con las que la cámara ha sido calibrada, la matriz de proyección y los coeficientes de distorsión.

En la figura 6.1 se muestra la arquitectura de la solución a implementar en ROS:

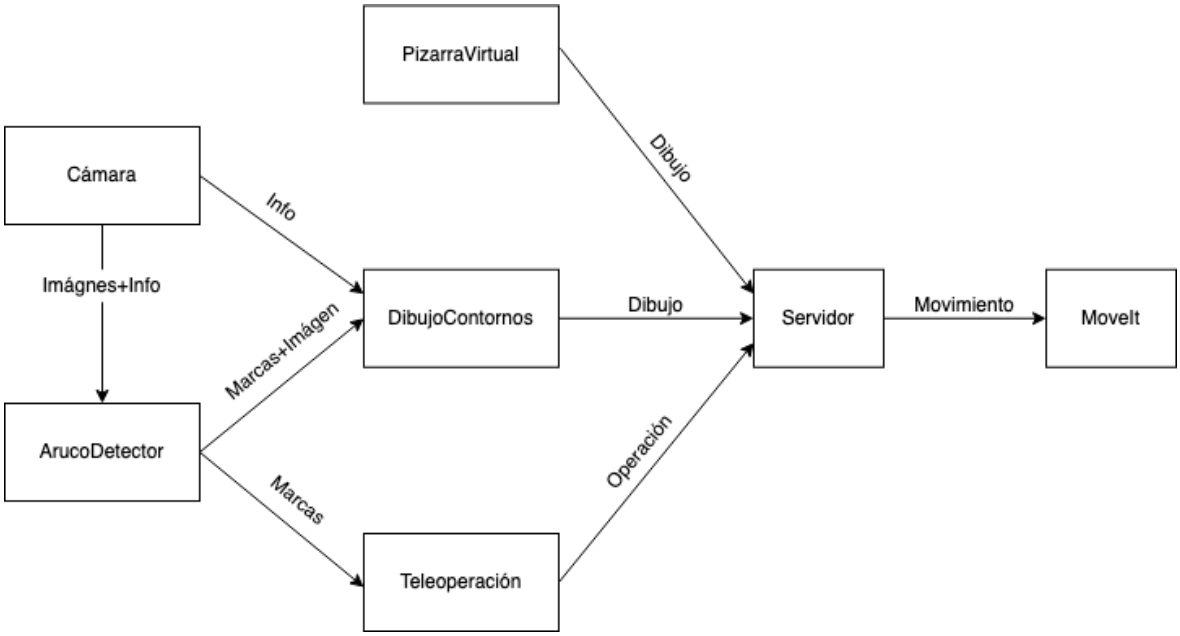


Figura 6.1: Arquitectura en alto nivel de la solución a implementar en ROS. Los rectángulos representarían nodos o conjunto de nodos y las flechas la comunicación vía *topics* y mensajes.

A continuación se van a explicar los principales *topics* y mensajes utilizados. La explicación del nodo Servidor se puede encontrar en el apartado 6.3 y, la explicación del nodo ArucoDetector se encuentra en el Anexo D. En los siguientes capítulos se encuentra el desarrollo y resultados conseguidos de los demostradores implementados.

6.2. Mensajes y topics

Como se ha comentado en la figura 6.1, la comunicación entre los nodos se va a realizar mediante *topics*. Se van a utilizar tanto mensajes ya definidos como mensajes propios, estos últimos se ubican en el paquete *campero_ur10_msgs*.

6.2.1. Cámara

La cámara utilizada dispone de los siguientes sensores: un par estéreo infrarrojo y un sensor RGB. Sólo se va a utilizar la cámara RGB, es decir, imágenes a color. De los *topics* que publica la cámara RGB se van a emplear los siguientes:

- */camera/color/camera_info*: de este *topic* obtenemos información del sensor de la cámara, por ejemplo, matriz de rectificación, matriz de proyección, etc. Los mensajes publicados son del tipo *sensor_msgs/CameraInfo* [25].

- */camera/color/image_rect_color*: aquí se publican las imágenes a color sin comprimir de la cámara. El tipo de mensaje que utiliza es *sensor_msgs/Image* [26].

El nodo de la cámara utilizada se lanza con el siguiente comando:

```
roslaunch realsense2_camara rs_rgbd.launch
```

6.2.2. Marcas ArUco

Los mensajes del paquete de marcas ArUco utilizado [9] no contienen suficiente información como para ser utilizados en el demostrador "Dibujo Contornos". Dicho demostrador necesita no solo la localización del centro de las marcas Aruco, sino también la posición de las esquinas de cada una de las marcas en la imagen. En la figura 6.2 se muestra el centro de la marca junto con sus respectivas esquinas.

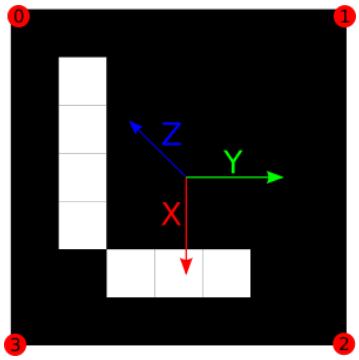


Figura 6.2: Marca ArUco. El centro de la marca representa la posición y orientación de la misma respecto de la cámara. Los puntos rojos representan las esquinas de la marca y, están numeradas según su orden en el vector que las almacena.

Por tanto, se han definido los siguientes mensajes:

- *ArucoMarker*: representa la información de una marca, es decir, su id, posición y orientación con respecto a la cámara y las posiciones de sus cuatro esquinas en la imagen. Implementación disponible en la figura 6.3.

```
######################################
# CONSTANTS
######################################
# longitud vector de esquinas
uint8 NUM_IMG_POINTS = 4

######################################
# Message
######################################
# identificador
uint32 id
# posición y orientacion
geometry_msgs/Pose pose
# posicion de las esquinas en la imagen
geometry_msgs/Point32[] img_points
```

Figura 6.3: Archivo de definición del mensaje *ArucoMarker.msg*.

- *ArucoMarkerArray*: representa un vector de marcas ArUco, figura 6.4.

```
# vector de marcas ArUco
ArucoMarker[] markers
```

Figura 6.4: Archivo de definición del mensaje *ArucoMarkerArray.msg*.

- *ArucoMarkersImg*: representa un vector de marcas ArUco junto con la imagen donde han sido detectadas, figura 6.5.

```
# marcas ArUco
ArucoMarkerArray markers
# imagen que contiene las marcas
sensor_msgs/Image img
```

Figura 6.5: Archivo de definición del mensaje *ArucoMarkersImg.msg*.

El demostrador "Dibujo Contornos" hará uso del *topic /aruco_detector/markers_img*, donde se publican mensajes de tipo *ArucoMarkersImg.msg*, mientras que, el demostrador "Teleoperación" se suscribirá al *topic /aruco_detector/markers_pose*, donde se publican mensajes de tipo *ArucoMarkerArray.msg*.

6.2.3. Dibujos

Las aplicaciones de dibujo pueden enviar dos tipos de dibujos: dibujo real, los puntos que forman dicho dibujo pertenecen al espacio 3D del mundo; y dibujo virtual (figura 6.7), los puntos contenidos en dicho dibujo pertenecen a un plano en 2D. Los dibujos constan de una serie de trazos (figura 6.6), cada uno formado por un conjunto de puntos. Además, los dibujos virtuales constan de un tamaño (anchura y altura en píxeles), su origen se localiza en el punto (0,0) y el valor máximo que puede tomar un punto es (W-1, H-1), siendo W y H la anchura y altura en píxeles del dibujo.

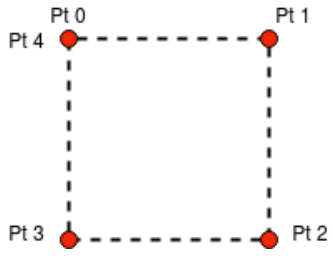


Figura 6.6: Concepto de trazo. Los círculos representan los puntos del trazo y las líneas discontinuas el trazo dibujado. Los trazos están formados por un conjunto de puntos. El orden de aparición de los puntos en el trazo indica el camino a dibujar. Los trazos pueden ser abiertos o cerrados. Con un mayor número de puntos se pueden lograr trazos más complejos, por ejemplo, líneas curvas. La distancia entre puntos no tiene por qué ser equidistante.

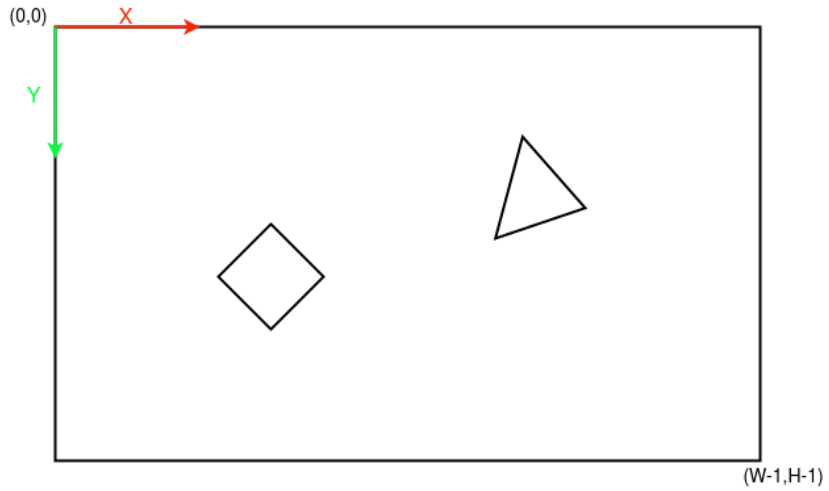


Figura 6.7: Concepto de dibujo virtual. Tiene un tamaño de *W* píxeles de ancho y *H* píxeles de alto. La posición origen corresponde con la (0,0). Los puntos se encuentran en el rango [0,W) para el eje X y [0,H) en el eje Y. Este dibujo consta de dos formas, un triángulo y un rombo, cada una de ellas representaría un trazo distinto.

Para representar los puntos, trazos y dibujos se han definido los siguientes mensajes de ROS:

- *ImgPoint*: representa un punto, tanto 2D como 3D, de un dibujo. Si es un punto 2D la variable *z* no se utiliza. Implementación en la figura 6.8.

```
float32 x # posicion eje x
float32 y # posicion eje y
float32 z # posicion eje z
```

Figura 6.8: Archivo de definición del mensaje *ImgPoint.msg*. Permite definir puntos en un espacio 3D y 2D, en este último caso no se utiliza la variable *z*.

- *ImgTrace*: representa un trazo de un dibujo, formado por un vector de mensajes *ImgPoint*, como se ve en la figura 6.9.

```
# trazo de una imagen
ImgPoint[] points
```

Figura 6.9: Archivo de definición del mensaje *ImgTrace.msg*. Compuesto por un vector de puntos (mensajes *ImgPoint*).

- *ImgDraw*: representa un dibujo virtual o real, el tipo de dibujo se define con la variable *type*, cuyo valor puede ser uno de los dos definidos como constantes en el mensaje. Los dibujos reales ya cumplen con las dimensiones del lienzo al estar definidos en coordenadas del mundo 3D, por tanto, las variables *W* (anchura) y *H* (altura) no se utilizan. El mensaje está formado por un vector de mensajes *ImgTrace*. Estos mensajes

se publican en el *topic /image_points*, al cual se suscribe el "Servidor" para su posterior procesamiento. El mensaje de ROS se puede visualizar en la figura 6.10.

```
#####################################
# CONSTANTS
#####################################
# tipo: dibujo real, plano 3D
uint8 REAL_POINTS = 0
# tipo: dibujo virtual, plano 2D
uint8 BOARD_POINTS = 1

#####################################
# Message
#####################################
uint8 type # tipo de dibujo
ImgTrace[] traces # trazos
int32 W # ancho dibujo virtual
int32 H # altura dibujo virtual
```

Figura 6.10: Archivo de definición del mensaje *ImgDraw.msg*. Compuesto por un tipo de dibujo, *type*, cuyo valor solo puede ser una de las dos constantes definidas; un vector de trazos, mensajes *ImgTrace*; y un tamaño, anchura (*W*) y altura (*H*), en el caso de que se trate de un dibujo virtual.

6.2.4. Teleoperación

El último demostrador, "Teleoperación", permite mover el *end-effector* del robot en el plano XY del mundo 3D a partir de unas órdenes generadas con el desplazamiento manual de una marca ArUco. Se ha definido el mensaje *MoveOp.msg* (figura 6.11) formado por dos variables, *vx* y *vy*. Dichas variables indican el desplazamiento incremental (positivo o negativo) que debe realizar el *end-effector* en las coordenadas X e Y. El *topic* en el cual se publican estos mensajes es */campero_ur10_move*.

```
float32 vx
float32 vy
```

Figura 6.11: Archivo de definición del mensaje *MoveOp.msg*. Las variables *vx* y *vy* indican el desplazamiento en sentido positivo o negativo en los ejes X e Y, respectivamente, que debe realizar el *end-effector*.

6.3. Servidor

El "Servidor" va a ser el nodo, llamado *campero_ur10_server*, encargado de recibir las operaciones de los distintos demostradores, transformar dichas operaciones en movimientos del robot y transmitirlos a MoveIt, para su posterior planificación y ejecución. Además, el Servidor también se encarga de configurar ciertos parámetros de MoveIt, por ejemplo, restricciones de orientación.

El “Servidor” tiene dos modos de funcionamiento disponibles: *dibujo* y *teleoperación*. Ambos modos comparten una serie de características comunes, explicadas en el siguiente apartado.

6.3.1. Configuración de MoveIt

El Servidor es el encargado de enviar, a través del nodo *move_group*, los movimientos a MoveIt para su posterior planificación y ejecución. Se ha especificado *manipulator* como grupo de planificación del robot.

Como se explicó anteriormente los demostradores van a funcionar con la restricción de que el *soporte* esté perpendicular al plano de trabajo, aunque se puede deshabilitar esta opción para el modo *teleoperación*. Para que el *soporte* esté perpendicular hay que añadir una restricción de orientación sobre el *link* del *end-effector* del grupo de planificación, en este caso *wrist_3_link*. La restricción se ha definido mediante el uso de los ángulos de rotación *RPY*, estableciendo a *roll* un valor de π *radianes* y a *pitch* y *yaw* un valor de *0 radianes*. Con esta configuración de valores *RPY* conseguimos que el rotulador se mantenga con una orientación vertical perpendicular al plano de dibujo. Además, hay que definir una pequeña tolerancia de error de ángulo de *0.01 radianes* en cada uno de los ejes, ya que si se establece a *0* MoveIt es incapaz de encontrar soluciones.

En ambos modos se va a utilizar la planificación de caminos cartesianos guiados por *waypoints*, lista de puntos que sirven de guía para trazar trayectorias. Esta planificación requiere de dos parámetros adicionales: *eef_step*, máxima distancia entre dos puntos consecutivos del *end-effector* en la trayectoria resultante, cuyo valor se ha establecido en *0.01 m*; y *jump_threshold*, se establece en 0 para evitar saltos no deseados derivados del cómputo de la cinemática inversa. Cuando MoveIt calcula un camino cartesiano devuelve un valor entre 0 y 1, que indica qué fracción de los *waypoints* son alcanzables. Para evitar problemas solo se ejecutarán aquellas trayectorias que alcancen a todos los *waypoints* definidos.

6.3.2. Modo Dibujo

El Servidor en este modo se va a encargar de procesar los dibujos recibidos de los demostradores “Pizarra Virtual” y “Dibujo Contornos”, para ello se suscribe al *topic /image_points*. Existen dos tipos de dibujos, real y virtual, y sus transformaciones a movimientos del robot son similares, pero difieren en la transformación de los puntos. El servidor requiere de ciertos parámetros para llevar a cabo estas transformaciones, por ejemplo, tamaño y posición del lienzo, offset de posición de la punta del rotulador respecto al *end-effector*, etc. Estos parámetros se definen en el archivo de configuración *ur10_draw.config*, cuya explicación se puede encontrar en el Anexo C.

El robot va a dibujar sobre un lienzo, en este caso se trata de un Din A3, como se aprecia en esta figura 7.1. Para definir geométricamente el lienzo se necesita conocer su posición en el mundo, que corresponde con la esquina con menor x e y (esquina superior izquierda desde referencia robot UR10); y su tamaño. Al tratarse de un Din A3 su tamaño es de *0.42 metros* de ancho y *0.297 metros* de alto. Al transformar los puntos de un dibujo al lienzo, estos tendrán una posición en el eje z constante. Para obtener unos mejores resultados se recomienda que el *aspect ratio* [27] del dibujo virtual sea el mismo que el del lienzo.

El robot se dirige siempre al primer punto de un trazo con el rotulador levantado, para evitar contacto previo con el lienzo. Una vez alcanza dicho punto, apoya el rotulador sobre el lienzo para poder pintar. Al acabar un trazo, levanta de nuevo el rotulador y, de haberlo, se dirige al primer punto del siguiente trazo.

A continuación se muestra en pseudocódigo el algoritmo utilizado para transformar un dibujo a trayectoria:

```
def draw2move(img: ImgDraw):
    # puntos de guiado para la planificación
    waypoints = [];
    # posición y orientación actual del end-effector
    # la orientación se mantiene constante en todo momento
    current_pose = move_group.getCurrentPose();

    # 1. Procesar trazos
    for (trace: img.traces):
        pts = trace.points;
        if (pts.size == 0): continue;
        dst = current_pose;
        # 1.1. Ir al primer punto con el rotulador levantado
        dst.position = drawPt2worldPt(pts[0], z_pen_up);
        waypoints.add(dst);
        # 1.2. Insertamos todos los puntos
        for (pt: pts):
            dst.position = drawPt2worldPt(pt, z_pen_down);
            waypoints.add(dst);
        # 1.3. Levantamos el rotulador al terminar
        dst.position = drawPt2worldPt(pt.last, z_pen_up);
        waypoints.add(dst);

    # 2. Planificar trayectoria y ejecutar
    if (waypoints.size == 0): return;
    if (move_group.computeCartesianPath(waypoints) == 1):
        move_group.execute();
```

En la transformación se utilizan dos constantes: *z_pen_up*, altura del *end-effector* para levantar el rotulador del lienzo, su valor se define en el archivo de configuración; y *z_pen_down*,

altura del *end-effector* para pintar sobre el lienzo.

Tanto el parámetro *z_pen_down* como la forma de transformar los puntos del dibujo a la referencia mundo varían según el tipo de dibujo:

- **Dibujo virtual:** para la transformación de un punto de un dibujo virtual al espacio 3D del mundo, se necesita conocer: posición del lienzo en el mundo, *board_min_x* y *board_min_y*; tamaño del lienzo en metros, ancho (*w_board*) y alto (*h_board*); y la corrección de posición de la punta del rotulador respecto al *end-effector*, *correct_x* y *correct_y*. Se necesita además un factor de escala, calculado a partir del tamaño del lienzo y del dibujo:

$$Scale = (w_{scale}, h_{scale}) = (w\_board/draw.W, h\_board/draw.H) \tag{6.1}$$

  La altura de dibujado toma el valor del parámetro de configuración *z_pen_down*, este valor se estableció llevando al *end-effector* a la altura deseada y guardando el valor de su posición en el eje Z. También hay que tener en cuenta que el ancho se corresponde con el eje X en el dibujo y con el eje Y en el mundo, y viceversa para el alto. A continuación, se muestra la función de transformación en pseudocódigo:

```
def virtualPt2worldPt(pt: ImgPoint, z: double)
                          -> geometry_msgs::Point:
    geometry_msgs::Point res;
    res.x = board_min_x + pt.y * w_scale + correct_x;
    res.y = board_min_y + pt.x * h_scale + correct_y;
    res.z = z;
    return res;
```

- **Dibujo real:** los puntos de este tipo de dibujo pertenecen al espacio 3D del mundo, por tanto, sólo hay que desplazarlos para que el *end-effector* posicione la punta del rotulador sobre dichos puntos. Para ello se hace uso de los parámetros de configuración *correct_x* y *correct_y*. La altura de dibujado, *z_pen_down*, es constante y se calcula a partir de la suma de la altura del lienzo, parámetro *board_z*, mas la corrección en el eje Z de la punta del rotulador respecto al *end-effector*, parámetro *correct_z*. La función de transformación en pseudocódigo quedaría de la siguiente forma:

```
def realPt2worldPt(pt: ImgPoint, z: double)
                          -> geometry_msgs::Point:
    geometry_msgs::Point res;
    res.x = pt.x + correct_x;
    res.y = pt.y + correct_y;
    res.z = z;
    return res;
```

6.3.3. Modo Teleoperación

En este modo el Servidor se suscribe al *topic /campero_ur10_move* y espera a recibir mensajes, de tipo *MoveOp*. La transformación de estos mensajes a movimientos del robot, y su posterior ejecución, es muy rápida, por tanto, el tamaño de la cola de mensajes que es capaz de almacenar el nodo se ha establecido en 100, de esta forma se evita la pérdida de mensajes y, por consiguiente, la teleoperación funciona de forma más fluida.

Al principio se utilizó la planificación de trayectorias sin *waypoints*, puesto que en este modo cada movimiento del robot solo va a constar de una posición objetivo. Pero los resultados conseguidos no fueron buenos, debido al excesivo tiempo que tardaba en encontrar una solución para el desplazamiento. Por tanto, se pasó a utilizar la planificación de caminos cartesianos mediante *waypoints*, mejorando así el tiempo de respuesta de la aplicación.

A continuación se muestra en pseudocódigo la transformación de un mensaje de teleoperación a trayectoria del robot:

```
def teleop2move(op: MoveOp):
    # 1. Obtenemos la posición actual del end-effector.
    current_pose = move_group.getCurrentPose();

    # 2. Actualizamos la posición actual
    # con el desplazamiento del mensaje recibido.
    # La orientación y altura se mantienen igual.
    current_pose.position.x += op.vx;
    current_pose.position.y += op.vy;

    # 3. Vector de waypoints.
    waypoints = [current_pose];

    # 4. Planificación y ejecución.
    if (move_group.computeCartesianPath(waypoints) == 1):
        move_group.execute();
```

Capítulo 7

Primer demostrador: Pizarra Virtual

Con el objetivo de poner en práctica los conocimientos aprendidos y poner en funcionamiento por primera vez el robot UR10, se propone desarrollar la siguiente aplicación.

7.1. Descripción de la aplicación

Esta aplicación permite al usuario realizar un dibujo en el ordenador, que posteriormente es enviado al robot para que lo replique sobre un lienzo. El robot debe realizar exactamente el mismo dibujo, es decir, debe pintar el mismo número de trazos y en el mismo orden, siguiendo los puntos establecidos en cada trazo. En la figura 7.1 se puede ver el *setup* utilizado para esta aplicación.

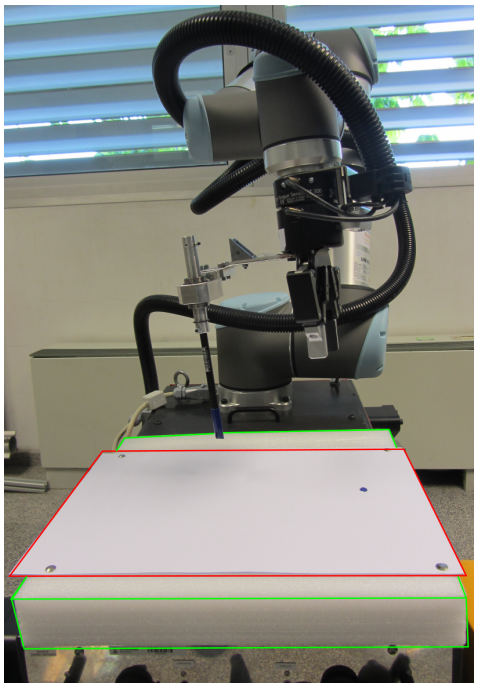
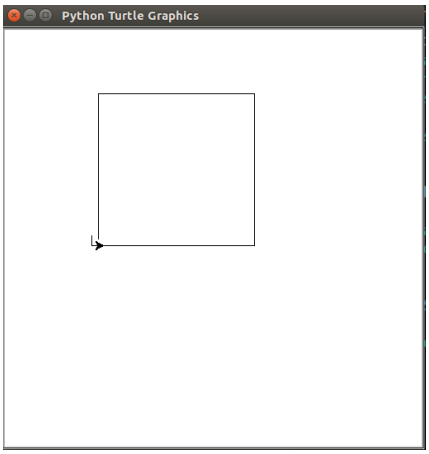


Figura 7.1: *Setup* utilizado por el robot para dibujar. Las líneas rojas delimitan el lienzo sobre el que se puede pintar y, las líneas verdes la mesa de trabajo, que sirve como apoyo para el rotulador y como protección para el robot.

Además, con el objetivo de poder hacer distintos tipos de dibujo, se han desarrollado dos aplicaciones:

- *DrawBoardCV*: permite realizar dibujos a mano alzada utilizando el ratón del ordenador (figura 7.2a).
- *DrawBoardTurtle*: permite realizar dibujos con líneas rectas mediante el uso del teclado (figura 7.2b).

(a) Interfaz de la aplicación *DrawBoardCV*, permite realizar dibujos a mano alzada con el ratón.



(b) Interfaz de la aplicación *DrawBoardTurtle*, permite realizar dibujos más precisos utilizando el teclado.

Figura 7.2: Interfaces de usuario de las aplicaciones desarrolladas para el demostrador Pizarra Virtual.

7.2. Desarrollo de la aplicación

La implementación de la aplicación se encuentra en el paquete *draw_board*. Al tener que realizar dos aplicaciones de dibujo que comparten una serie de características, comunicación con ROS y manejo de la información del dibujo, se decidió crear una clase para cada una, *DrawNode* y *Draw* respectivamente. Además, ambas clases se encuentran contenidas dentro de la clase *Board*, con el objetivo de abstraer el comportamiento y facilitar el desarrollo de las aplicaciones. Por tanto, ambas aplicaciones harán uso de esta clase en su implementación. En la figura 7.3 se muestra el esquema en alto nivel de la solución implementada.

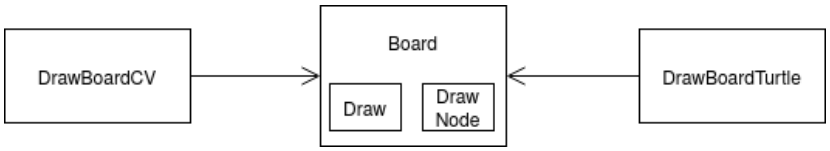


Figura 7.3: Arquitectura de la solución planteada para implementar las aplicaciones de dibujo, *DrawBoardCV* y *DrawBoardTurtle*. Ambas aplicaciones hacen uso de la clase *Board*, que se encarga de la comunicación con ROS y el manejo de los dibujos.

El nodo creado por la clase *DrawNode*, llamado *draw_board*, publica los dibujos en el nodo */image_points* cuando recibe la orden de enviar. La clase *Draw* ofrece las siguientes funcionalidades: borrar el dibujo, añadir un nuevo trazo, añadir un punto al último trazo creado y transformar un dibujo a mensaje de tipo *ImgDraw*. Los puntos que almacena la clase tienen que estar en el rango [0,W) en el eje X y [0,H) en el eje Y, dónde W y H representan el ancho y la altura en píxeles del dibujo.

La aplicación *DrawBoardCV* ha sido implementada utilizando la librería OpenCV [7], puesto que permite dibujar con el ratón de manera sencilla sin tener que escribir mucho código. Para dibujar un trazo simplemente hay que presionar el botón izquierdo del ratón y soltar cuando se desee finalizar el trazo. El usuario además puede interactuar con las siguientes teclas: tecla 'ESC', el programa termina; tecla 'c', el dibujo se borra; y con la 's', se pública un mensaje con el dibujo realizado. Para lanzar este programa se utiliza el siguiente comando:

```
python draw_board_cv [-W <width>] [-H <height>]
```

Por otro lado la aplicación *DrawBoardTurtle* se ha desarrollado utilizando la librería Turtle Graphics [28]. Esta aplicación permite crear una ventana para dibujar junto con un cursor que indica dónde se está pintando. Para pintar hay que desplazar el cursor por la pantalla utilizando las flechas del teclado. El cursor tiene dos estados: "levantado" y "pintar". En el estado "levantado" la aplicación no pinta en el dibujo mientras se desplaza el cursor, mientras que en el estado "pintar" el cursor sí pinta mientras se desplaza. Para cambiar de un estado a otro hay que pulsar la tecla 'p'. Se puede borrar el dibujo pulsando la tecla 'c' y se puede enviar el dibujo mediante la tecla 's'. Por último, para terminar el programa, basta con pulsar la tecla 'z'.

```
python draw_board_turtle [-W <width>] [-H <height>]
```

7.3. Resultados

Para las pruebas en el entorno real sólo se pudo utilizar la aplicación *DrawBoardCV*, esto es debido a que en el ordenador del robot Campero no se pudieron instalar las librerías necesarias para hacer funcionar la aplicación *DrawTurtle*.

El tamaño de los dibujos en la aplicación es de 420×297 píxeles, por lo que mantiene el mismo *aspect ratio* de 1.41 que el lienzo: un Din A3 cuyo tamaño es de $0{,}42 \times 0{,}297$ metros. El robot se mueve a una velocidad máxima de 0.05 rad/s por articulación (véase 4.2.1). A continuación se muestran imágenes comparativas, entre el dibujo original y el dibujo realizado por el robot, de los resultados conseguidos, figuras 7.4 y 7.5. Además, hay un vídeo disponible en la lista de reproducción [10] en el que se pueden visualizar el proceso de dibujado y los resultados.

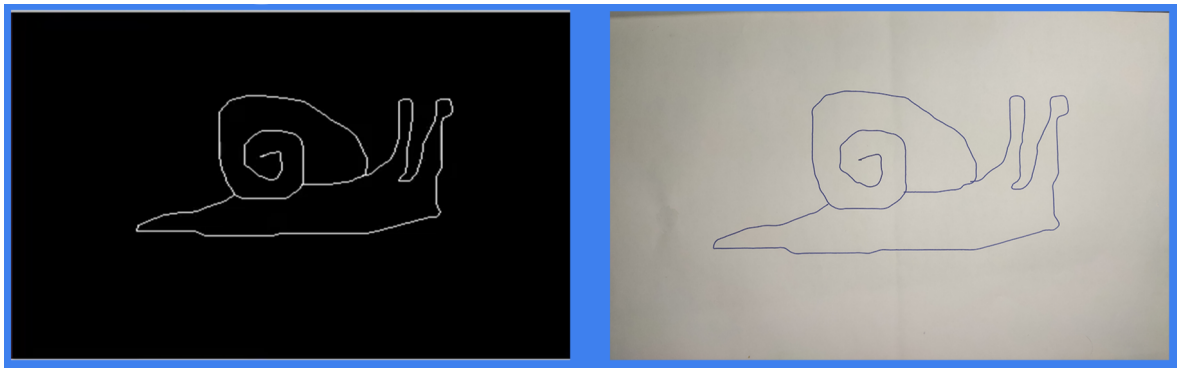


Figura 7.4: Dibujo de un caracol, el cual está formado por varios trazos. A la izquierda se encuentra el dibujo original realizado en la aplicación *DrawBoardCV* y a la derecha el dibujo realizado por el robot. El tiempo total empleado por el robot fue de 48 segundos.

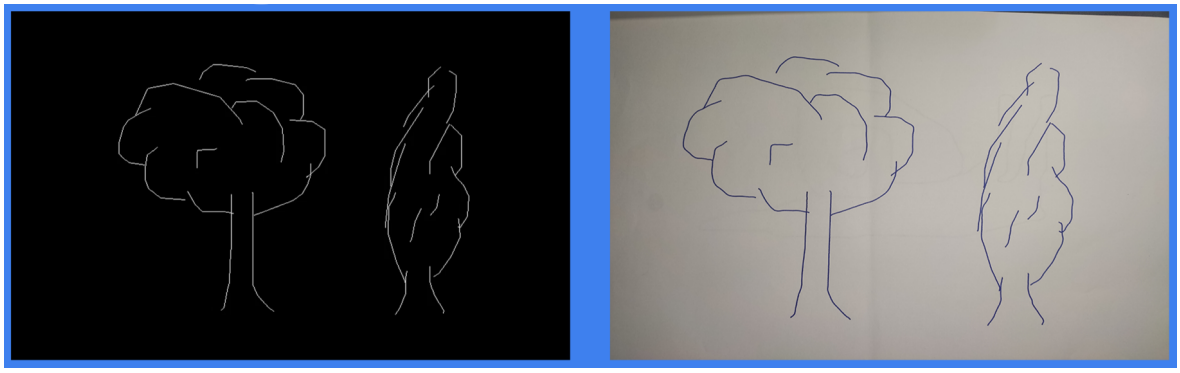


Figura 7.5: Dibujo de unos árboles, se aprecia que está formado por varios trazos. A la izquierda se encuentra el dibujo original realizado en la aplicación *DrawBoardCV* y a la derecha el dibujo realizado por el robot. El tiempo total empleado por el robot fue de 2 minutos y 16 segundos.

Como última prueba se hizo una comparación entre un dibujo realizado con una velocidad máxima lenta, 0.05 rad/s, y un dibujo realizado con una velocidad máxima de 0.15 rad/s. El resultado, como se aprecia en la figura 7.6, es que a mayor velocidad el rotulador empieza a vibrar demasiado, lo que provoca perturbaciones en el trazo del dibujo. Por otro lado, con una menor velocidad máxima el trazado del dibujo es mucho más suave.

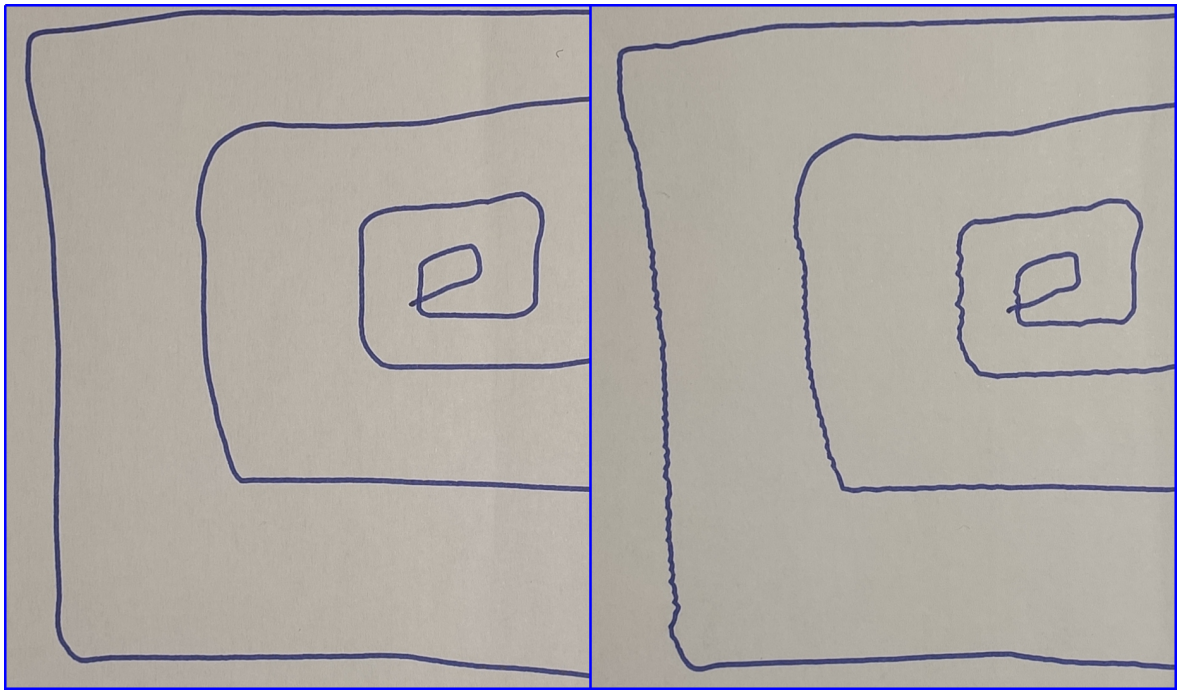


Figura 7.6: Comparación de un dibujo realizado a baja velocidad máxima (0.05 rad/s), imagen de la izquierda, frente al mismo dibujo realizado a una velocidad máxima mayor (0.15 rad/s), imagen de la derecha. En la imagen de la derecha se aprecian perturbaciones en el trazo dibujado, debidas a la vibración del rotulador provocada por el robot al moverse a una alta velocidad. El tiempo empleado por el robot a velocidad baja fue de 3 minutos y 28 segundos, mientras que a velocidad alta tardó 1 minuto y 10 segundos.

Capítulo 8

Segundo demostrador: Dibujo Contornos

Al obtener unos resultados satisfactorios con el primer demostrador, se procedió a realizar una segunda aplicación de mayor complejidad que incluye el uso de la visión por computador para el análisis de imágenes en tiempo real. Este demostrador tiene como objetivo demostrar que se pueden realizar aplicaciones enfocadas a procesos más industriales, por ejemplo, soldar una línea, aplicar pegamento, etc.

8.1. Descripción de la aplicación

La aplicación consiste en el reconocimiento de los contornos de los objetos dispuestos sobre la mesa de trabajo para que, posteriormente, el robot dibuje dichos contornos sobre el lienzo, como se aprecia en la figura 8.1. Para esta aplicación se va a utilizar un lienzo con cuatro marcas ArUco, una en cada esquina, que nos permitirán transformar puntos del plano imagen a puntos en coordenadas de la cámara RGB. El nodo encargado de llevar a cabo esta tarea es *image_inpainting* del paquete *image_inpainting*, el cual ha sido desarrollado utilizando la librería OpenCV para C++.

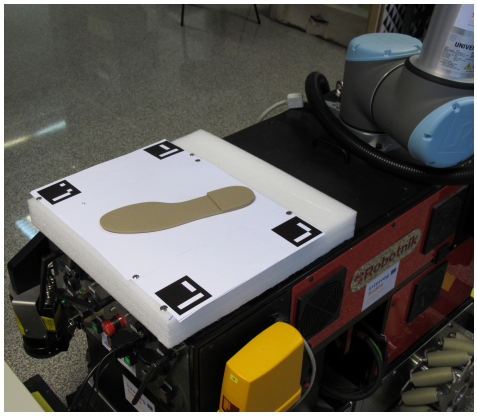


Figura 8.1: Mesa de trabajo junto con el lienzo y las cuatro marcas utilizadas en el segundo demostrador. Sobre el lienzo se encuentra una suela de zapato cuyo contorno se quiere extraer.

Los pasos seguidos para obtener los resultados son los siguientes:

1. Inicio de la cámara RGB de la RealSense D435.

2. Los nodos *aruco_detector* e *image_inpainting* reciben la información relativa al sensor de la cámara.

3. Movemos el robot a la posición "see_aruco" (Figura 3.8b), definida en el paquete de MoveIt.

4. El nodo *aruco_detector* obtiene la imagen de la cámara, la procesa y consigue la información de cada marca. Posteriormente publica la imagen y las marcas en el *topic /aruco_detector/markers_img*.

5. El nodo *image_inpainting* recibe el mensaje de dicho *topic* y, a continuación, realiza las siguientes acciones:

   a) Cambio de perspectiva sobre la imagen, con el objetivo de facilitar la búsqueda de contornos.

   b) Calculo de los parámetros de transformación de la cámara a la imagen rectificada.

   c) Obtención de los contornos sobre la imagen rectificada.

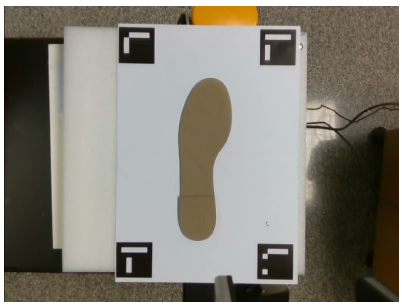
   d) Transformación de los puntos de los contornos obtenidos a puntos del mundo y publicación de los resultados en el *topic /image_points*.

6. Finalmente, el nodo *campero_ur10_server* recibe el "dibujo" y planifica y ejecuta la trayectoria resultante.

8.2. Desarrollo de la aplicación

8.2.1. Cambio de perspectiva de la imagen

El primer paso consiste en realizar un cambio de perspectiva sobre la imagen original, con el objetivo de obtener una imagen rectificada en la que solo es visible la zona delimitada por las marcas en el lienzo, como se ve en la figura 8.2. Este cambio de perspectiva facilita la detección de los contornos sobre los objetos situados en la mesa de trabajo.



(a) Imagen original obtenida de la cámara RGB de la RealSense™ Depth Camera D435.

(b) Imagen rectificada aplicando un cambio de perspectiva sobre la imagen original.

Figura 8.2: Cambio de perspectiva de la imagen original realizada por el nodo *image_inpainting*, con el objetivo de obtener la zona delimitada por las marcas en el lienzo.

Para obtener este cambio de perspectiva se han seguido los siguientes pasos:

1. Obtención de los cuatro puntos que delimitan la zona del lienzo y las marcas en la imagen original. Estos puntos se consiguen mediante un filtrado sobre los puntos de las esquinas de todas las marcas. Los puntos resultantes son: *top_left*, *top_right*, *bottom_right* y *bottom_left*. Al conjunto formado por estos cuatro puntos se le denomina puntos fuente.

2. Cálculo de los puntos correspondientes a cada punto fuente en la imagen rectificada, denominados puntos destino.

```
# 1. Obtenemos ancho y alto formado por
# los puntos fuente en la imagen original.
width_top = distance(top_left, top_right)
width_bottom = distance(bottom_left, bottom_right)
maxW = max(width_top, width_bottom)
height_left = distance(top_left, bottom_left)
height_right = distance(top_right, bottom_right)
maxH = max(height_left, height_right)

# 2. Calculamos los puntos destino.
top_left_dst = (0, 0)
top_right_dst = (maxW, 0)
bottom_right_dst = (maxW, maxH)
bottom_left_dst = (0, maxH)
```

3. Obtención de la matriz de homografía H a partir de la matriz P, formada por los puntos fuente en coordenadas homogéneas; y la matriz Q, formada por los puntos destino en

coordenadas homogéneas.

$$\begin{aligned} &H_{3\times3} * P_{3\times4} = Q_{3\times4} \\ &H_{3\times3} * P_{3\times4} * P_{4\times3}^{T} = Q_{3\times4} * P_{4\times3}^{T} \\ &H_{3\times3} = Q_{3\times4} * P_{4\times3}^{T} * (P_{3\times4} * P_{4\times3}^{T})^{-1} \end{aligned} \tag{8.1}$$

4. Se obtiene la imagen rectificada a partir de la imagen original y la matriz de homografía. La nueva imagen tendrá un tamaño de *maxW* píxeles de ancho y *maxH* píxeles de alto.

Para el paso siguiente, apartado 8.2.2, se necesita conocer la posición de las marcas ArUco en la imagen rectificada. El centro de una marca se considera su posición en dicha imagen. El centro se obtiene calculando el punto medio de las esquinas de la marca en la imagen original y, por último, aplicando a dicho punto la matriz de homografía H.

8.2.2. Parámetros de transformación de la cámara RGB al plano imagen

Para pasar los puntos en coordenadas de la cámara RGB al plano imagen (imagen rectificada sin distorsión) se necesita obtener una matriz de transformación ${}^{I}T_{C}$, formada por la matriz de rotación R y el vector de traslación t. Para calcular ${}^{I}T_{C}$ es necesario disponer de la matriz intrínseca de la cámara K, que se obtiene directamente del *topic /camera/color/camera_info*, y de 8 puntos conocidos. Estos puntos se corresponden con las posiciones de las cuatro marcas ArUco con respecto a la cámara y sus correspondientes en el plano imagen. Con esta información se puede obtener R y t a partir de la siguiente fórmula:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K_{3\times3} * (R_{3\times3}|t_{3\times1}) * \begin{bmatrix} X_x^C \\ X_y^C \\ X_z^C \\ 1 \end{bmatrix} \tag{8.2}$$

Donde (u, v) representa un punto en el plano imagen y X^C representa un punto en coordenadas de la cámara. Se ha utilizado la función *solvePnP* de la librería OpenCV [29] para resolver el sistema de ecuaciones generado a partir de la fórmula 8.2 y los 8 puntos conocidos. Esta función devuelve el vector de traslación t y el vector de rotación r. Posteriormente, el vector de rotación r se transforma a matriz de rotación R aplicando la función *Rodrigues*.

La función *solvePnP* no es determinista y puede devolver resultados erróneos. Para comprobar que R y t son correctas se transforma la posición de las marcas en coordenadas de la cámara al plano imagen aplicando la fórmula 8.2. Los puntos obtenidos se comparan con sus correspondientes ya conocidos y, si la distancia en píxeles entre ellos es mayor que un valor fijo (parámetro *max_dist_error*) se considera un resultado no válido y se cancela el

procesamiento actual. Indicar también que el nodo almacena la imagen y los contornos de dicha imagen que tenga menor error de distancia medio.

8.2.3. Obtención de contornos

Para la extracción de contornos se utiliza segmentación de imágenes. Para segmentar la imagen recibida se desarrolló un primer método sencillo llamado 'Simple' y, posteriormente, partiendo del anterior método se creó otro llamado 'Watershed', en el cual se aplica el algoritmo de watershed [30]. El método a emplear en el procesamiento de la imagen se especifica con el parámetro *contour_method*, si vale 0 se aplica el primer método, si su valor es 1 se aplica el segundo.

Método Simple

El procedimiento seguido para la segmentación de imágenes con este método es el siguiente:

1. Si se especifica el parámetro del nodo *apply_sharp* a True se aplica un filtro para agudizar los bordes de los objetos de la imagen. Posteriormente, se convierte la imagen a escala de grises.

2. Sobre la imagen en escala de grises se aplica un filtro *blur* con un tamaño de *kernel* establecido en el parámetro *blur_ksize*. Con este filtro se consigue suavizar la imagen. Su uso es optativo y se desactiva estableciendo el tamaño de *kernel* a 0.

3. Ahora se binariza la imagen aplicando el método Otsu. Además, se eliminan las marcas ArUco de la imagen binarizada.

4. Segmentación de la imagen. Para ello primero se extraen las etiquetas de cada componente encontrado mediante la función *connectedComponents*, que acepta un argumento de conectividad que se puede configurar a través del parámetro *conectivity_way*. Cada componente se corresponde con un objeto distinto. Después, se crea una imagen por cada componente existente y se asigna cada píxel de la imagen a la imagen de su componente correspondiente.

5. Se llama a la función de extracción de contornos, apartado 8.2.3, y se le pasa como argumento las imágenes de los objetos.

En la figura 8.3 se puede ver el resultado de extraer un contorno con este método.

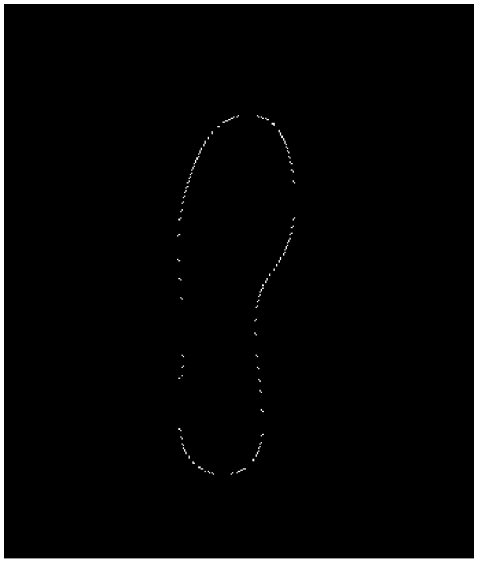


Figura 8.3: Contorno extraído de una suela de zapato, figura 8.1, utilizando el método Simple desarrollado.

Método Watershed

Este método es similar al anterior pero se diferencia en la utilización del algoritmo de watershed y en la aplicación de transformaciones morfológicas. Los pasos son los siguientes:

1. Transformación de la imagen a escala de grises y, opción de aplicar un filtro *blur*.

2. Binarización de la imagen y eliminación de las marcas ArUco.

3. Aplicación de transformaciones morfológicas, para una explicación más detallada consultar el enlace [31]:

    - *opening*: se aplica una transformación de erosión seguida de una dilatación. Es útil para eliminar ruido. El número de veces que se aplican las transformaciones se establece con el parámetro del nodo *number_iterations*, si se asigna 0 no se usa.
    - *dilate*: transformación de dilatación. Se configura mediante los parámetros: *dilate_type*, tipo de dilatación a aplicar (no aplicar, *MORPH_RECT*, *MORPH_CROSS*, *MORPH_ELLIPSE*); y *dilate_size*, a partir del cual se establecen los valores del *kernel* a usar.
    - *erode*: transformación de erosión. Consta de los mismos parámetros que *dilate*, pero sus nombres son *erode_type* y *erode_size*.

4. Calculamos la región de píxeles desconocidos a partir de las transformaciones anteriores y calculando la distancia Euclídea de cada píxel al píxel con valor 0 más cercano a través de la función *distanceTransform*. Se calculan las componentes conexas y sobre su resultado se marca con valor 0 aquellos píxeles que pertenecen a la región desconocida.

5. Aplicamos el algoritmo watershed, el cual ya viene implementado en la librería de OpenCV, y extraemos los objetos encontrados.

6. Por último, se llama a la función de extracción de contornos, apartado 8.2.3.

Los resultados obtenidos con este método no fueron buenos debido a que segmentaba demasiado la imagen, como se aprecia en la figura 8.4. Por tanto, se descartó su uso en las pruebas finales.

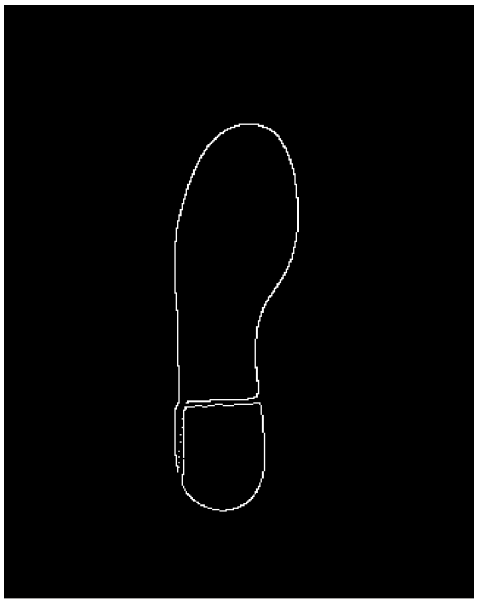


Figura 8.4: Contorno extraído de una suela de zapato, figura 8.1, utilizando el método Watershed desarrollado. Se observa que el método ha dividido el objeto en dos contornos distintos.

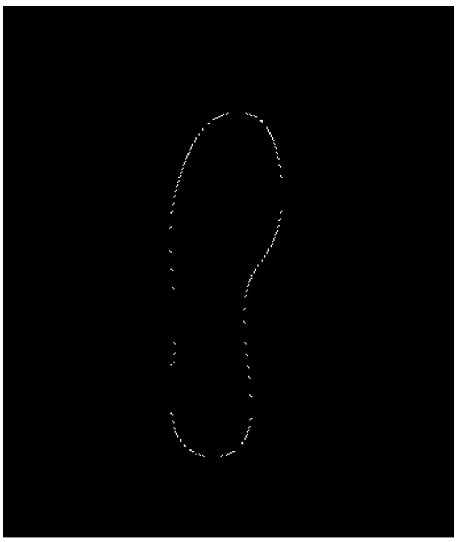
Extracción de puntos del contorno

En este paso se extraen los puntos que conforman el contorno de cada objeto obtenido de la imagen procesada. Por cada objeto resultante se ejecutan los siguientes pasos:

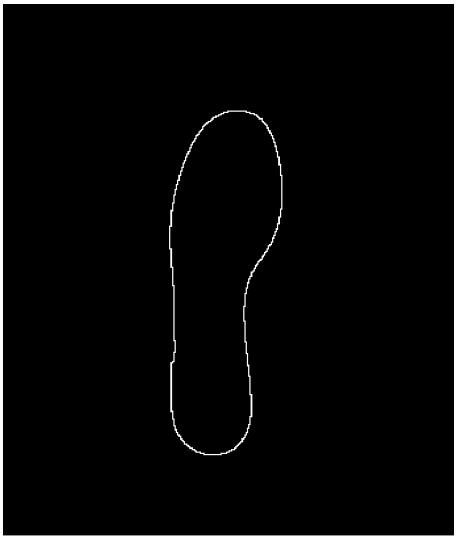
1. Búsqueda de los contornos de cada objeto mediante la función *findContours*. Dentro de esta función se ha establecido *RETR_EXTERNAL* como modo de recuperación de contornos y *CHAIN_APPROX_SIMPLE* como método de aproximación de contornos.

2. Sobre el conjunto de contornos extraídos del objeto nos quedamos con el que tenga mayor perímetro. Además, el tamaño de este contorno tiene que que ser mayor que el parámetro *max_dist_error*. Con esta restricción se consigue eliminar contornos no deseados, por ejemplo, pequeñas sombras formadas por un objeto en la mesa de trabajo. Si no hay ningún contorno que cumpla esta restricción se procede con el siguiente objeto.

3. Por último, se añaden los puntos del contorno resultante al conjunto de contornos finales a dibujar. Algunos contornos no conseguían obtener un número de puntos suficientes como para obtener un contorno cerrado y que el robot pueda realizar un dibujo preciso, como se aprecia en la figura 8.5a. Para arreglar esto, antes de añadirlos al conjunto final, se aplica la función *concaveman* [32], que es una implementación del algoritmo *concave hull*. Esta función consta de un parámetro *alpha*, el cual se puede configurar a través del parámetro *concaveman_alpha* del nodo, que determina la precisión del contorno resultante. A mayor valor de *alpha*, menor precisión del contorno y menor tiempo de cómputo. La aplicación de la función es opcional y se habilita mediante el parámetro *apply_concaveman*. En la figura 8.5 se observa la comparación entre aplicar y no aplicar *concaveman* sobre un mismo contorno.



(a) Contorno extraído de un objeto que consta de 204 puntos, insuficientes para lograr un contorno cerrado, lo cual impide la realización de un dibujo preciso por parte del robot.



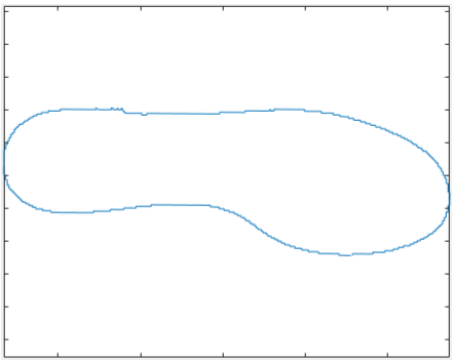
(b) Contorno extraído de un objeto al cual se le ha aplicado la función *concaveman* [32], con un *alpha* de 0.5. Al aplicar la función se consigue extraer 714 puntos y por lo tanto, un contorno cerrado.

Figura 8.5: Comparación entre un contorno extraído con y sin el uso de la función *concaveman*. El contorno se ha obtenido con el método Simple. El objeto al cual pertenece el contorno se puede observar en la figura 8.1.

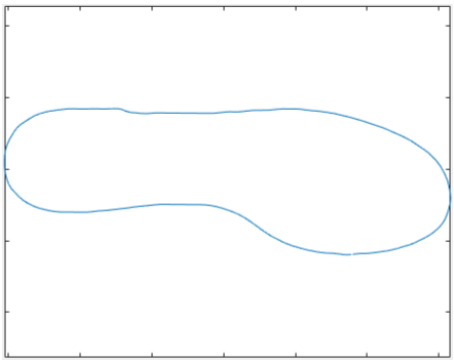
8.2.4. Transformación de contornos

Una vez conseguidos los contornos, hay que transformar los puntos que los conforman a coordenadas del mundo y, posteriormente, publicar los resultados en el *topic /image_points*. Los pasos seguidos son los siguientes:

1. Los puntos de los contornos están en coordenadas píxeles discretas, por tanto, se puede interpolar entre píxeles formas curvas con el objetivo de suavizar los contornos. La interpolación hace uso de un filtro Gaussiano, cuyo tamaño de *Kernel* se establece con el parámetro *smooth_path_kernel*. La explicación del método se detalla en el Anexo F. Su aplicación es opcional y se habilita mediante el parámetro *apply_smooth_path*. En la figura 8.6 se muestra la extracción de un contorno con y sin aplicar el suavizado.



(a) Contorno extraído de un objeto, se aprecia que el resultado obtenido presenta una trayectoria rectilínea con cambios bruscos entre puntos del contorno.



(b) Contorno extraído de un objeto al cual se le ha aplicado un filtro Gaussiano con un tamaño de *kernel* de 11. El resultado es una trayectoria suavizada donde no se aprecian cambios bruscos entre puntos consecutivos.

Figura 8.6: Comparación entre un contorno extraído con y sin el uso del suavizado de contornos, explicación en el Anexo F.

2. Los puntos resultantes del paso 1, habiendo aplicado la interpolación o no, se transforman a coordenadas de la cámara RGB. Esto se consigue, primero, calculando el rayo de la cámara al plano imagen para cada punto:

$$\begin{bmatrix} P_1^I \\ P_2^I \\ 1 \end{bmatrix} = K_{3\times3}^{-1} * \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{8.3}$$

   Sabemos que todos los puntos pertenecen al mismo plano formado por las marcas ArUco, por tanto, su distancia a la cámara RGB d es constante. La distancia se calcula como la media de la posición en el eje Z de las cuatro marcas en coordenadas de la cámara RGB. Multiplicando cada rayo obtenido por d obtenemos cada punto en el

plano formado por las marcas ArUco en referencia plano imagen:

$$\begin{bmatrix} P_x^I \\ P_y^I \\ P_z^I \end{bmatrix} = d * \begin{bmatrix} P_1^I \\ P_2^I \\ 1 \end{bmatrix} \tag{8.4}$$

Por último, se transforman los puntos resultantes en coordenadas del plano imagen P^I a coordenadas de la cámara RGB X^C. Para ello se utiliza la matriz de transformación del plano imagen a la cámara RGB ${}^CT_I$, que es la inversa de la matriz ${}^IT_C$ obtenida en el apartado 8.2.2.

$$\begin{bmatrix} X_x^C \\ X_y^C \\ X_z^C \\ 1 \end{bmatrix} = ({}^CT_{I_{4\times4}}) * \begin{bmatrix} P_x^I \\ P_y^I \\ P_z^I \\ 1 \end{bmatrix} \tag{8.5}$$

3. Ahora se transforman los puntos en coordenadas de la cámara RGB X^C a coordenadas del mundo X^W, en este caso del robot Campero. Para ello hay que realizar una transformación intermedia a coordenadas de la cámara RealSense D435 X^R. Esto se consigue mediante la siguiente cadena de transformación:

$$\begin{bmatrix} X_x^W \\ X_y^W \\ X_z^W \\ 1 \end{bmatrix} = ({}^WT_{R_{4\times4}}) * ({}^RT_{C_{4\times4}}) * \begin{bmatrix} X_x^C \\ X_y^C \\ X_z^C \\ 1 \end{bmatrix} \tag{8.6}$$

Donde ${}^RT_C$ es la matriz de transformación de la cámara RGB a la Realsense D435 y ${}^WT_R$ es la matriz de transformación de la cámara D435 al mundo. Ambas matrices de transformación se obtienen a través del *TransformListener* disponible en el paquete *tf* de ROS.

4. Por último, se crea el mensaje de tipo *ImgDraw* a partir de los puntos resultantes del paso anterior. En dicho mensaje, cada contorno extraído corresponde a un trazo del dibujo.

8.3. Resultados

Como el demostrador dispone de varios parámetros configurables a través de ROS, se desarrolló un programa que permite establecer valores para cada uno de ellos de forma sencilla. La explicación del programa está disponible en el Anexo E. La configuración de valores de los parámetros puede variar dependiendo del entorno, la cámara o los objetos a reconocer.

En este caso se utilizó la siguiente configuración:

- *max_dist_error*: 3 píxeles.
- *contour_method*: Simple.
- *conectivity_way*: *8-way*
- *blur_ksize*: 0, es decir, no se aplica el filtro Blur.
- *apply_sharp*: no se aplica el filtro.
- *min_contour_size*: 70 píxeles.
- *concaveman*: aplicar con un *alpha* de 0.5.
- *suvizado de contornos*: aplicar con un *kernel* Gaussiano de tamaño 11.

Para lanzar la aplicación basta con con ejecutar el siguiente comando:

```
roslaunch image_inpainting exec.launch
```

A continuación se presentan algunos de los resultados obtenidos, comparando el objeto real con el contorno dibujado por el robot. En la lista de reproducción [10] hay disponible un vídeo donde se pueden visualizar más resultados, así como el proceso de dibujado del robot.

El primer resultado muestra la extracción del contorno de una suela de zapato. En la figura 8.7 se observa el contorno extraído del objeto dibujado sobre la imagen rectificada. En la figura 8.8 se encuentra el dibujo realizado por el robot UR10 y su comparación con el objeto real.

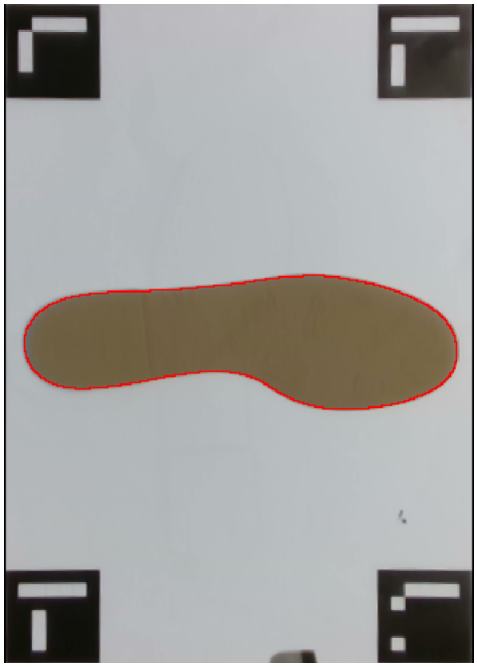
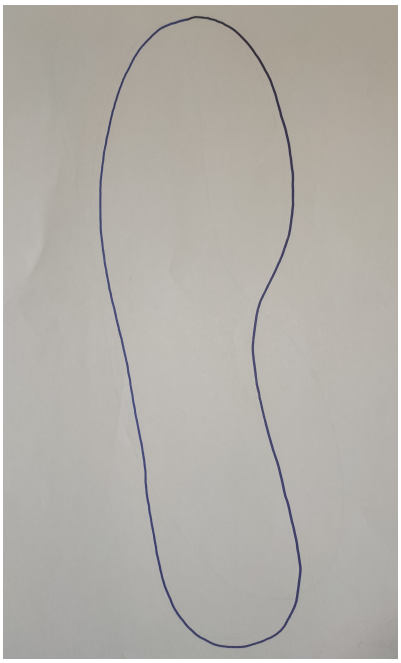


Figura 8.7: Contorno extraído de una suela de zapato dibujado, en color rojo, sobre la imagen rectificada.



(a) Dibujo del contorno extraído de la suela realizado por el robot UR10.

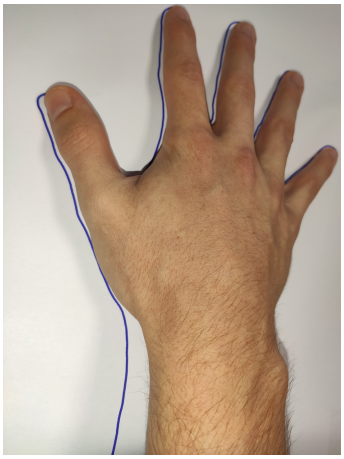
(b) Suela colocada sobre el dibujo realizado por el robot. Se aprecia que la forma del objeto y el dibujo del contorno coinciden.

Figura 8.8: Dibujo del contorno de una suela de zapato, figura 8.7 y, comparación con el contorno real.

El segundo resultado a mostrar se trata de la extracción del contorno del antebrazo de una persona. En la figura 8.9 se puede ver el contorno extraído dibujado sobre la imagen rectificada. En el contorno se aprecia que en la parte inferior izquierda hay un fragmento de sombra que no ha sido posible eliminar. En la figura 8.10 está disponible el dibujo realizado por el robot UR10 y su comparación con el contorno real.

Figura 8.9: Contorno extraído del antebrazo de una persona, dibujado, en color violeta, sobre la imagen rectificada. Se aprecia que en la parte inferior izquierda hay un fragmento de sombra incluido como parte del contorno.

(a) Dibujo del contorno del antebrazo realizado por el robot UR10.



(b) Antebrazo colocado sobre el dibujo realizado por el robot. Se aprecia que el contorno y el dibujo coinciden.

Figura 8.10: Dibujo del contorno del antebrazo de la figura 8.9 y, comparación con el contorno real.

Capítulo 9

Tercer demostrador: Teleoperación

Un robot manipulador puede trabajar de manera autónoma, pero para la realización de ciertas tareas se requiere de la intervención de un operador humano, especialmente en entornos no estructurados y dinámicos en los cuales los problemas de percepción y planificación automática son muy complejos. En este capítulo se presenta el desarrollo de una aplicación sencilla de teleoperación que demuestra que el robot es capaz de ser teleoperado.

9.1. Descripción de la aplicación

La aplicación consiste en mover el *end-effector* en coordenadas cartesianas en el eje X e Y, por tanto, queda excluido el control de la altura de actuación. Para ello se hace uso de una marca ArUco y de la cámara RGB. El operario puede mover la marca en el espacio de visión de la cámara y el *end-effector* realizará el mismo movimiento incremental que la marca, es decir, se desplazará en la misma dirección y recorriendo la misma distancia. En la figura 9.1 se muestra el espacio de trabajo del operario utilizado en las pruebas realizadas.

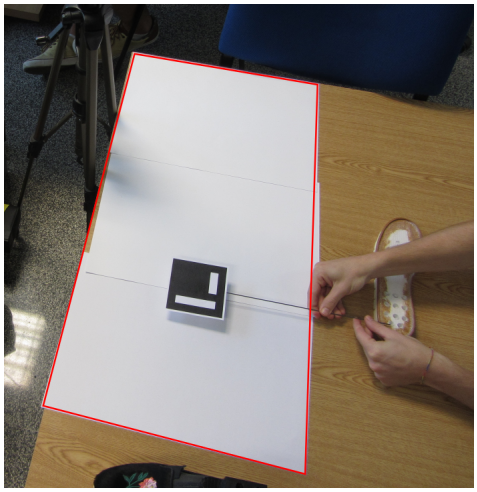


Figura 9.1: Espacio de trabajo del operario. La marca ArUco está pegada a una varilla que permite un control más preciso de la misma. Las líneas rojas delimitan el área de búsqueda de la marca ArUco por la cámara.

9.2. Desarrollo de la aplicación

El nodo encargado de la teleoperación se llama *campero_ur10_teleop_aurco*, incluido en el paquete *campero_ur10_teleop*. Para recibir la posición de la marca respecto a la cámara, el nodo se suscribe al *topic /aruco_detector/markers_pose*. Las operaciones generadas las publica en el *topic /campero_ur10_move*. El nodo tiene que conocer el identificador de la marca a utilizar, para ello consta de un parámetro llamado *id*. Una vez el nodo es iniciado, se mantiene en un bucle infinito, recibiendo posiciones de la marca y enviando operaciones, hasta que el usuario detenga el programa.

Para llevar a cabo un movimiento es necesario conocer dos posiciones de la marca: *last_pose*, última posición conocida; y *new_pose*, posición actual de la marca. Las posiciones no almacenan el valor en el eje Z. Como el valor de la posición puede variar de un instante a otro, se toma un número de muestras por cada posición y se calcula la media. El número de muestras necesarias se configura a través del parámetro *ns*, un mayor número de muestras implica una mayor precisión y un mayor tiempo de procesamiento, por lo que se pierde fluidez en la teleoperación.

Una vez se hayan obtenido las dos posiciones, es necesario realizar una serie de comprobaciones. Si la distancia entre las dos posiciones es mayor que un valor fijado (parámetro *mp*) se dice que el vector que va desde *last_pose* a *new_pose* está saturado. Para evitar movimientos excesivamente largos a causa de esta saturación se re-escala el vector, dividiendo sus componentes por su módulo, de esta forma no sobrepasa el umbral fijado. Además, con el objetivo de reducir el número de movimientos generados por el ruido en la percepción de la marca ArUco se descartan las acciones con un módulo inferior a un valor fijo, llamado *dp*.

Una vez realizadas las comprobaciones, se publica el mensaje en el *topic /campero_ur10_move*. Posteriormente, la posición *last_pose* tomará el valor de *new_pose* y, el valor de esta última se restablece.

Comando para lanzar la aplicación Teleoperación:

```
python campero_ur10_teleop_aruco [--id <aruco_id>] /
[--ns <samples>] [--dp <min_distnace>] [--mp <max_distance>]
```

9.3. Resultados

Para las pruebas se utilizó una marca ArUco con identificador ‘1’ y de tamaño *0.096 m*. El número de muestras por posición se fijó en 10. La distancia mínima en *0.001 m* y la máxima en *0.1 m*.

Al realizar las primeras pruebas, la teleoperación no funcionaba como se esperaba, puesto que el robot realizaba pocos movimientos y de muy poco recorrido. El problema provenía

de que el nodo de detección de marcas ArUco publicaba una gran cantidad de mensajes en un intervalo de tiempo muy pequeño, por tanto, la diferencia de posición entre un mensaje y otro era demasiado pequeña. La solución encontrada consistió en modificar dicho nodo para que publicase un mensaje cada 5 *frames* que recibiese de la cámara.

Al ser una aplicación sencilla de muestra, sólo se puede mover el *end-effector* en dos dimensiones. En este caso se ha usado la teleoperación para "repasar" el contorno de una suela (figura 9.2) y para doblar una suela (figura 9.3). Esto último puede ser útil, por ejemplo, para trabajos que requieran realizar pruebas de manera precisa sobre materiales deformables. Los resultados también se pueden visualizar en un vídeo disponible en la siguiente lista de reproducción [10].

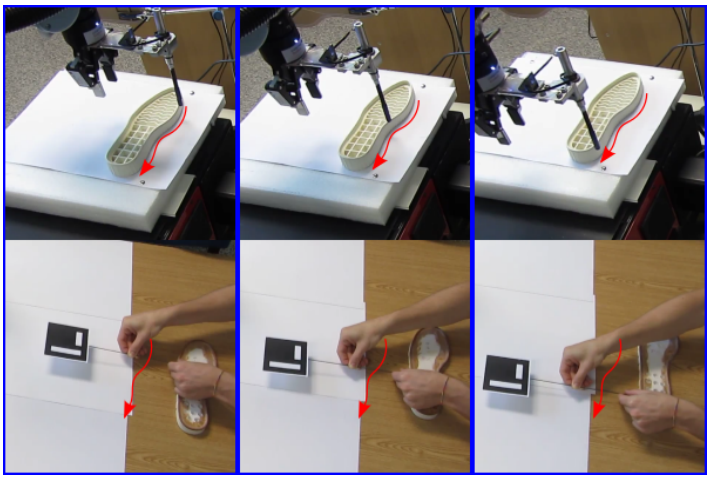


Figura 9.2: Aplicación de Teleoperación usada para "repasar" el contorno de una suela.

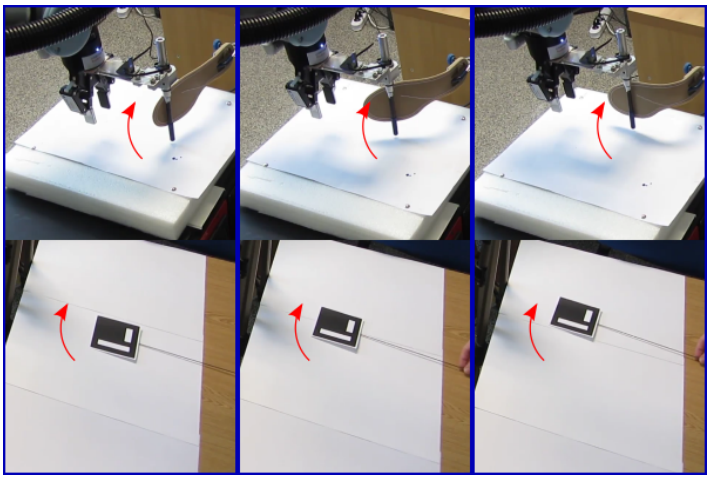


Figura 9.3: Aplicación de Teleoperación usada para doblar una suela, útil para hacer pruebas sobre materiales deformables.

Capítulo 10

Conclusiones y líneas futuras

10.1. Conclusiones

Una vez terminado el trabajo, se puede afirmar que se han cumplido todos los objetivos propuestos. Se ha logrado configurar y poner en funcionamiento el robot manipulador UR10, del robot Campero, tanto en simulación como en un entorno real. Además, se han diseñado e implementado con éxito tres aplicaciones que demuestran un correcto funcionamiento. Para llegar hasta este punto se ha tenido que pasar por distintas fases: aprendizaje del entorno ROS y los diversos programas que incluye, aprendizaje del *framework* MoveIt para el desarrollo de aplicaciones orientadas a robots manipuladores, configuración del robot UR10 junto al robot Campero para trabajar en simulación, puesta a punto del UR10 real para permitir su control con ROS y, por último, el desarrollo de diversas aplicaciones prácticas. Un punto muy importante a destacar es que la realización de un buen trabajo en simulación permite ahorrar mucho tiempo a la hora de probar lo implementado en un entorno real.

Se puede decir que la filosofía de ROS funciona, puesto que gracias a los paquetes desarrollados por la comunidad se consigue ahorrar tiempo de trabajo. Por otra parte, se podría reprochar que la documentación sobre ROS está muy dispersa y, en muchas ocasiones es muy costoso encontrar información sobre un tema en particular. Siguiendo con la filosofía de ROS, todo el trabajo desarrollado se encuentra disponible de forma pública en un repositorio de Github (Anexo G).

10.2. Líneas futuras

Como ya se habló en la Introducción 1, este trabajo ha sido el primero en utilizar el robot UR10 junto con ROS en el Departamento de Informática e Ingeniería de Sistemas. Por tanto, con este trabajo se ha recopilado toda la información necesaria para poder utilizar el robot, así como, diversos tutoriales detallados en los anexos. Todo esto, con el objetivo de que sirva como fuente de referencia para futuros proyectos.

Los demostradores implementados son sencillos y tienen margen de mejora, puesto que su objetivo era demostrar el correcto funcionamiento del robot en diferentes tipos de tareas. Además, no se ha hecho uso ni de la pinza ni del sensor de torque y presión, por tanto, sería interesante la utilización de estos elementos para tareas de manipulación en proyectos futuros. Se podría combinar también con percepción del entorno en tiempo real mediante el uso de sensores y/o cámaras o permitir su teleoperación. Otro proyecto interesante sería la utilización del robot UR10 junto con el robot móvil Campero en tareas más complejas, por ejemplo, el transporte de objetos.

Capítulo 11

Bibliografía

[1] UR10 Especificaciones Técnicas. https://www.universal-robots.com/media/1801318/esp_semea_199912_ur10_tech_spec_web_a4.pdf. [Online; accedido el 16 de Junio de 2021].

[2] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[3] Rafael Muñoz-Salinas Rafael Medina-Carnicer Sergio Garrido-Jurado, Francisco Madrid-Cuevas. ArUco: a minimal library for Augmented Reality applications based on OpenCV. https://www.uco.es/investiga/grupos/ava/node/26. [Online; accedido el 23 de Julio de 2021].

[4] COMMANDIA Universidad de Zaragoza. http://commandia.unizar.es.

[5] Intel® RealSense™ Depth Camera D435. https://www.intelrealsense.com/depth-camera-d435/. [Online; accedido el 16 de Junio de 2021].

[6] MoveIt! Motion Planning Framework. https://moveit.ros.org/. [Online; accedido el 16 de Junio de 2021].

[7] Itseez. Open Source Computer Vision Library. https://github.com/itseez/opencv, 2015. [Online; accedido en julio de 2019].

[8] Intel. ROS Wrapper for Intel® RealSense™ Devices . https://github.com/IntelRealSense/realsense-ros, 2019.

[9] PAL Robotics. ArUco ROS Wrapper. https://github.com/pal-robotics/aruco_ros. [Online; accedido el 16 de Junio de 2021].

[10] Lista de vídeos de resultados del trabajo. https://youtube.com/playlist?list=PL8nq2oaZdXaHJQJ9toVDRm7ssh3V488Ac. [Online; accedido el 31 de Julio de 2021].

[11] Keyence America. PL (Performance level). https://www.keyence.com/ss/products/safetyknowledge/performance/level/. [Online; accedido el 16 de Junio de 2021].

[12] Leon Jung Darby Lim Yoonseok Pyo, Hancheol Cho. *ROS Robot Programming (English)*. ROBOTIS, 12 2017.

[13] ROS Tutorials. http://wiki.ros.org/ROS/Tutorials. [Online; accedido el 16 de Junio de 2021].

[14] Jackie Kay Ioan Sucan. URDF Github. https://github.com/ros/urdf. [Online; accedido el 16 de Junio de 2021].

[15] Robert Haschke Stuart Glaser, William Woodall. Xacro Github. https://github.com/ros/xacro. [Online; accedido el 16 de Junio de 2021].

[16] URDF Tutorials. http://wiki.ros.org/urdf. [Online; accedido el 16 de Junio de 2021].

[17] Xacro Tutorials. http://wiki.ros.org/xacro. [Online; accedido el 16 de Junio de 2021].

[18] MoveIt! Motion Planning Framework Install. https://moveit.ros.org/install/. [Online; accedido el 16 de Junio de 2021].

[19] MoveIt! Kinetic Tutorials. http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/index.html. [Online; accedido el 16 de Junio de 2021].

[20] MoveIt! Concepts. https://moveit.ros.org/documentation/concepts/. [Online; accedido el 16 de Junio de 2021].

[21] Universal Robot ROS Driver. https://github.com/UniversalRobots/Universal_Robots_ROS_Driver. [Online; accedido el 16 de Junio de 2021].

[22] MoveIt! Setup Assistant Tutorial. http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html. [Online; accedido el 23 de Julio de 2021].

[23] Polyscope Install Urcap. https://github.com/UniversalRobots/Universal_Robots_ROS_Driver/blob/master/ur_robot_driver/doc/install_urcap_cb3.md. [Online; accedido el 16 de Junio de 2021].

[24] Intel Open Source Technology Center. Hand-eye calibration tools for robot arms. https://moveit.ros.org/moveit/ros/2020/08/26/moveit-calibration.html. [Online; accedido el 22 de Julio de 2021].

[25] Mensaje con Información de la Cámara. http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/CameraInfo.html. [Online; accedido el 25 de Julio de 2021].

[26] Mensaje de tipo Imagen. http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/Image.html. [Online; accedido el 25 de Julio de 2021].

[27] Aspect Ratio of an Image. https://en.wikipedia.org/wiki/Aspect_ratio_(image). [Online; accedido el 27 de Julio de 2021].

[28] Python Turtle Graphics. https://docs.python.org/3/library/turtle.html. [Online; accedido el 31 de Julio de 2021].

[29] Función solvePnp para la obtención de los vectores de rotación y traslación. https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga549c2075fac14829ff4a58bc931c033d. [Online; accedido el 31 de Julio de 2021].

[30] F. Meyer. Color image segmentation. In *1992 International Conference on Image Processing and its Applications*, pages 303–306, 1992.

[31] Transformaciones morfológicas. https://docs.opencv.org/4.5.2/d9/d61/tutorial_py_morphological_ops.html. [Online; accedido el 31 de Julio de 2021].

[32] Stanislaw Adaszewski. concaveman-cpp a very fast 2D concave hull implementation. https://github.com/sadaszewski/concaveman-cpp. [Online; accedido el 31 de Julio de 2021].

[33] Technical specifications ur10. https://www.universal-robots.com/media/50880/ur10_bz.pdf. [Online; accedido el 16 de Junio de 2021].

[34] Función obtención kernel Gaussiano. https://docs.opencv.org/4.5.2/d4/d86/group__imgproc__filter.html#

gac05a120c1ae92a6060dd0db190a61afa. [Online; accedido el 31 de Julio de 2021].

[35] Ioan A Sucan. Ejemplos escenas de MoveIt. https://github.com/isucan/plannerarena/tree/master/problems/pr2_scenes. [Online; accedido el 22 de Julio de 2021].

[36] MoveIt. Publish Scene Node. https://github.com/ros-planning/moveit/blob/master/moveit_ros/planning/planning_components_tools/src/publish_scene_from_text.cpp. [Online; accedido el 22 de Julio de 2021].

[37] MoveIt. Function Load Scene File. http://docs.ros.org/en/jade/api/moveit_core/html/planning__scene_8cpp_source.html#l01015. [Online; accedido el 22 de Julio de 2021].

[38] Ignacio Herrera Seara. Repositorio Github del trabajo. https://github.com/nachoh8/CamperoUR10-TFG. [Online; accedido el 15 de Agosto de 2021].

Anexos

Anexos A

Escenas en MoveIt

Para definir escenas en MoveIt hay que utilizar el formato de archivo '.scene'. Sin embargo, la documentación sobre dicho formato no se encuentra disponible, debido a que la página web que lo alojaba ha sido eliminada. Por tanto, tuve que investigar, y mediante ejemplos sacados de Github [35] y analizando el código fuente del programa que carga las escenas [36] [37] logré sacar el formato del fichero.

En la figura A.1 se puede ver el formato '.scene'. El archivo comienza con un nombre de la escena seguido de uno o mas objetos, uno debajo de otro. Cada objeto empieza por el carácter '*' seguido de un nombre. El archivo termina con el carácter '.'. Además cada objeto se compone de una o más formas. El formato de las formas se puede ver en la figura A.2. Las formas empiezan con el nombre del tipo, seguido de una fila con una serie de valores que dependen del tipo y terminan con la posición (en metros), orientación (en cuaternión) y color (en formato RGBa, con valores del 0 al 1).

```
1  <nombre_escena>
2  * <nombre_objeto> # declaracion de un objeto
3  <numero_de_formas> # int >= 1
4  <forma> # lista de formas, una debajo de otra
5  . # final del fichero
```

Figura A.1: Formato del archivo '.scene'.

```
box
<x> <y> <z> # size (meters): double
<x> <y> <z> # position (meters): double
<rx> <ry> <rz> <rw> # orientation (quaternions): double
<r> <g> <b> <a> # color in RGBA format: float in range [0,1]

cylinder
<r> <l> # radius and length (meters): double
<x> <y> <z> # position (meters): double
<rx> <ry> <rz> <rw> # orientation (quaternions): double
<r> <g> <b> <a> # color in RGBA format: float in range [0,1]

cone
<r> <l> # radius and length (meters): double
<x> <y> <z> # position (meters): double
<rx> <ry> <rz> <rw> # orientation (quaternions): double
<r> <g> <b> <a> # color in RGBA format: float in range [0,1]

sphere
<r> # radius (meters): double
<x> <y> <z> # position (meters): double
<rx> <ry> <rz> <rw> # orientation (quaternions): double
<r> <g> <b> <a> # color in RGBA format: float in range [0,1]

mesh
<v> <t> # number of vertex and triangles : int >= 0
<x> <y> <z> # position (meters): double
<rx> <ry> <rz> <rw> # orientation (quaternions): double
<r> <g> <b> <a> # color in RGBA format: float in range [0,1]

plane
<a> <b> <c> <d> # ecuation of a plane -> ax + by + cz + d = 0: double
<x> <y> <z> # position (meters): double
<rx> <ry> <rz> <rw> # orientation (quaternions): double
<r> <g> <b> <a> # color in RGBA format: float in range [0,1]
```

Figura A.2: Formato de las formas de los archivos '.scene'.

Hay tres formas de cargar una escena en MoveIt:

- **código:** creando la escena a través de código, mediante la clase *PlanningScene* de MoveIt.
- **nodo:** lanzando el nodo *publish_scene_from_text* [36] y pasándole como argumento un fichero '.scene'.
- **Rviz:** mediante el botón "Import From Text" de la pestaña "Scene Objects" del menú de MoveIt en Rviz.

También se han encontrado con diversos problemas a la hora de cargar las escenas:

- Las formas de tipo plano siempre son transparentes, independientemente del color asignado.
- Cargando la escena desde el nodo o generando la escena por código, los colores establecidos no los tiene en cuenta y salen siempre verdes.
- La escena no carga si un objeto tiene más de una forma de tipo plano.

- Si un objeto tiene más de una forma, que no sea un plano, a veces ocurren diversos errores.

Por los problemas comentados anteriormente se recomienda definir una única forma por cada objeto.

En la figura A.3 se ve la escena definida para el este trabajo. Está compuesta por cinco planos que forman la pantalla protectora y una caja representando la mesa de trabajo.

```
scene
* p_right
1
plane
1 0 0 0
0.67 0 0
0 0 0 1
0.1 0.1 1 1
* p_left
1
plane
1 0 0 0
-0.67 0 0
0 0 0 1
0.1 0.1 1 1
* p_up
1
plane
0 0 1 0
0 0 2
0 0 0 1
0.1 0.1 1 1
* p_front
1
plane
0 1 0 0
0 0.5 0
0 0 0 1
0.1 0.1 1 1
* p_back
1
plane
0 1 0 0
0 -0.5 0
0 0 0 1
0.1 0.1 1 1
* o_box
1
box
0.5 0.5 0.05
-0.39 0 0.745
0 0 0 1
1 0 0 1
.
```

Figura A.3: Archivo *ws_walls.scene*, que define el espacio de trabajo del robot en MoveIt. El objeto “o_box” representa la mesa de trabajo y el resto de objetos, que son planos, forman la pantalla protectora alrededor del robot Campero.

Anexos B

Programa de detención del robot

Este programa, *stop_robot.py*, permite detener el movimiento actual del robot en cualquier momento mediante la interacción del usuario por terminal. El programa corre en un bucle infinito esperando la interacción del usuario. Cuando el usuario presiona la tecla 'Enter', el programa envía a MoveIt la operación de detención inmediata, si pulsa 'z' el programa termina. El programa se ejecuta con el siguiente comando:

```
python stop_robot.py
```

El código del programa desarrollado es el siguiente:

```
1  import sys
2  from moveit_commander import move_group
3  import rospy
4  import moveit_commander
5
6  NODE_NAME = "stop_campero_ur10"
7  UR10_PLANNING_GROUP = "manipulator"
8
9  TIME_SLEEP = 0.1
10
11 move_group = None
12
13 # stop current robot execution
14 def stop():
15     global move_group
16
17     rospy.loginfo("Stopping robot")
18     move_group.stop()
19
20 def main():
21     global move_group
22
23     moveit_commander.roscpp_initialize(sys.argv)
24     rospy.init_node(NODE_NAME, anonymous=True)
25
```

```
    move_group =
    moveit_commander.MoveGroupCommander(UR10_PLANNING_GROUP)

    rospy.loginfo("---Stop Program Ready---")

    finish = False

    while not rospy.is_shutdown() and not finish:
        s = raw_input(
"---Press Enter to Stop the Robot or z to end program---")

        if s == "":
            stop()
        elif s == "z":
            finish = True

        rospy.sleep(TIME_SLEEP)

    rospy.loginfo("End program")


if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

Anexos C

Archivo de configuración de dibujar

En la figura C.1 se ve el archivo de configuración de dibujar, en formato de texto sencillo, del nodo "Servidor" (apartado 6.3). Los parámetros se indican con el carácter '*' seguido de un nombre sin espacios, la línea siguiente contiene el valor de dicho parámetro. Este archivo de configuración consta de 10 parámetros de configuración: altura del *end-effector* para pintar y levantar el rotulador, corrección de posición de la punta del rotulador respecto al *end-effector* y posición en el mundo y tamaño en metros del lienzo.

```
# posicion en el eje Z del end-effector cuando el rotulador esta sobre el papel
*Z_PEN_DOWN
1.048

# posicion en el eje Z del end-effector cuando el rotulador esta levantado
*Z_PEN_UP
1.2

# diferencia de posicion en el eje Y entre el end-effector y la punta del rotulador
*CORRECT_Y
-0.129

# diferencia de posicion en el eje X entre el end-effector y la punta del rotulador
*CORRECT_X
0.0435

# diferencia de posicion en el eje Z entre el end-effector y la punta del rotulador
*CORRECT_Z
0.264

# posicion(esquina con menor x e y) y tamaño del lienzo
*BOARD_X
-0.47

*BOARD_Y
-0.21

*BOARD_Z
0.785

*W_BOARD
0.42

*H_BOARD
0.297
```

Figura C.1: Archivo de configuración de dibujar del "Servidor".

Anexos D

Detector marcas ArUco

En este trabajo se ha utilizado el paquete *aruco_ros* [9] para la detección de marcas ArUco, que cuenta con los programas necesarios para el procesamiento de imágenes y extracción de la información de las marcas. También dispone de una serie de nodos que publican en *topics* la posición de las marcas respecto a la cámara.

Como se explicó en el apartado de mensajes de las marcas ArUco 6.2.2, la aplicación "Dibujo Contornos" requiere de más información que la proporcionada por los nodos del paquete *aruco_ros*. Por tanto, ha sido necesario la creación de un nuevo nodo, *aruco_detector*, dentro de este paquete que se adecue a las necesidades de los demostradores. El cual ha sido desarrollado a partir del código del nodo *simple_single* del mismo paquete.

No hubo que realizar muchos cambios al código del nodo *simple_single*. El nodo requiere de la información relativa al sensor de la cámara RGB, la cual se consigue con el *topic /camera/color/camera_info*; y la obtención de las imágenes de dicha cámara, a través del *topic /camera/color/image_rect_color*. La extracción de los puntos de las esquinas de las marcas en la imagen se consigue mediante una función ya implementada en el paquete. Posteriormente, hay que cambiar el tipo de mensajes que envía y los *topics* donde publica la información extraída.

El nodo *aruco_detector* publica la posición de las marcas junto con los puntos de sus esquinas en el *topic /aruco_detector/markers_pose*, que contiene mensajes de tipo *ArucoMarkerArray*; y la información anterior junto con la imagen recibida en el *topic /aruco_detector/markers_img*, donde se publican mensajes de tipo *ArucoMarkesImg*.

El comando para lanzar el nodo es el siguiente:

```
roslaunch aruco_ros aruco_detector.launch
```

En la figura D.1 se puede ver dibujado el centro y las esquinas de cada marca en la imagen capturada por la cámara. En la figura D.2 se observan las marcas ArUco vistas en la nube de puntos publicada por la cámara, además de ver una referencia al centro de cada marca.

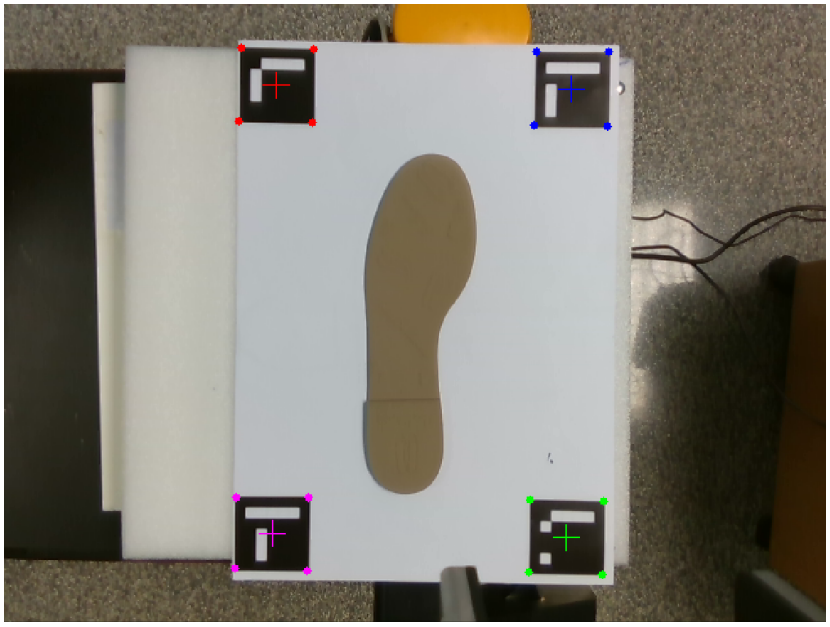


Figura D.1: Marcas ArUco en la imagen capturada por la cámara. Las cruces representan el punto central de cada marca en la imagen y, los círculos los puntos de las esquinas de cada marca en la imagen. Los puntos han sido extraídos a partir del nodo *aruco_detector*.

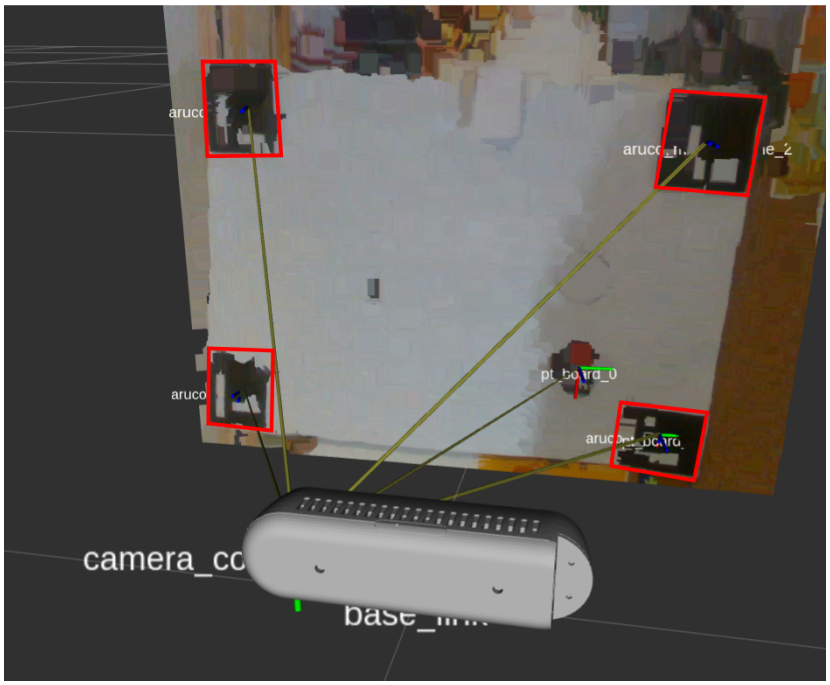


Figura D.2: Marcas ArUco vistas en la nube de puntos de la cámara. Las marcas se encuentran marcadas por un recuadro rojo. Del eje óptico de la cámara parten cuatro líneas al centro de cada una de las marcas. El centro se corresponde con la posición de la marca respecto a la cámara, conseguida gracias al nodo *aruco_detector*.

Anexos E

Configuración parámetros segundo demostrador

El segundo demostrador implementado, capítulo 8, cuenta con una serie de parámetros para el procesamiento de imágenes y extracción de contornos. Con el objetivo de facilitar pruebas con el uso de distintas configuraciones de los parámetros se desarrolló un programa que proporciona una interfaz de usuario que permite establecer valores para cada parámetro de manera sencilla. Este programa, llamado *image_inpainting_gui.py* y disponible en el paquete *image_inpainting*, ha sido implementado mediante la librería OpenCV para Python.

Los parámetros se configuran a través del servidor de parámetros de ROS. Cada parámetro consta de un *trackbar* en la interfaz, que permite establecer el valor deseado. Los *trackbar* solo pueden tomar valores enteros y constan de un valor mínimo y máximo. Hay parámetros cuyos valores no son de tipo entero, por tanto, es necesario realizar una conversión de tipos antes de actualizar su valor en el servidor. Por ejemplo, el parámetro *concaveman_alpha* es de tipo double y antes de enviar el valor devuelto por el *trackbar*, éste se divide entre 100.

Además, el demostrador puede recibir comandos a través del *topic /image_inpainting/cmd*. Los comandos disponibles son los siguientes:

- 'send': el demostrador publica los contornos que almacena en el *topic /image_points*.
- 'reset': el demostrador elimina los contornos y la imagen que almacena como resultados.
- 'update_on'/'update_off': habilita/deshabilita el procesamiento de nuevas imágenes en el demostrador.
- 'update_image': el demostrador actualiza el valor de sus parámetros locales con la configuración del servidor y, vuelve a procesar la imagen almacenada, si la hay, para extraer de nuevo los contornos.

Cada uno de estos comandos dispone de un botón en la interfaz de usuario, a excepción de 'update_on/off', para los cuales solo hay un botón que funciona a modo de interruptor.

En la figura E.1 se puede ver la interfaz desarrollada para el control de los valores de los parámetros y los botones para enviar los comandos. Para ejecutar el programa hay que utilizar el siguiente comando:

```
python image_inpainting_gui.py
```

Figura E.1: Interfaz de usuario del programa *image_inpainting_gui* que permite configurar los valores de los parámetros del segundo demostrador (capítulo 8) a través de *trackbars*. También dispone de cuatro botones para enviar comandos al demostrador.

Anexos F

Suavizado de contornos

Para el suavizado de contornos, aplicado en el segundo demostrador, se desarrolló la función *smoothPath*. Toma como argumentos un vector para la coordenada X, denominado *vx*, y otro para la coordenada Y, llamado *vy*, de los puntos del contorno. La función devuelve el resultado de aplicar un filtro Gaussiano a cada vector por separado.

El tamaño del kernel del filtro se especifica con el parámetro *smooth_path_kernel*, cuyo valor debe ser un número impar entero positivo. Cuanto mayor sea el tamaño, mayor será el suavizado del contorno.

Mediante la función *getGaussianKernel* de OpenCV [34] se obtienen los coeficientes del filtro Gaussiano. Se le pasa como argumentos: el tamaño de *kernel* y sigma, correspondiente a la desviación normal. El valor de sigma se establece mediante una relación con el tamaño del *kernel*:

$$\sigma = \frac{kernel\_size}{\sqrt{2 * \pi}} \tag{F.1}$$

Para evitar que se difuminen los bordes del contorno se encapsulan *vx* y *vy* sobre sí mismos:

$$\begin{aligned} vxExt &= [vx, vx, vx] \\ vyExt &= [vy, vy, vy] \end{aligned} \tag{F.2}$$

El *kernel* se aplica sobre los nuevos vectores *vxExt* y *vyExt* y se obtiene como resultado dos vectores del mismo tamaño. Como el tamaño de estos vectores es superior a los originales, *vx* y *vy*, se elimina de cada uno los *n* primeros elementos y los *n* últimos, siendo *n* el tamaño de los vectores originales.

Además, los valores de los vectores obtenidos están a una escala distinta que los originales, por tanto, hay que multiplicarlos por un factor de escala f:

$$f = \frac{|\max_i vx_i|}{|\max_i vxExt_i|} \tag{F.3}$$

Los resultados obtenidos se pueden apreciar en la figura F.1, en la que se muestra una comparación aplicando el suavizado y sin aplicar sobre el mismo contorno.

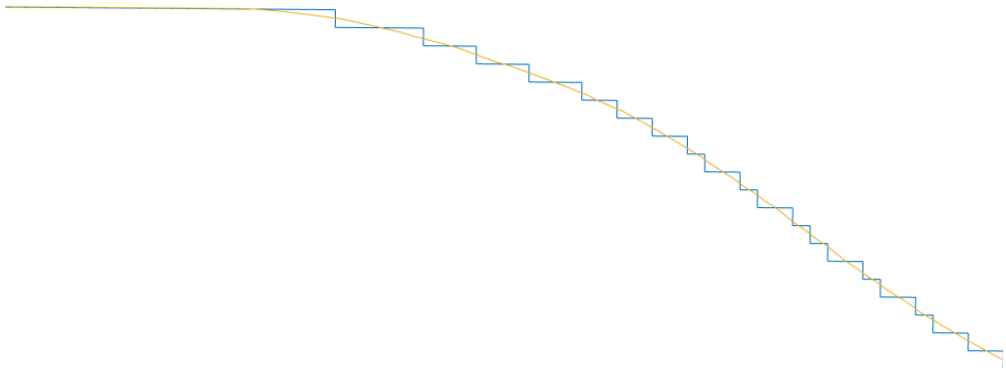


Figura F.1: Diferencia entre aplicar o no la función *smoothPath* sobre el mismo contorno, estableciendo el tamaño de *kernel* en 11. El trazo azul se corresponde con el contorno sin suavizar y el amarillo con el contorno suavizado.

Anexos G

Repositorio de Github del trabajo

Se ha utilizado la plataforma Github para llevar un control de versiones sobre el trabajo realizado. El repositorio creado [38] se encuentra disponible de forma pública, siguiendo la filosofía de ROS, para que cualquier persona pueda consultarlo y descargarlo.

El repositorio (figura G.1) consta de dos carpetas principales: *src/*, contiene los paquetes propios desarrollados para este trabajo; y *test/*, contiene archivos *rosbag* para permitir hacer pruebas con el segundo demostrador sin la necesidad de tener la cámara Realsense D435. Además, el archivo *README.md* explica como configurar el proyecto y cómo ejecutar cada paquete.

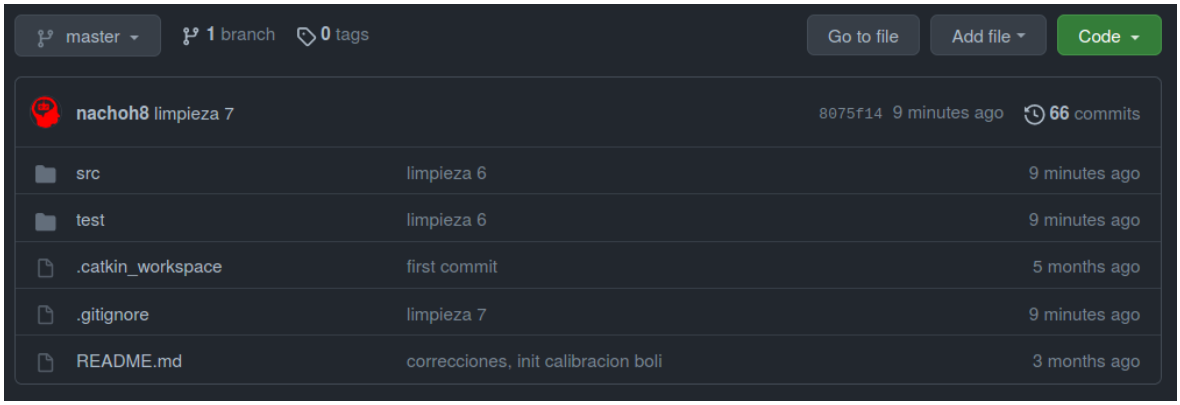


Figura G.1: Repositorio de Github.

Lista de Figuras