UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIEROS INDUSTRIALES VALENCIA

TRABAJO FIN DE MASTER EN INGENIERÍA INDUSTRIAL

DESARROLLO DE APLICACIONES INDUSTRIALES CON ROBOTS COLABORATIVOS UTILIZANDO EL MIDDLEWARE DE CONTROL DE ROBOTS ROBOT OPERATING SYSTEM

AUTOR: CLAUDIA ANAÏS GONZÁLEZ MORENO

TUTOR: ÁNGEL VALERA FERNÁNDEZ

COTUTORA: MARINA VALLÉS MIQUEL

Curso Académico: 2017-18

RESUMEN

Este proyecto consiste en la realización de una aplicación para un robot colaborativo mediante la utilización del *software* de control *Robot Operating System (ROS)*. El presente documento cuenta con diversas partes: en primer lugar, se hace una introducción teórica en la que se explica en qué consiste la robótica colaborativa. Posteriormente, se comentan los distintos métodos de programación para el robot *UR3* que se va a utilizar con el fin de realizar una comparativa y justificar la utilización de *ROS*. A continuación, se entra en detalles sobre *MoveIt!*, el complemento de *ROS* en concreto que se utilizará para la creación de las aplicaciones que forman este proyecto, su funcionamiento y los elementos que lo conforman. Después, se explican a nivel teórico cómo funcionan los modelos en *ROS* y, por último, se explican brevemente los conceptos de visión artificial utilizados dentro de las aplicaciones creadas.
En segundo lugar, se realiza el desarrollo práctico del proyecto. Éste empieza con un desglose del material que se ha utilizado, y los distintos paquetes de *ROS* que ya existían y que se han utilizado. Posteriormente se entra a explicar las modificaciones que se tuvieron que llevar a cabo en el modelo para que se correspondiera con el robot real, y ya se entra en lo que son las aplicaciones realizadas: la primera sería la que define el entorno, después se entra en los programas de manejo de la cámara y la definición del objeto en movimiento, y, por último, los programas para configurar el movimiento del robot que permiten la evasión de los obstáculos definidos.
También se añaden unas conclusiones para cerrar el documento y se adjunta una bibliografía con las fuentes de la información utilizadas.
Por último, se incorpora un manual de instalación y recomendaciones de uso de las distintas aplicaciones creadas.

Palabras Clave: robot colaborativo, *ROS, UR3*, aplicación, *software,* evasión de obstáculos, visión artificial, programa.

RESUM

Aquest projecte consisteix en la realització d'una aplicació per a un robot col·laboratiu mitjançant la utilització del *software* de control *Robot Operating System (ROS)*. El present document consisteix de diverses parts: primerament, es fa una introducció teórica en la que s'explica en què consisteix la robòtica col·laborativa. Posteriorment, es comenten els distints mètodes de programació per al robot *UR3* que es va a utilizar amb el fi de realitzar una comparativa i justificar la utilització de *ROS*. A continuació, se entra en detalls sobre *MoveIt!*, el complement de *ROS* que es va a utilizar per al desenvolupament de les aplicacions que formen aquest projecte, el seu funcionament i els elements que el conformen. Després, s'expliquen a nivell teòric com funcionen els models dins de *ROS*, i, per acabar, s'expliquen breument els conceptes de visió artificial utilitzats dins de les aplicacions creades.

En segon lloc, es realitza el desenvolupament pràctic del projecte. Comença en un desglossament del material emprat, i els distints paquets de *ROS* que existien previament i que s'han utilitzat. Posteriorment s'entra a explicar les modificacions que s'hagueren de fer al model per adaptar-ho al robot real, i ja s'entra en les aplicacions creades: la primer d'elles seria la que definix l'entorn, després es parla dels programes de utilització de la càmera i la definició del objecte que es mou, i, per últim, els programes per a configurar el moviment del robot que permeten l'evació dels obstacles definits.

També s'afegixen unes conclusions per a tancar el document i s'adjunta una bibliografía en les fonts d'informació utilitzades.

Per últim, s'incorpora un manual d'instal·lació i recomendacións d'us de les distintes aplicacions creades.

Paraules clau: robot col·laboratiu, *ROS, UR3*, aplicació, *software,* evasió d'obstacles , visió artificial, programa.

ABSTRACT

This project consists of the completion of an application for a collaborative robot using the control software *Robot Operating System (ROS)*. The present document consists of different parts: firstly, a theoretical introduction is made. This explains what collaborative robots are and what can be done with them. After this, there is an explanation of the different programming methods that can be used on the robot *UR3* that is used for this project. This is done so the methods can be compared and the utilization of *ROS* is justified. Then, *MoveIt!,* the *ROS* complement that is used for the applications created for this project, is explained alongside its elements and how it works. Afterwards, there is a theoretical explanation on how *ROS* models work, and finally, the artificial vision concepts used for the created applications are discussed.

Secondly, a practical report of the project is given. This begins with a list of the materials used and the different *ROS* packages that already existed and were used. Then, the modifications made to the original model, in order for it to look like the real model, are discussed, and after that, the different programs that were created are explained. These are; the program that allows you to define the robot's environment, the camera programs, and the app that creates the moving object in the environment, and lastly, the programs that allow the configuration of the movement of the robot and allow for obstacle avoidance.

Also, conclusions to close the document and a bibliography that contains all the sources used are added.

To finish with this document, there is an installation manual and some recommendations in order to use the different created programs.

Keywords: collaborative robot, *ROS, UR3*, application, *software,* obstacle avoidance , artificial vision, program.

ÍNDICE

DOCUMENTOS CONTENIDOS EN EL TFM

- Memoria
- Presupuesto

ÍNDICE DE LA MEMORIA

ÍNDICE DEL PRESUPUESTO

ÍNDICE DE FIGURAS

DOCUMENTO DE LA MEMORIA

CAPÍTULO 1. INTRODUCCIÓN

1.1 INTRODUCCIÓN Y MOTIVACIÓN.

La robótica colaborativa se encuentra a la orden del día, se trata de uno de los avances más curiosos de los últimos años a nivel industrial puesto que permite que los robots y las personas trabajen compartiendo espacio de trabajo. Estos robots se mueven a velocidades y fuerzas que permiten que en caso de colisión no se produzcan heridos. Sin embargo, en muchos casos, el robot se detiene cuando esta colisión ya se ha producido. Resulta, por tanto, interesante desarrollar alguna forma de informar al robot sobre su entorno para que pueda adaptar los movimientos al mismo esquivando elementos e incluso personas si fuera posible, incluso evitando tener que adquirir e instalar costosos sensores al robot.

Es el objetivo general de este trabajo, por tanto, utilizando las herramientas que provee el *software* para la programación y el control de robots *ROS (Robot Operating System)* se busca poder definir el entorno del robot, además de intentar facilitar una programación sencilla e intuitiva para los operarios que puedan utilizar el robot.

1.2 OBJETIVOS QUE BUSCA CUMPLIR ESTE TRABAJO.

Como se ha comentado al final del apartado anterior, el principal objetivo es desarrollar un programa que permita definir el entorno de trabajo de un robot colaborativo para que, en el momento de realizar un movimiento, lo tenga en cuenta y pueda esquivar los distintos elementos que lo rodean. De forma adicional, se busca que esta aplicación pueda ser utilizado por personas que no estén familiarizadas excesivamente con la programación de robots, y menos utilizando el *software* de *ROS* que requiere de muchas horas de uso para su completa familiarización. Así pues, se procederá a desarrollar un programa que permita la programación manual de forma sencilla e intuitiva.

Para poder alcanzar estos objetivos generales se establecen los siguientes objetivos secundarios, cuyo cumplimiento supondrá que se han alcanzado los objetivos finales de este proyecto:

- Familiarización con el entorno de trabajo de *Ubuntu* y con los distintos conceptos de *ROS*, sus elementos y su forma de programación (lenguaje *C++).* Esto supone uno de los primeros objetivos que ha de cumplirse ya que, si no se conoce la forma de trabajo de este *software*, resultará imposible el cumplimiento de los demás objetivos.
- Estudio del funcionamiento del paquete de control de trayectorias de robots que se encuentra disponible en *ROS*: *MoveIt!*, de su arquitectura interna y los elementos que lo componen.
- Comprensión de cómo funcionan la construcción de modelos de robots dentro de *ROS* ya que el robot que se va a utilizar existe, pero tiene ciertas peculiaridades que han de añadirse.
- Generación del nuevo modelo del robot.

- Generación del nuevo paquete de configuración para *MoveIt!* utilizando el nuevo modelo de robot creado.
- Creación de un conjunto de programas (paquete) que generen la información del entorno, y que contenga dos nodos, uno que defina completamente el entorno fijo del robot, y otro que defina posibles objetos en movimiento.
- Producción de un paquete que permita la configuración de una cámara y el seguimiento de objetos con características determinadas a partir de este.
- Conexión entre programas que permita que los movimientos del objeto se correspondan con los del elemento que se siga con la cámara.
- Generación de una serie de programas que permitan de forma sencilla guardar las posiciones del robot, para después poder ejecutarlas de forma cíclica.

Por último, a la hora de realizar el documento se realizarán las siguientes consideraciones:

- El entorno del robot es fijo, es decir, los elementos que rodean al robot no cambiarán ni realizarán movimientos.
- A la hora de crear el programa del objeto que se mueve, por simplicidad, el objeto tendrá unas dimensiones fijas y la única información que se recibirá desde la cámara es la posición de su centro geométrico.

CAPÍTULO 2. DESARROLLO TEÓRICO.

2.1 ROBOTS COLABORATIVOS.

Los robots colaborativos o *cobots* son unos robots especialmente diseñados para trabajar compartiendo espacio con los humanos e incluso realizando tareas que requieren de la interacción con ellos. Se crearon en 1996 por profesores de la Northwestern University en Illinois, EEUU. La empresa *KUKA* en una colaboración con el Centro Aerospacial Alemán desarrolló el primer robot colaborativo de uso industrial, el *LBR 3* en el año 2004.

Sin embargo, no fue hasta la década de 2010 que su diseño, producción y su uso empezó a externderse. Tras el desarrollo del *LBR 3* por parte de *KUKA*, otras grandes empresas del sector también desarrollaron y lanzaron sus modelos de *cobots*, como pueden ser *Universal Robots* o *FANUC*. [1]

Figura 2.1 [1]: Robot colaborativo

Los *cobots* ofrecen una gran variedad de ventajas respecto a los robots industriales tradicionales. La primera de ellas es, como ya se ha comentado anteriormente, que comparten espacio de trabajo con los humanos, por lo que no siempre es necesario instalar una pantalla protectora [1]. Esto sumado al hecho de que sus *footprints* (la superficie de su base de sujeción) suelen ser muy reducidas [2], hacen que el espacio necesario para que el robot lleve a cabo sus funciones es mucho menor que en el caso de un robot industrial. El poder prescindir de estas pantallas es debido a que, por su diseño, o bien se mueven a velocidades reducidas, o bien estas velocidades se reducen en el momento en el que se detecta (ya sea por sensores externos o equipados en el robot) que hay humanos cerca de su zona de trabajo. Todo esto se encuentra regulado por distintas normativas y estándares internacionales, se entrará a hablar de ellos más adelante en este documento.

Otro de los grandes reclamos que las empresas a los robots convencionales es su sencillez a la hora de instalarlos y programarlos [3], ya que muchos de ellos ofrecen métodos por los que los usuarios de estos aparatos no requieren de tener estudios superiores o de saber métodos convencionales de programación. Además, los *cobots* son también aparatos muy versátiles, por lo que es muy fácil reprogramarlos y cambiar la herramienta que se encuentra al final del brazo para cambiar su función dentro de una planta o de una línea de producción [2][3]. Sobre todo están enfocados para empresas pequeñas y medianas con producción propia, aunque podrían encontrar su lugar en grandes entornos industriales.

Si se continúa con los elementos que diferencian los cobots de los robots industriales tradicionales, también es necesario añadir que los primeros suponen un desembolso económico mucho menor a la hora de ser adquiridos [2] y están hechos de materiales mucho más ligeros. No obstante, este hecho sumado a, como se ha comentado anteriormente, que en muchas de las aplicaciones que se llevan a cabo no existe pantalla protectora, hacen que las cargas que puedan manipular estos aparatos sean mucho menores que las de un robot industrial [2]. Esto se hace para que en el caso de que haya un impacto con uno de los humanos con los que interactúa o con algún elemento del entorno, la inercia no sea lo suficientemente grande como para causar daños elevados.

2.1.1 Normativas y estándares que afectan a los robots colaborativos.

Si bien es cierto que los robots colaborativos son una tecnología al alza en este momento, ya hay organismos de estandarización como puede ser la ISO (*International Standard Organization*) que ya han desarrollado normativas respecto a los tipos de robot colaborativo que existe y las medidas de seguridad que deberían seguirse en función de la aplicación que se quiera llevar a cabo con ellos. Concretamente, son las normativas ISO 10218-1:2011, ISO 10218-2:2011 e ISO/TS 15066:2016 las que hacen referencia a este tipo de robots[1]. Las dos primeras se tratan de normas que hablan sobre la robótica en general y la última se publicó de forma complementaria a estas dos, debido a que no cubrían las reglas de seguridad para robots colaborativos.

2.1.1.1 Tipos de características de robots colaborativos según la ISO 10218-1:2011

Esta normativa separa las características que se pueden encontrar en este tipo de robots en cuatro:

- Parada de seguridad monitorizada: Esta es la funcionalidad se utiliza sobre todo cuando la interacción robot-humano es esporádica, y el humano sólo necesita entrar puntualmente al espacio del robot. El resto del tiempo este último se encuentra funcionando de forma autónoma. Cuando el robot se encuentra trabajando en este modo, su movimiento se frena completamente mediante la acción de los frenos del robot cuando una persona entra dentro de una zona previamente delimitada. Esto permite que el movimiento del robot se detenga sin necesidad de realizar una parada de emergencia que detenga el robot. Esta forma de trabajo precisa de sistemas de detección que comprueben si un humano se encuentra en la zona de seguridad [4].
- Guiado manual: Con este modo, se puede “enseñar” de forma manual al robot un camino. Existe un mecanismo que es capaz de liberar el robot y permite moverlo de forma manual hasta una posición determinada, esta posición es almacenada como un *waypoint*, con varios se pueden crear distintos tipos de aplicaciones de posicionado o

pick & place. Para el caso de los robots colaborativos, control de la fuerza y velocidad es necesario para evitar daños y accidentes [4][5].

- Monitorización de la velocidad y la separación: Con este modo lo que se consigue es información sobre el entorno en el que se encuentra el robot colaborativo mediante distintos tipos de sensores, ya sean láseres o sistemas de visión que sean capaces de detectar si se va a producir una colisión con algún objeto o persona que se pueda encontrar. Esto también permite que la velocidad o el comportamiento del robot varíe en función de su proximidad a los obstáculos. Este modo es el más apropiado cuando la interacción humano-robot sea habitual, y puede configurarse de la forma que más se adapte a la aplicación que se quiera llevar a cabo [4][5].
- Limitaciones de potencia y fuerza: Este es el tipo al que se corresponden los que se conocen más habitualmente como robots colaborativos, ya que trabajan con cargas más limitadas que los robots industriales, reduciendo así su fuerza y trabajan con potencias mucho menores. Mientras que todos los casos expuestos anteriormente se podrían aplicar a robots industriales convencionales, estos suponen un nuevo tipo que presenta una geometría mucho más redondeada, como el que puede verse en la figura 2.1 (para evitar daños en el caso en el que se produzca una colisión con parte del entorno) y no requieren de ningún tipo de visión ni sensor que controle el entorno puesto que se detienen en el momento en el que se produce un contacto con un objeto [4][5].

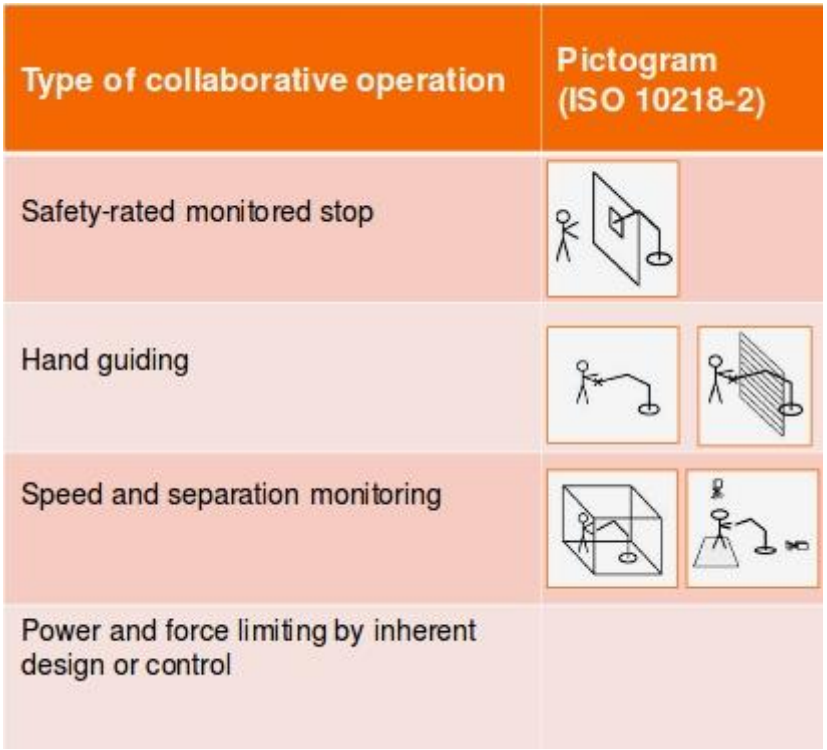

Figura 2.2 [2]: Pictogramas de la ISO 10218-2 que indican el tipo de operación colaborativa.

Cabe destacar, que según la página de la ISO [6], esta normativa se encuentra en revisión, por lo que es posible que pronto estos tipos de operaciones colaborativas o bien, se modifiquen o se amplíen.

Para el caso del proyecto del que se habla en esta memoria, tenemos un robot UR3 de *Universal Robots* y podría considerarse que se está trabajando en el caso de aplicación colaborativa de "Limitaciones de potencia y fuerza", ya que, en caso de contacto, o incluso movimiento brusco, el robot se detendrá.

2.2 MÉTODOS DE PROGRAMACIÓN DEL UR3.

Existen diversas formas de generar el movimiento del robot *UR3* de *Universal Robots*, que es el modelo de robot que se va a utilizar para este proyecto. En concreto se analizarán tres de ellas, explicando sus ventajas e inconvenientes para, al final, seleccionar una.

2.2.1 Configuración inicial del robot.

Polyscope se trata de una interfaz gráfica desarrollada por Universal Robots. Supone el primer contacto con cualquier tipo de software antes de poder realizar cualquier maniobra con el robot ya que es aquí donde se configura y donde se inicia el movimiento del mismo. *Polyscope* se ejecuta en la pantalla táctil que viene junto con el robot, está basado en el sistema operativo Debian Linux. [7]
La pantalla inicial tiene el aspecto mostrado en la figura 2.3:

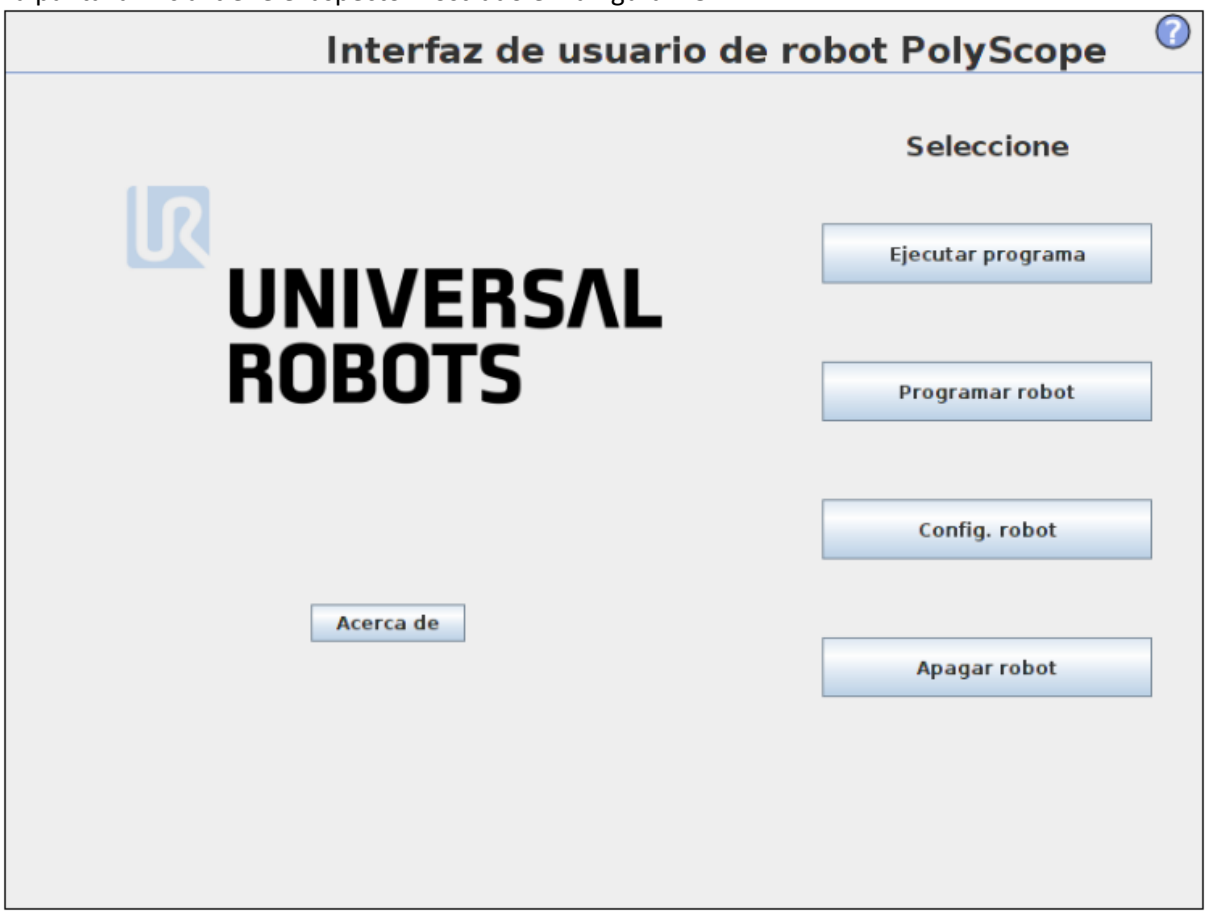

Figura 2.3: Pantalla de inicio de *Polyscope*

Para poder iniciar cualquier tipo de movimiento en el robot, ya sea manual o programado, es necesario encender el robot. Para esto, se entra en la configuración del robot, y la pantalla nos mostrará el siguiente menú:

Figura 2.4: Pantalla de configuración del robot en *Polyscope*

De este menú, sobre todo resultarán útiles a la hora de trabajar con el robot las opciones de "Inicializar robot" y "Configurar red". Si se intenta enviar algún tipo de comando al robot sin que este se haya inicializado, éste no responderá. La pantalla de inicialización es la siguiente:

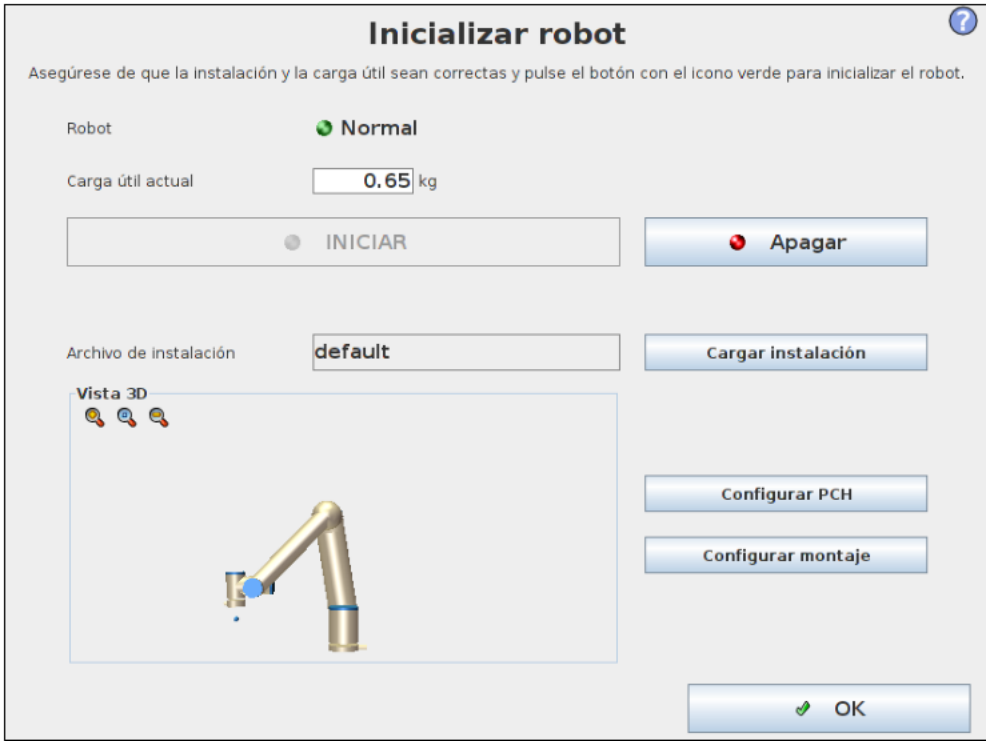

Figura 2.5: Pantalla de inicialización del robot en *Polyscope*

En esta pantalla se puede ver en qué estado se encuentra el robot. Si es la primera vez que se inicializa, el robot se encontrará apagado. Este estado indica que el robot no está recibiendo tensión y sus frenos se encuentran activos. Si se acciona "Iniciar" una vez, el robot pasa a estar en estado inactivo, esto significa que ya se encuentra recibiendo tensión, pero los frenos siguen

accionados. Por último, si se vuelve a pulsar “Iniciar” el robot pasa a estar en estado activo (el que puede verse en la figura 2.5), similar al anterior, pero con la diferencia de que las articulaciones ya se encuentran libes, por lo que ya se pueden enviar instrucciones para que el robot se mueva. Ahora se analizarán los distintos métodos de envío de instrucciones para el robot.

2.2.2 Movimiento manual y programas más básicos usando *Polyscope*

El método más sencillo para generar movimiento en el UR3 es mediante el control manual que existe dentro del *software Polyscope*. Para acceder a él, una vez inicializado el robot, se ha de volver a la pantalla inicial y seleccionar “Programar robot”, una vez aquí vemos las siguientes pestañas.

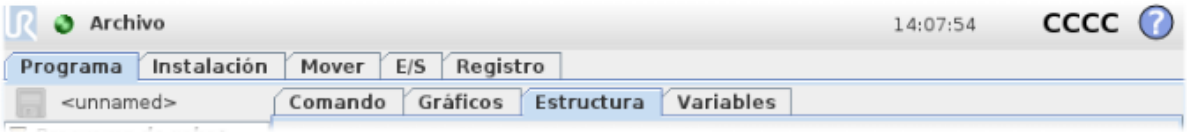

Figura 2.6 [4]: Pestañas de la opción “Programar robot” de *Polyscope*

Si se selecciona la opción “Mover”, se entra en la siguiente pantalla:

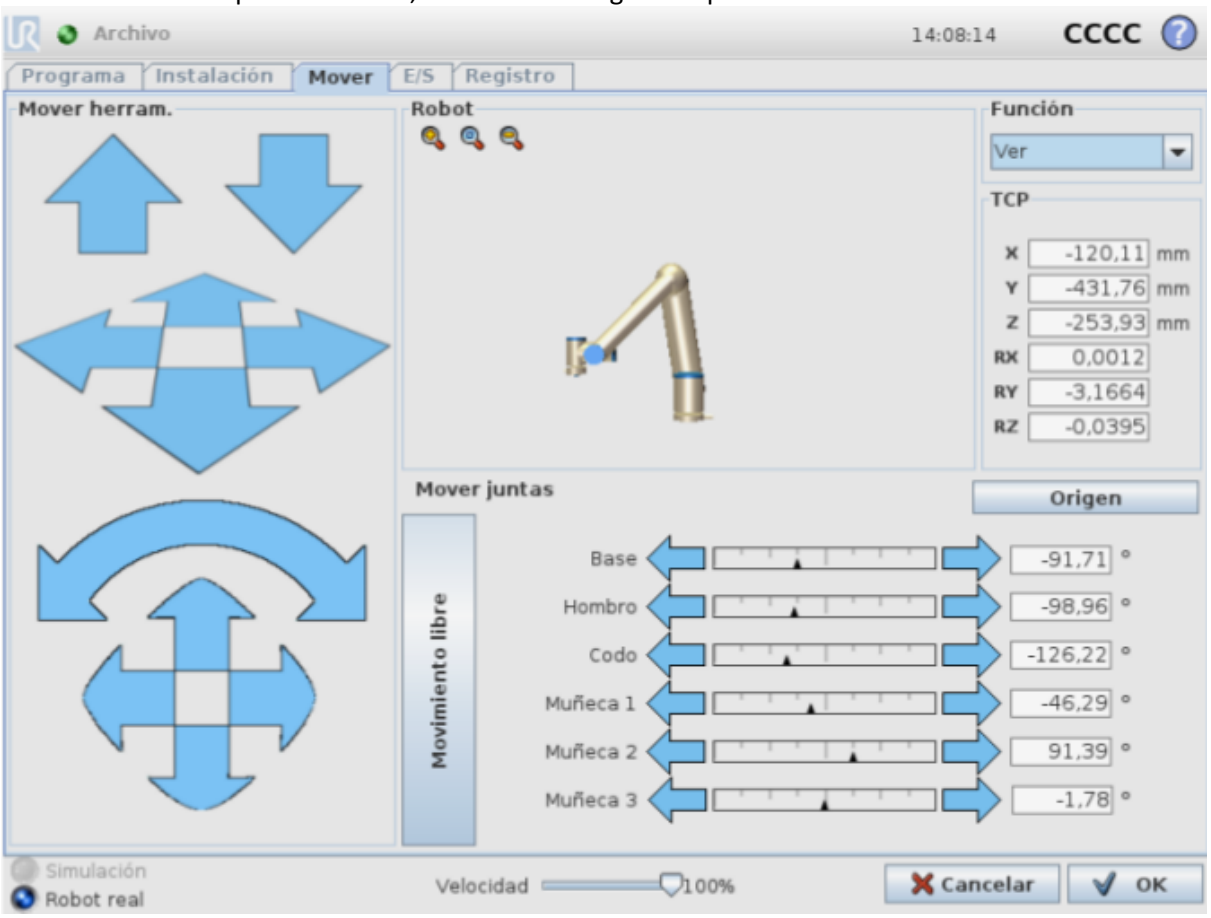

Figura 2.7 [4]: Pantalla de movimiento manual

Una vez en esta ficha, se puede, o bien controlar el ángulo en el que se encuentran las distintas articulaciones desde la sección derecha inferior de la figura *x*. También se puede dirigir el final del brazo (también conocido como *end-effector*) con los controles disponibles en la parte izquierda de la pantalla. Como se puede ver, también es posible controlar la velocidad a la que

el robot se mueve (en términos relativos, se puede configurar la velocidad máxima de las articulaciones como una de las funcionalidades que ofrece *Polyscope*) y es posible ver una representación del movimiento del robot en una pantalla.

Además, *Polyscope* nos ofrece también la posibilidad de realizar aplicaciones básicas para cubrir necesidades de programación que puedan surgir sin necesidad de conocer ningún lenguaje. Para ello, se crea un "Programa vacío" en la pestaña de "Programa", y pulsando en "Estructura" aparece el siguiente menú:

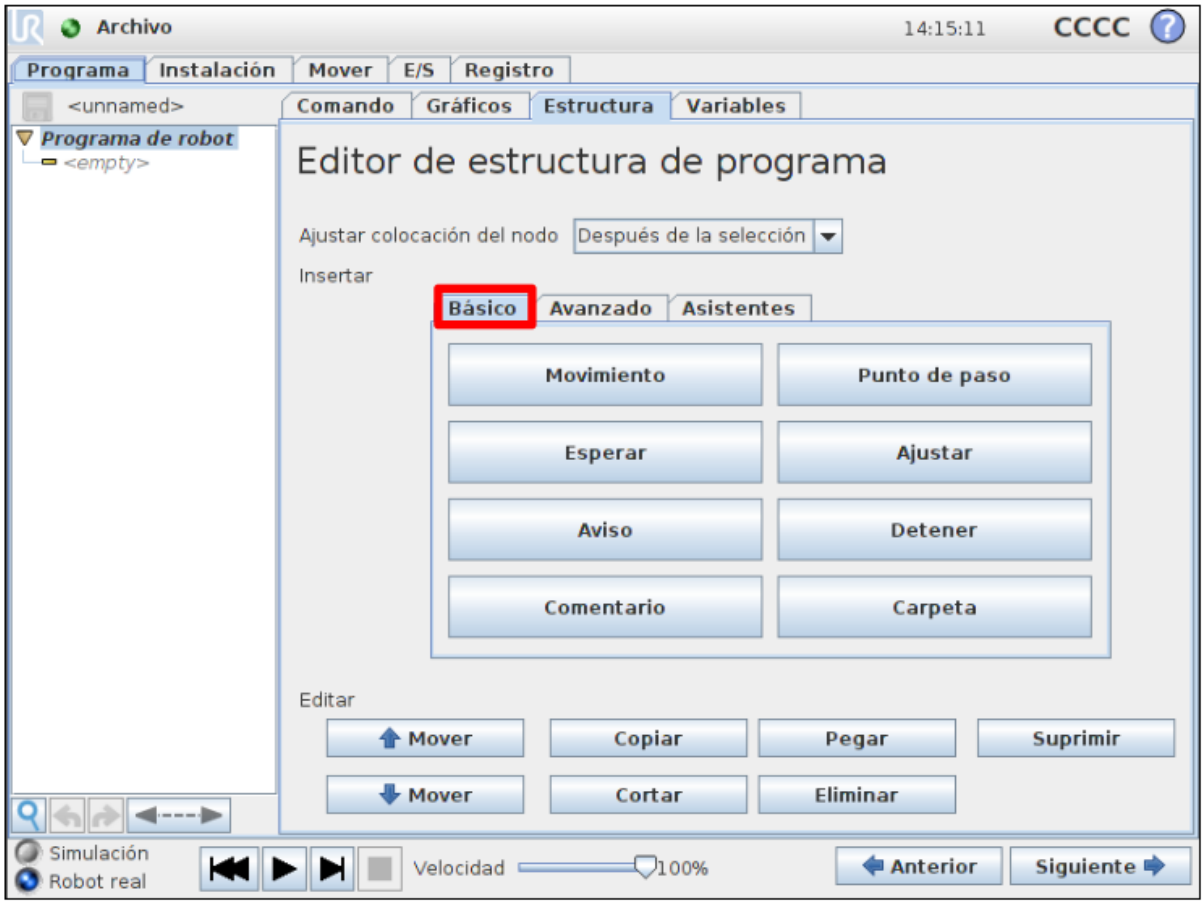

Figura 2.8 [3]: Menú de programación de *Polyscope*

Dentro de "Básico" se tienen los comandos como puede verse en la figura 2.8, y se pueden añadir las siguientes operaciones en el programa:

- Movimiento: Permite definir los parámetros de un movimiento del robot, es decir, controla cómo se va a comportar el desplazamiento del mismo de un punto de paso a otro. Aquí se puede configurar la velocidad y la aceleración de las articulaciones, así como los tipos de movimiento. Existen cuatro tipos de movimientos programados de serie:
    - MoveJ: Todas las articulaciones se mueven a la vez con el fin de hacer el movimiento más corto hasta la posición final.
    - MoveL: Las articulaciones se mueven de forma que la trayectoria descrita por la herramienta es una línea recta
    - MoveP: Similar a la lineal, pero para ciertos puntos de paso se puede indicar un radio de curvatura que hará que no pase exactamente por el punto.
    - MoveC: Se considera un sub-caso del anterior, se trata de movimientos circulares. Precisan de 3 puntos (inicial, paso y final) [7].

- Puntos de paso: Son los puntos de la trayectoria del robot. Se obtienen moviendo el robot físicamente hasta el punto deseado y posteriormente guardándolos en el programa [8].
- Esperar: Se configura una espera, o bien mediante una señal E/S, o bien una espera temporizada.
- Ajustar: Se puede indicar aquí la inicialización de elementos externos como puedan ser salidas digitales o analógicas que estén conectadas al robot.
- Aviso: Envío de mensajes al usuario, advertencias o errores.

Estas opciones se encuentran explicadas de forma mucho más amplia junto con los métodos de programación avanzados (que incluyen expresiones condicionales y bucles) en el manual disponible en la página de *Universal Robot.*

2.2.3 Scripts

Los *Scripts* son programas o líneas de programas que se pueden cargar directamente en el robot o en el programa que se esté creando mediante *Polyscope* y que funcionan de forma muy similar a cualquier programa hecho mediante un lenguaje de programación estándar. Sin embargo, *Universal Robots* tiene su propio lenguaje para sus unidades: *URScript*. Este lenguaje funciona de forma muy similar a otras formas de programación, de hecho, es bastante similar a *Python*, puesto que también se pueden definir y utilizar variables, con la diferencia de que únicamente se tienen los siguientes tipos [9]:

• none
• bool
• number (Aquí están incluidas tanto las *int* como las *float*).
• pose
• string

Como se puede ver, la mayoría son conocidas ya que se utilizan en los lenguajes estándar. Sin embargo, la variable tipo *pose* no es tan habitual y se trata de un vector en el que se define una posición para el robot. Su formato es el siguiente: p[x,y,z,ax,ay,az] donde x, y, z son las coordenadas del *end-effector* en el espacio y ax, ay, az su orientación. Se puede utilizar para dentro de un programa enviar instrucciones de movimiento [3]. No obstante, para poder utilizar esto es necesario un conocimiento de la cinemática del robot y realizar pruebas previas para asegurarse de que la posición que se va a alcanzar no pone en peligro el robot o su entorno.

También es similar a muchos lenguajes en el hecho de que se puede controlar el flujo de los comandos a través de estructuras condicionales: *if, elif* (equivalente a *else if* en otros lenguajes) y *else.* A su vez, también existen bucles basados en la instrucción *while* y se puede utilizar la expresión *break* para salir de los mismos [9].

Pero, sobre todo, resulta interesante utilizar este lenguaje debido al uso de funciones. Éstas pueden ser creadas desde cero (como en la mayoría de los lenguajes de programación) o pueden utilizarse las ya existen [3] y que se encuentran completamente documentadas junto con su uso y las variables que necesitan en el manual de usuario de *URscript* disponible en la página de *Universal Robots.* Estas funciones predeterminadas permiten controlar, además del robot, cualquier aparato externo que se haya conectado al mismo, como pueden ser cintas transportadoras o cámaras.

Como puede verse es mucho más flexible y versátil que la forma de programación mediante comandos de *Polyscope*. Sin embargo, requiere de mayor conocimiento técnico para ser capaz de usar estas funciones.

2.2.4 *Robot Operating System* (ROS)

Los métodos de programación expuestos anteriormente para el *UR3* son los que desarrolló la misma empresa, y es ésta la que da la documentación para que los usuarios sean capaces de empezar a usarlo y desarrollar aplicaciones con él. Sin embargo, todo lo que se aprenda con esta documentación únicamente es válido para robots de la misma marca, ya que *Polyscope* y *URscript* sólo son utilizados por los robots *UR3, UR5* y *UR10.*

Existe, no obstante, un método de programación de robots de uso cada vez más extendido que permite que todo lo que se desarrolla para un tipo de robot pueda ser utilizado por otros modelos e incluso otras marcas. *Robot Operating System* (ROS) como queda reflejado en su página web: "*ROS es un firmware flexible para el desarrollo de software de robots. Se trata de una colección de convenciones, librerías y herramientas cuyo objetivo es la simplificación de la tarea de crear comportamientos de robots complejos y robustos a través de una gran variedad de plataformas robóticas*" [10]. Este *software* soporta diversos lenguajes de programación como C++, *Python* o *Java* [11], por lo que al eliminar los lenguajes exclusivos de cada marca, ayuda a que pueda ser utilizado por un mayor número de robots y al ser *software* libre, está en constante evolución y desarrollo.

Las aplicaciones se desarrollan en *paquetes* que son las formas de agrupar la información necesaria para llevar a cabo cualquier aplicación [12]. Están formados por *nodos,* librerías, archivos de configuración, etc. Los nodos son la unidad básica de programación que realizan las funcionalidades que se indican con el código. Una de las grandes ventajas de ROS es que la comunicación entre distintos nodos en ejecución se realiza de forma muy sencilla gracias a sus protocolos, estén estos en el mismo ordenador o, aunque sean dos máquinas distintas las que ejecutan los programas (una vez se han conectado por métodos convencionales) [12]. Entre estas formas de comunicación se encuentran:

- Topics: Sistemas de envío de datos basados en publicación y subscripción a un canal (*Topic*) [13].
- Servicio: Uno de los nodos llama al servicio, recibe los datos del emisor, procesa la información y la devuelve. Se trata de un envío de información puntual, no como el caso del *topic* que puede estar constantemente enviándose [14].
- Acciones: Similar a los servicios, pero con la diferencia de que utilizan una situación actual y un objetivo, es decir, se pregunta si se ha llegado al objetivo y la respuesta llega cuando se alcanza [15].

Estos conceptos se ampliarán más adelante en este documento. *Roscore* es el servicio dentro del *software* de *ROS* que permite que esta comunicación sea posible ya que permite que los nodos sean visibles entre sí, y es el que establece las comunicaciones [16]. Cuando se lanza se inicializa lo siguiente:

- El *ROSMaster*: provee el servicio por el que los nodos se pueden nombrar y registrar en el sistema. También monitoriza el sistema de *topics* y el de *services* con los nodos que se encargan de publicar y recibir la información. El *Master* se encarga de, como se ha comentado antes, que los nodos sean capaces de encontrarse entre sí y una vez se han localizado, establecer una comunicación *peer-to-peer* [17].
- El *Ros Parameter Server*: se trata de una especie de librería en la que se almacenarán parámetros y variables que puedan ser necesitados por los distintos nodos, este concepto se ampliará más adelante.

- El *rosout:* Se trata de un nodo que se suscribe, registra y puede reenviar mensajes [18].

Las ventajas de utilizar este método de programación, como brevemente se ha comentado anteriormente, son numerosas ya que permite que el mismo código pueda ser utilizado por gran cantidad de tipos distintos de robots, está dotado de una gran cantidad de herramientas que complementan su funcionalidad como por ejemplo simuladores o puentes que permiten la utilización de otras librerías especializadas, la comunicación entre dispositivos (ya sean cámaras, robots, sensores u otros elementos) se puede producir de forma eficaz y sencilla, los lenguajes que utiliza son muy generales y gran parte de los programadores conoce al menos uno de ellos, por lo que su aprendizaje se simplifica y existen una gran cantidad de tutoriales que permiten aprender el funcionamiento de este *software* así como su funcionamiento.

Sin embargo, y a pesar de esto, se trata de un método para programar los robots bastante complejo y que presenta una curva de aprendizaje lenta. Otro de los inconvenientes es que está basado en el sistema operativo *Linux* y para poder aprovechar el completo potencial de *ROS* es necesario un periodo de acondicionamiento a este sistema.

2.2.5 Tabla resumen comparativa de los métodos.

En la siguiente tabla se presenta un resumen comparativo de los tres tipos de programación del *UR3* que se han comentado anteriormente comparando su dificultad, el nivel de documentación respecto a ellos que existe, su flexibilidad, cómo es su curva de aprendizaje y su universalidad.

Tabla 1: Comparativa entre métodos de programación

Método	Dificultad	Flexibilidad	Documentación	Curva de aprendizaje	Válido para otros robots
Polyscope	Baja	Poca	Buena	Rápida	Sí (Universal Robots)
Urscript	Media – Baja	Media	Buena	Rápida	Sí (Universal Robots)
Robot Operating System	Alta	Alta	Muy buena, pero deslocalizada	Lenta	Sí (Todos aquellos que soporten ROS)

Para el presente proyecto se utilizará ROS debido a que ofrece mayor versatilidad y flexibilidad, y debido a su facilidad para realizar comunicación con otros dispositivos ya que esto puede resultar interesante a la hora de realizar la aplicación.

2.3 MOVEIT!

MoveIt! se trata de uno de los diversos paquetes que se encuentra disponible en *ROS.* Se trata de un software que facilita el control de trayectorias para robots, con especial enfoque a robots con brazos. Incorpora las siguientes funciones: planificación de movimientos, manipulación, percepción 3D, cinemática, control y navegación. Además, provee al usuario de una plataforma fácil de utilizar para desarrollar nuevas aplicaciones robóticas [19].

Al estar basado en *ROS* ofrece las mismas ventajas que éste, como por ejemplo que las aplicaciones que se diseñen para un robot, con una alta probabilidad y sin tener que realizar un gran número de cambios, serán válidas para otras unidades (incluso de otras marcas). A su vez, *MoveIt!* hace uso del sistema de comunicación entre distintos nodos de *ROS* para poder funcionar, esto se verá más adelante cuando se explique la arquitectura de este *software*.

2.3.1. Arquitectura de alto nivel de *MoveIt!*

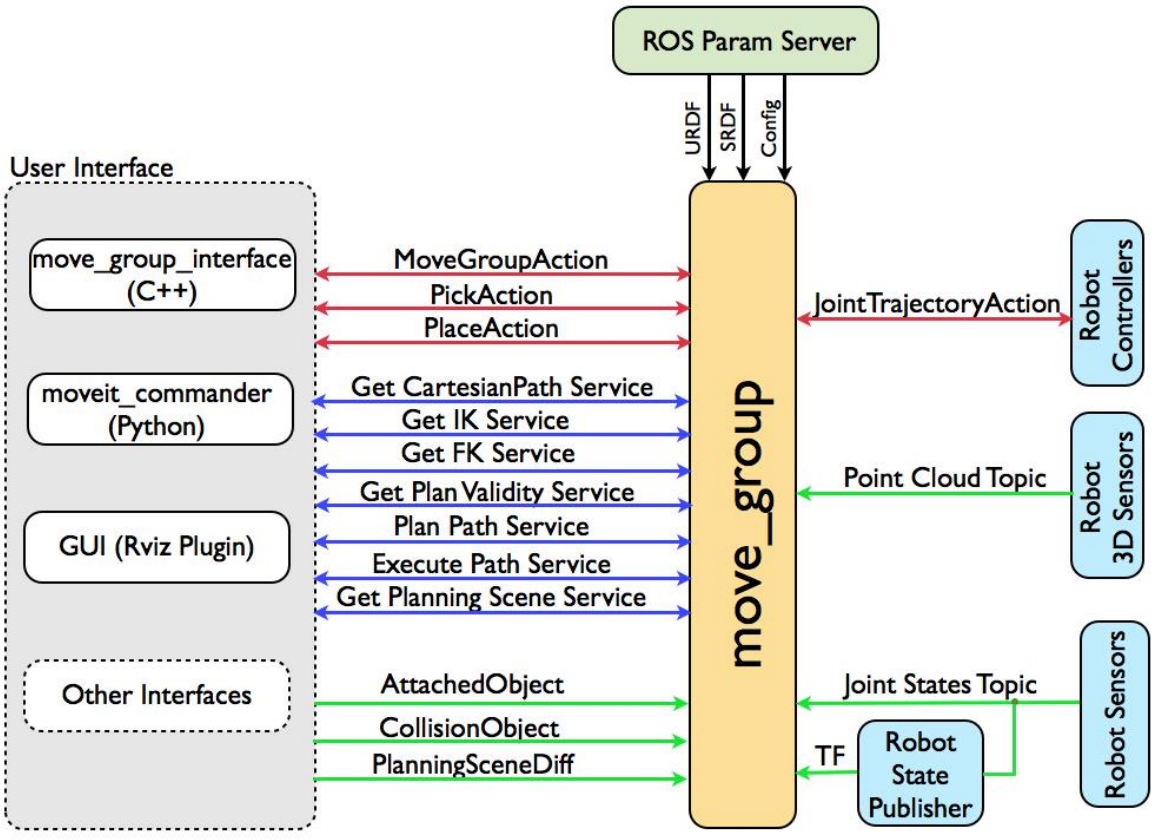

Figura 2.9 [5]: Esquema de la arquitectura de *MoveIt!*

Como se puede ver en la figura 2.9, el núcleo principal de este *software* se trata del nodo llamado *move_group* [20]. A la izquierda de este pueden verse las distintas formas que tiene el usuario de interactuar con dicho nodo. Las más destacables son:

- *Move_group_interface* (C++): Consiste en un programa que envía la información al nodo *move_group*. Este código se realiza en el lenguaje C++. Esta es una de las formas más flexibles puesto que permite, no sólo enviar instrucciones al robot, sino que se pueden

añadir objetos al entorno, establecer condiciones, flujos, avisos, etc. Este es el que se explorará en más detalle a la hora de desarrollar la aplicación a la que hace referencia esta memoria y se utilizará junto con la *Graphic User Interface*.

- *Moveit_commander* (Python): se trata de un programa que ya viene de serie junto con el paquete de *MoveIt!* y permite de forma muy sencilla y a través de una serie de comandos, controlar el robot por terminal. Dentro de este programa se pide seleccionar el conjunto de articulaciones que se quiere controlar y, posteriormente, mover el robot con una serie de comandos. Se puede acceder a la lista de comandos que ofrece este programa escribiendo *help* al ejecutarlo. Este programa se recomienda, sobre todo, para pruebas.
- *Graphic User Interface* (GUI, *Rviz*): Esta es la forma más intuitiva y sencilla de controlar el robot. *Rviz* carga el modelo del robot y aquí aparecen unas flechas que permiten cambiar su posición hacia la dirección que sea arrastrado. Esto permite ver los movimientos que hará el robot, e incluso realizar la planificación antes de ejecutar el movimiento, evitando así fallos y colisiones. En esta interfaz también se pueden cambiar de forma muy sencilla distintas opciones para la planificación, como puede ser el planeador, si se consideran los objetos que hay en el entorno para evitar su colisión, etc.

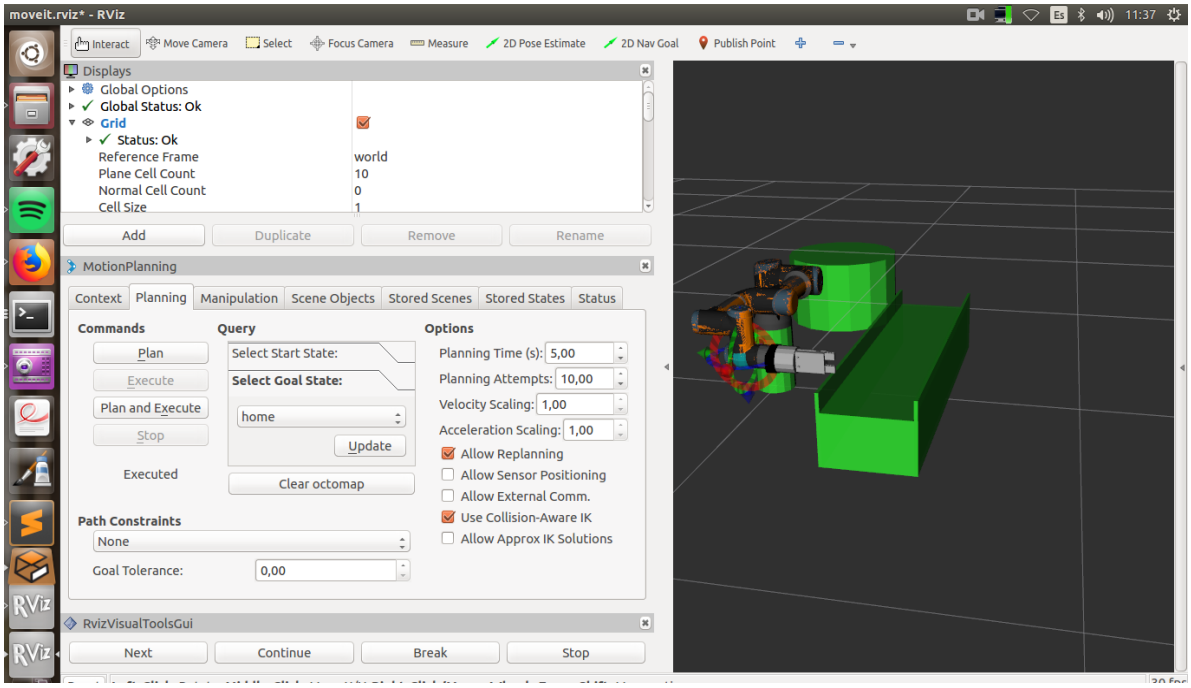

Figura 2.10: Apariencia de la *GUI* de *MoveIt!* en *Rviz.*

Como puede verse en el esquema representado en la figura 2.9, la interacción del usuario está conectada por diversos elementos al nodo *move_group*. Estos elementos se tratan de los distintos tipos de comunicación que existe en *ROS* y que lo que hacen es enviar la información que genera el usuario al nodo. Cada color de flecha representa un tipo de comunicación distinto, estos han sido explicados de forma breve anteriormente en este documento, sin embargo, es necesario ampliar ligeramente estas definiciones para comprender completamente el funcionamiento de *MoveIt!:*

Las flechas verdes representan comunicación vía *topic*. Como se ha comentado antes, se trata de un sistema de comunicación basado en publicación y subscripción a un canal, este canal es lo que se conoce como *topic*, y los diferentes elementos que publican y se suscriben a él son los nodos. Estos *topics* suelen transportar datos de un tipo determinado. La comunicación en este caso es de un único sentido, es decir, el nodo que publica no recibe confirmación ni información

de los subscriptores. Sin embargo, sí es posible que varios nodos se encuentren publicando información en un *topic* y que existan varios nodos suscritos a él [13] [21].
En este caso concreto, se puede ver que la información se produce desde la interfaz de usuario al nodo *move_group.* La información que se transmite de esta forma al núcleo es la referente a los distintos objetos que pueden aparecer en el entorno, o bien los que se tienen que tener en cuenta a la hora de planificar las trayectorias porque pueden producir colisión (*topic: CollisionObject*), aquellos que se encuentran adjuntos al robot (*topic: AttachedObject*) y por último también así se transporta la información que dicta si ha habido una diferencia entre el entorno actual y el siguiente (*topic: PlanningSceneDiff*).

Las flechas azules son comunicaciones realizadas vía *service.* Los servicios son un sistema de comunicación basado en llamadas y respuestas. Se diferencian de los *topics* en que, en éstos, la información se envía sin saber si el receptor está listo para recibirla o incluso puede perderse parte de la misma si la conexión tarda en establecerse, mientras que en los *services* están basados en un sistema cliente/servidor. El servidor llama al cliente, y hasta que este no responde no se produce el envío de información. También cabe destacar que por lo general los servicios se utilizan para que el cliente lleve a cabo un procesado de los datos y se devuelva al servidor el resultado [14]. Estos servicios por lo general se utilizan en procesos que requieren de tiempo de cálculo y que se utilicen de forma esporádica en el proceso que se esté llevando a cabo [21].
Para el caso de *MoveIt!* la información que se transmite de esta forma son: el envío de *waypoints* al robot, el recibir la cinemática del robot, tanto la inversa como la directa, planificar una trayectoria, recibir confirmación de si es válida, ejecutarla y recibir la información de cambios en el entorno.

Y, por último, las flechas rojas representan la última forma de comunicación más general que se puede encontrar en *ROS*, las acciones. Las *actions* se utilizan sobre todo cuando se busca cumplir un determinado objetivo. Se trata de algo similar a los *services* explicados anteriormente ya que también requieren de solicitud y respuesta, pero con la diferencia de que éstos son asíncronos. Las *actions* establecen un objetivo (*goal*) que inicia un comportamiento o un proceso, y establecen una señal cuando llegan a este (*result*). Se implementan utilizando una combinación de *topics* [15] [21].
Por tanto, en este esquema, y como es lógico por lo expuesto en su definición, las *actions* que existen son el movimiento del robot, la operación de *pick* y la de *place* (por ser operaciones que requieren del movimiento de un grupo de elementos que forman el *end-effector*).
En la parte superior de la figura 2.9 se puede ver que *move_group* está conectado por tres flechas de color negro a otro elemento llamado *ROS Param Server*. El *Parameter Server* se trata de otra de las ventajas que ofrece *ROS* a la comunicación entre nodos. Básicamente, es una especie de "diccionario" o base de datos multivariable y compartido que es accesible para los nodos operativos del sistema, y éstos lo utilizan para almacenar y recuperar parámetros durante la ejecución. Se utiliza principalmente para datos de configuración. Es visible de forma global para que sea más fácil de localizar y utilizar para las distintas herramientas [22].
En el caso de este nodo *move_group* la información que se guarda en este *Parameter Server* es la que corresponde a los archivos *URDF, SRDF y Config*. A continuación, se explicará para qué sirve cada uno de ellos:

- El archivo *URDF (unified robot description format)* es el que contiene la información del modelo del robot [23]. Dentro de este fichero se puede definir, a partir de modelos 3D de las partes que lo componen, el nombre de las piezas y qué tipo de articulación existe entre ellas. También permite definir otras propiedades que puedan existir, como la inercia, el peso o incluso límites de movimiento si se utiliza un fichero *.xacro* como *URDF.*
- El fichero *SRDF (Semantic Robot Description Format)* contiene información sobre el robot que no puede recoger el *URDF* y que puede resultar interesante para algunas

aplicaciones [24]. Sobre todo, recoge la información de cómo se agrupan las distintas partes y articulaciones del robot y la relación entre ellas en caso de existir alguna condición, como por ejemplo, dos partes que van juntas están en contacto continuado.

- Por último, los archivos *Config* suelen tener la extensión.*yaml* y son los que configuran distintos parámetros dentro de *MoveIt!*, como la velocidad y aceleración máxima de las juntas, o aspectos de los planificadores que pueden usarse.

Toda esta información es leída por *ROS* y se envía a *move_group*, como puede verse, a través del *Parameter Server.*

Ahora, en el lado derecho de la figura 2.9 se puede ver que este nodo está conectado a través de una *action* con los controladores del robot. Estos reciben las instrucciones de movimiento por parte del nodo *move_group* y devuelven el *result* cuando se ha alcanzado la posición deseada, o bien responden con un error cuando esta no puede alcanzarse. Por otra parte, el nodo *move_group* hace las veces de cliente a la hora de conectarse con el controlador del robot puesto que es un servidor el que ha de establecer el contacto [20].

Por otra parte, si existen sensores que reciban información sobre el entorno del robot, como un sensor 3D, este puede enviar la información al nodo vía *topic.* Y, por último, se tienen los sensores del robot que ayudan a mantener control de la posición. Éstos envían información constantemente también vía *topic* tanto a *move_group* como a otro nodo llamado *robot_state_publisher*.
En la página de información de *ROS, WikiROS* explican que este nodo: "*permite publicar el estado de un robot a* tf*. Una vez el estado es publicado, está disponible para todos os componentes del sistema que también usan* tf*. El paquete toma los ángulos de las articulaciones del robot como información de entrada y publica las posiciones 3D de las articulaciones del robot, utilizando un árbol cinemático del modelo del robot".* En otras palabras, este nodo utiliza la información que posee internamente del modelo cinemático del robot, y conociendo la posición de las articulaciones, permite computar y transmitir la información de la posición 3D del robot [26]. Esto combinado con el uso del paquete *tf* permite controlar todos los distintos ejes de coordenadas de las piezas que forman el robot, la información global de la posición del robot [20] y la relación que existe entre los distintos sistemas (tranformadas) y que la comunicación entre ellos pueda realizarse de la forma más directa y sencilla posible y a lo largo del tiempo [27]. Esta información se envía a *move_group* a través de uno o varios *topics*.

2.3.2 Tratamiento de la cinemática en *MoveIt!*

En *MoveIt!* existe una infraestructura basada en complementos(*plugins*): actualmente hay dos opciones para la resolución de la cinemática del robot con la que se está trabajando. La primera de ellas es optar por utilizar la que calcula de forma automática cuando se aplica el *MoveIt! Setup Assistant*, esto es el asistente para generar paquetes de configuración de *MoveIt!*. Éste utiliza el *solver* numérico basado en jacobianos *KDL*.
Sin embargo, muchos usuarios prefieren utilizar sus propios algoritmos de resolución de cinemática inversa, para esto existe el *IKFast Plugin* que permite generar el código en C++ para que dichos algoritmos funcionen en un robot en concreto [20].

2.3.3 Planificación del movimiento en *MoveIt!*

La planificación del movimiento es aquello que se encarga de la problemática de mover el brazo a una determinada configuración permitiendo al *end-effector* llegar a una posición sin que el robot se choque con ningún obstáculo, ya sea un objeto externo o las propias partes del robot que pueden interponerse en el movimiento [28]. Estos planificadores se incorporan a *MoveIt!* en forma de *plugin* para que sea más fácil la comunicación y el uso de diversos tipos de planificadores. El nodo *move_group* se conecta con estos planificadores a través de una *action* o un *service* de *ROS* [20]. Por el momento, el planificador que funciona por defecto es el *OMPL.* Sin embargo, también se encuentran disponibles otros como *CHOMP.* Se entrará en detalle sobre estos más adelante en el documento.

La planificación funciona de la siguiente forma: En primer lugar, se envía una petición de planificación de movimiento que claramente especifica lo que se quiere hacer con el robot, entonces el planificador encuentra una trayectoria para todas las articulaciones en la que se tienen en cuenta colisiones y que permite llegar a la posición objetivo que se busca. *MoveIt!* también permite tener en cuenta posibles objetos que se puedan coger (por ejemplo, en aplicaciones de *pick and place*) a la hora de calcular las trayectorias del robot. A su vez, también se pueden añadir a la configuración las siguientes restricciones cinemáticas:

- Restricciones de posición: evitan que una de las partes del robot (también conocidas como *links)* alcancen una región determinada del espacio.
- Restricciones de orientación: evitan que uno de los *links* del robot se detengan en una orientación determinada, dados unos límites de rotación en el eje *x*, *y* o *z.*
- Restricciones de visibilidad: evitan que una parte de un *link* se encuentre dentro de la zona de visión de un sensor.
- Restricciones del usuario: Aquellas diseñadas por el usuario con una *callback* definida por el mismo [20] [28].

El resultado de esta planificación se trata de una trayectoria que mueve el brazo a la posición deseada. Se considera que es una trayectoria y no solo una serie de puntos puesto que la planificación realizada también tiene en cuenta las restricciones de velocidades y aceleraciones de cada articulación que se han especificado en los archivos *config* de *MoveIt!.*

Por último, *MoveIt* cuenta con una *pipeline* formada por planificadores de movimiento y adaptadores de solicitudes de planes. Son estos adaptadores los que permiten que se puedan pre-procesar las solicitudes de planes, y también hacen posible post-procesar las respuestas a estos planes. Esta operación quedar representada por la figura 2.11.

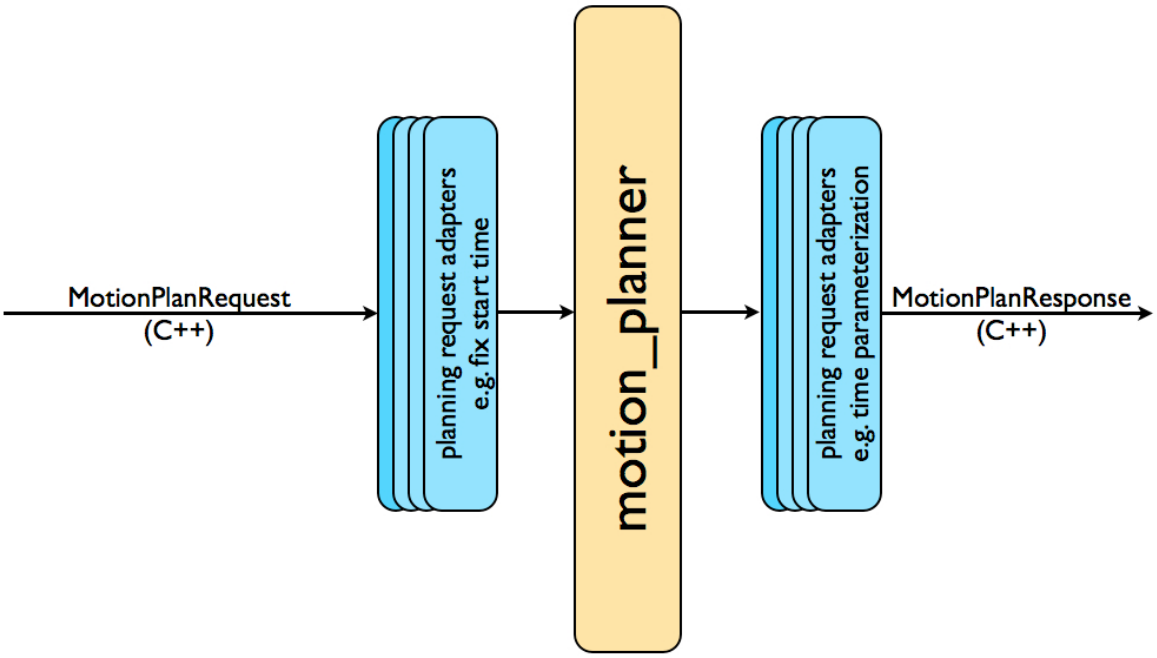

Figura 2.11[5]: Representación gráfica de los adaptadores de solicitud de planes

El pre-procesado puede utilizarse, por ejemplo, cuando el estado inicial del robot está ligeramente fuera de los límites de las articulaciones del robot, por lo que puede considerarse una aplicación muy útil. El post-procesado, por ejemplo, puede utilizarse en distintos tipos de operaciones, como por ejemplo, convertir los caminos generados en trayectorias [20].
Por defecto, *Moveit!* provee los siguientes adaptadores:

- *FixStartStateBounds*: Este es el que se corresponde a la situación que se ha utilizado como ejemplo para el pre-procesado, ya que se dedica a corregir la posición inicial de un robot en caso de que esta se encuentre fuera de los límites de las articulaciones que han sido establecidos. Si este adaptador no existiera, no sería posible computar una trayectoria, puesto que al intentar calcularla ya se tendría el error de que el robot está fuera de los límites. Sin embargo, esto sólo se utiliza cuando el robot está "muy alejado" de estos límites ya que, de ser así, utilizar esto puede no ser la mejor solución [28].
- *FixWorkspaceBounds:* Este adaptador define un entorno de trabajo cúbico de 10m³ [28].
- *FixStartStateCollision:* Con esto se intentará corregir un estado inicial en el que ya exisita colisión a partir de modificar ligeramente la posición original de las articulaciones del robot. El factor que determina cuánto se modificará esta posición es conocido como *jiggle_factor* y se trata de un porcentaje del rango de movimiento de la articulación en cuestión. El otro parámetro que se puede especificar para este adaptador es cuántos intentos aleatorios de arreglar la posición para que el resultado no tenga colisión puede llevar a cabo [20].
- *FixStartStatePathConstrains:* Similar al primero, pero este se aplicará cuando el estado inicial de un plan de movimiento no obedezca las restricciones de camino establecidas. Este adaptador intentará calcular un camino entre la posición actual y un punto final en el que estas restricciones se cumplan, esta nueva localización servirá como estado inicial para la planificación.

- *AddTimeParametetization*: Se corresponde a la situación expuesta como ejemplo en el caso de post-procesado. Utiliza los caminos que se planifican y realiza una parametrización basada en el tiempo para convertirlos en trayectorias utilizando las restricciones que existen previamente en el fichero *joint_limits.yaml*.

2.3.4. Planificador de escenarios de *MoveIt!*

Como se vio en la explicación de la trayectoria, *MoveIt!* permite enviar información a *move_group* referente al espacio en el que va a trabajar el robot. Esta información se generaba en parte desde la interfaz de usuario y en parte se recibía de los sensores del robot y es gestionada por el *planning scene monitor* que forma parte de *move_group*. Este espacio está compuesto por los objetos que puedan existir y por el estado del robot en sí mismo. La arquitectura en más detalle de esto tendría la siguiente forma:

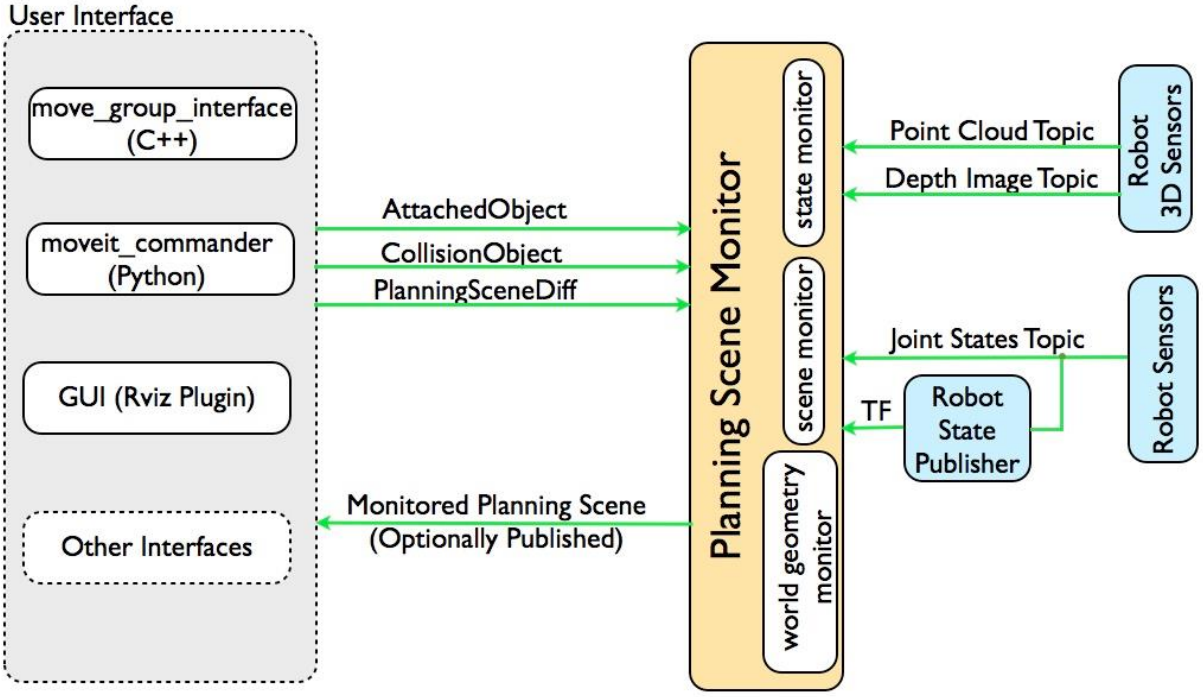

Figura 2.12 [5]: Arquitectura de alto nivel referente al *Planning Scene Monitor.*

Como puede verse en la figura 2.12, y como se ha comentado antes, el *Planning Scene Monitor* recibe la información de la posición actual del robot (tanto por el *topic Joint States,* como a través de *tf*), recibe la información de la interfaz de usuario con los objetos que se han creado para definir el espacio y la información proveniente de los sensores, y la envía al *World Geometry Monitor.* Este componente del *Planning Scene Monitor* es el que resulta más interesante, ya que es el que construye la geometría del mundo utilizando esta información. Utiliza una cuadrícula de ocupación para construir la representación 3D de entorno del robot. Esta cuadrícula se genera con el *occupancy grid monitor* que se trata de otro *plugin* que se encarga de manejar los distintos *inputs* que recibe (debido a los distintos tipos de sensores que existen). *MoveIt!* soporta dos tipos de inputs por defecto:

- Nubes de puntos
- Imágenes de profundidad

Además, existe la posibilidad de añadir más tipos de *inputs* al *occupancy grid monitor*.

Dentro del *topic CollisionObject* se definen los objetos que pueden colisionar con el robot de forma que, a la hora de realizar los cálculos y los planes de movimiento, *MoveIt!* los tiene en cuenta y los evita. Los objetos que pueden definirse de esta clase en *MoveIt!* son los siguientes:

- Mallas
- Formas primitivas (cajas, esferas, cilindros, etc.)
- *Octomaps*: Mapa de ocupación 3D que permite definir distintos tipos de entornos sin conocimiento predeterminado de ellos [29].

Sin embargo, esta operación es muy costosa a nivel computacional, es por tanto que *MoveIt!* incorpora la matriz de colisión permitida (*Allowed Collision Matrix, ACM*). La *ACM* almacena un valor binario que indica si es necesario realizar la comprobación de colisión de dos cuerpos, esto es debido a que en muchos casos es posible que estos cuerpos nunca se encuentren, o que estén en constante contacto (como pueden ser dos articulaciones que están consecutivas). Si el valor entre dos cuerpos en la *ACM* es 1, no hay necesidad de comprobar la colisión entre ellos, por lo que se ahorra bastante capacidad computacional [20].

2.3.5 Planificadores de movimiento disponibles en *MoveIt!*

2.3.5.1 *Open Motion Planning Library (OMPL)*

Se trata de una librería de planificadores que comprende una gran cantidad de algoritmos basados en la toma de muestras. Son planificadores que no incluyen en su código la visualización o la comprobación de colisiones con el fin de que se implementen junto con otras herramientas, y así es más fácil asegurar que se adaptarán a las mismas. Se trata por tanto de un complemento integrado ajeno a *MoveIt!*, desarrollado de forma externa [12].

En la actualidad esta librería cuenta con dos tipos de planificadores, los geométricos o los basados en control. Los primeros serán los que se utilicen para este proyecto. Sólo tienen en cuenta las restricciones geométricas y cinemáticas que puedan existir en el sistema. Se asume que cualquier camino encontrado puede convertirse en una trayectoria realizable [13].

Existen un gran número de planificadores dentro de esta categoría, sin embargo sólo se explicarán aquellos que sean utilizados para el proyecto. Estos son los siguientes:

Planificadores multi-búsqueda: son planificadores que construyen un *roadmap* del entorno completo y que puede ser utilizado para múltiples búsquedas [13]. Dentro de este tipo pueden encontrarse:

- *Probabilistic Roadmap Method* (*PRM*): Este es un algoritmo basado en el muestreado, en este caso se utiliza una serie de muestras para construir la hoja de ruta, y una segunda para comprobar si en dicho *roadmap* existe un camino que vaya del origen al objetivo. Dentro de esta librería existen diversas variantes del *PRM* [13]*:*
    - *PRM*:* El *PRM* normal intenta conectarse a un número fijo de vecinos, mientras que esta versión incrementa de forma gradual el número de intentos de conexión.

Planificadores de simple búsqueda: Estos planificadores normalmente genera un árbol de estados conectados a movimientos válidos. La diferencia entre ellos son los métodos heurísticos que se usan para establecer hacia dónde o cómo se expande dicho árbol. Muchos de los planificadores, a su vez, generan dos árboles, uno desde el origen y otro desde el objetivo y lo

que se busca es que lleguen a conectarse. Entre estos métodos podemos encontrar implementados en el *OMPL* los siguientes:

- *Rapidly-exploring Random Trees (RRT):* Existen muchas variedades de este algoritmo y depende de las restricciones que se le apliquen. Se generan caminos hacia el objetivo de forma aleatoria y se selecciona el camino cuyo final se encuentre más cerca del objetivo para ser el siguiente punto en el que se generen caminos [14]. Dentro de este podemos encontrar las siguientes variaciones basadas en este método [13]:
    - *RRT Connect*: Este planificador se trata de una versión bidireccional (es decir, genera dos árboles al inicio) del algoritmo RRT. Se considera una versión mejorada puesto que tiene mejores resultados que el RRT original.
    - *RRT **: Versión óptima de RRT. Este algoritmo converge en el camino óptimo como una función dependiente del tiempo.
    - *Transition-based RRT:* Intenta encontrar caminos cortos y de bajo coste computacional.
- *Expansive Space Trees (EST):* es un planificador que detecta el área menos explorada del espacio a partir de la medida de la densidad del espacio explorado, inclinándose hacia la exploración de las partes del espacio con la menor densidad [15]. Existen 3 versiones de este planificador: la original, la bidireccional y la versión basada en proyecciones [13]. En *OMPL* existen algunos planificadores que están basados en este pero no necesariamente se consideran variantes, el más interesante es:
    - *Single-query Bi-directional Lazy collision checking planner (SBL)*: Se considera una versión bidireccional del anterior con la diferencia de que sólo comprueba aquellos caminos en los que existe una posible solución (de ahí el apelativo de *lazy*).
- *Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE):* En este caso se utiliza una discretización para guiar la exploración del espacio de estados. En *OMPL* se ha implementado de forma simplificada utilizando únicamente una sola malla (discretización de un único nivel). La preferencia a la hora de explorar el espacio se da a la parte del espacio que ya ha sido explorada. Hay dos variantes:
    - *Bi-directional KPIECE (BKPIECE)*
    - *Lazy Bi-directional KPIECE (LBKPIECE)*

Además de algunos de estos existe la versión optimizada, la cual permite minimizar ciertos aspectos de estos algoritmos. Sin embargo, en muchos de ellos no se garantiza converger en un óptimo en el caso de que se busque mejorar otro parámetro que no sea la longitud del camino. Los planificadores de los anteriormente nombrados que se consideran optimizadores son:

- PRM*
- RRT*
- TRRT (Aunque este no tiene garantías de optimización).

2.4 - MODELOS *URDF* EN *ROS* Y *MOVEIT!*

URDF es el acrónimo de *Unified Robot Description Format*, y se trata de un fichero *XML* que permite describir completamente un robot indicando sus partes, articulaciones, propiedades físicas, etc [28]. Todos los robots que utilizan un simulador gráfico en *ROS* es gracias a que existe un fichero de estas características que lo describe. A su vez, los desarrolladores de *ROS* crearon los ficheros *xacro* que se trata de una optimización de los ficheros *URDF* para que estos no sean tan difíciles de leer y mantener. Esto es gracias a que permiten crear partes de código reutilizables para crear estructuras que se repiten, por ejemplo, para agilizar la creación de brazos y piernas que son idénticos, así como crear expresiones matemáticas y definir parámetros [42].

Al tratarse de un lenguaje de marcado (al igual que el *HTML*), la sintaxis del *URDF* está formada principalmente por etiquetas que contienen la información que permite definir las distintas partes y características del robot.

2.4.1 Etiquetas que permiten definir el robot.

A continuación, se hará una breve descripción de las etiquetas más básicas y los atributos que se pueden definir a partir de ellas:

- <robot>: Primera etiqueta del archivo. Permite definir si se trata de un *URDF* o de un *xacro*. También permite definir el nombre que va a recibir el robot que se describe a continuación.
- <xacro>: Esta etiqueta permite aplicar las ventajas que ofrece *xacro* respecto *URDF* como pueden ser propiedades asociadas a un nombre que se van a repetir a lo largo del código; incluir ficheros de propiedades que van a utilizarse o un macro, por ejemplo, para definir una serie de propiedades que van siempre juntas. Dentro de otras etiquetas también pueden utilizarse para crear condiciones, esto permite que el mismo modelo se pueda utilizar para simular el robot en distintas situaciones con distintas circunstancias.
- <link>: Esta sintaxis se utiliza para definir una de las partes que forman el robot. En esta etiqueta únicamente se define el nombre de dicha parte, sin embargo dentro de ella existen distintos tipos de etiquetas que permiten darle propiedades a esta pieza del robot [43].
- <visual>: Dentro de esta etiqueta se definen distintas propiedades que afectan a cómo se ve el modelo.
- <collision>: Esto define las propiedades de colisión de una de las partes. No siempre ha de coincidir con el visual.
- <geometry>: Esta se utiliza dentro de las dos anteriores, se define la forma del objeto que se está describiendo. Puede generarse a partir de una forma primitiva, como puede ser una caja, un cilindro o una esfera (dentro de cada una hay que especificar sus medidas), o bien, mediante la etiqueta *mesh* gracias a la cual se puede cargar un fichero cad.
- <origin>: De igual manera, esta etiqueta se usa principalmente dentro de visual y collision, pero también puede utilizarse dentro de inertial, y en este caso se utiliza para definir la posición en la que se encuentra el objeto en el espacio. Utiliza el origen del objeto definido anteriormente y lo sitúa en las coordenadas relativas a la última articulación que se ha definido que se indican. Se pueden dar coordenadas de eje x, y, z y además la rotación (si existe) respecto a estos ejes.

- <material>: Esto permite definir cosas referentes al visual del modelo, como por ejemplo el color de las partes del robot o la textura si se trata de un material en concreto. Esta etiqueta puede obviarse si se ha trabajado con ficheros cad que almacenan propiedades de los materiales.
- <inertial>: Permite añadir las propiedades inerciales a la pieza.
- <mass>: Se utiliza dentro de inertial y permite añadir el valor de la masa de la pieza en cuestión.
- <inertia>: Permite añadir el valor de la matriz de inercia. Únicamente se han de definir 6 parámetros puesto que la matriz rotacional de inercia es simétrica: ixx, ixy, ixz, iyy, iyz, izz.
- <joint>: Esta es la etiqueta utilizada para definir las articulaciones entre partes del robot. Los atributos asociados a esta y que han de definirse son el nombre de la articulación y el tipo. El tipo puede ser fijo, prismático, de revolución, planar (permite movimiento en el plano perpendicular al eje), *floating* (permite movimiento en todos los grados de libertad) o *evolute* (similar a una bisagra).
- <origin>: Utilización idéntica a la etiqueta que se define para las partes del robot.
- <parent>: Etiqueta que define cuál es la parte del robot que se define como *parent* dentro de la estructura en árbol del robot, se podría considerar que es la parte del robot que define el origen de la articulación. Este se indica con el atributo obligatorio *link*.
- <child> : Etiqueta que define cuál es la parte del robot que se define como *child* dentro de la estructura en árbol del robot, se podría considerar que es la parte del robot que define el fin de la articulación. Este se indica con el atributo obligatorio *link*.
- <axis>: Sirve para definir, en el caso de una articulación de revolución sobre qué eje se produce dicha revolución.
- <limit>: Se utiliza para definir límites para las articulaciones en caso de que existan. Se pueden utilizar los siguientes atributos: *lower* (indicar el límite inferior de una articulación), *upper* (indicar el límite superior de una articulación), *effort* (se impone la fuerza máxima que puede aplicar una articulación) o *speed* (se indica la velocidad máxima de dicha articulación)
- <dynamics>: Etiqueta que ayuda a definir propiedades físicas de la articulación. Los atributos que existen son *friction* (fricción) y *damping* (amortiguamiento).

2.5 CONCEPTOS DE VISIÓN UTILIZADOS.

Este proyecto cuenta con una cámara que se ha incorporado de forma adicional para que exista alguna forma de actualizar el entorno del robot en caso de que exista algún objeto en movimiento. Es, por tanto, necesario crear una aplicación que sea capaz de distinguir el objeto que se busca seguir de alguna forma, conocer información referente a este objeto (ya sea geométrica o de su color) y por último conocer sus coordenadas.

Para poder extraer toda la información que se busca, es menester aplicar a la imagen una serie de conceptos de visión que se explicarán a continuación.

2.5.1 Representación del color

En este trabajo se ha utilizado un *feed* constante de una cámara para hacer la detección de los objetos. Esta cámara capta las imágenes en color, ya que será así como se distinga el objeto a seguir. El modelo que se ha utilizado para la representación del color, en lugar del tradicional *RGB* que representa los colores en función de su composición de rojo, verde y azul, se ha optado

por usar el llamado *HSV (Hue, Saturation y Value)* que utiliza parámetros llamados *Matiz, Saturación* y *Valor* [45]. Una representación gráfica para ilustrar cómo funciona cada modelo se adjunta en la figura 2.13 a continuación:

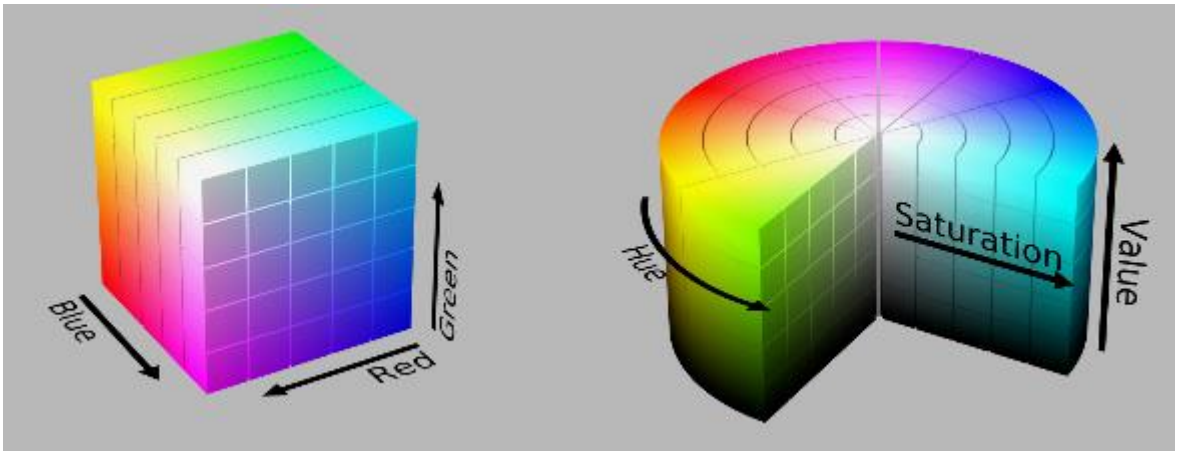

Figura 2.13[9]: Comparativa de los modelos de representación de *RGB* y *HSV* respectivamente.

En el caso del *RGB* puede verse que se trata de un modelo lineal que varía en función de los valores de rojo, verde y azul que toma el color que se busca. El modelo *HSV* cada dimensión del cono representa uno de los atributos: el matiz representa en el rango de una circunferencia los distintos colores, por ejemplo, en el grado 0 se encuentra el rojo, en el 120 el verde, y entre medias los tonos que los separan. La saturación juega con la cantidad el color que se tiene dentro de ese valor de grado en el que se encuentra [45]. Y por último el valor juega con el *brillo* del color, como si jugara con la cantidad de luz que ha recibido ese color, si el valor de este parámetro es bajo, es como si a dicho color se le iluminara con menos luz [46].

Esta representación se dice que se asemeja más a la forma de trabajar del ojo humano y se trata de un modelo de reconstrucción del color más fidedigno [46]. Es por tanto que se utilizará este modelo para trabajar con el color.

2.5.2 Segmentación de la imagen.

Este procedimiento busca a través de la aplicación de una serie de algoritmos, diferenciar en una imagen los objetos de interés respecto del resto de objetos. Para el objeto de este proyecto se ha utilizado una binarización que consiste en separar los objetos de interés dándoles un color blanco y al resto de elementos dándoles un color negro, esto puede verse en la figura 2.14 adjuntada a continuación.

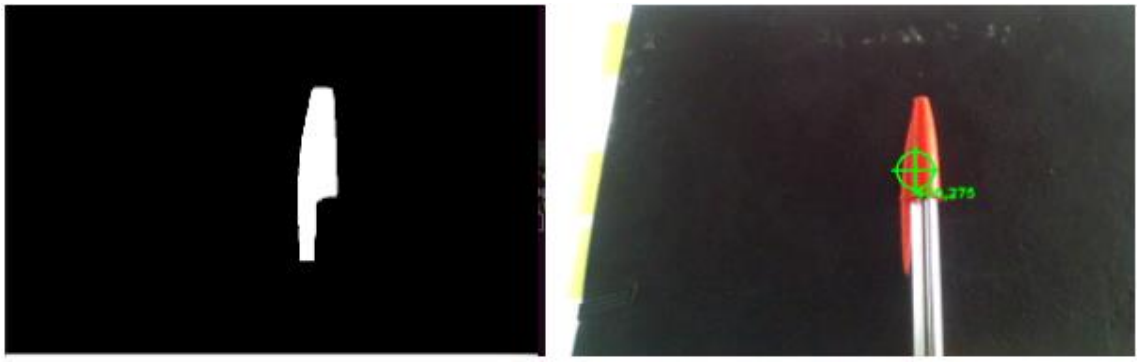

Figura 2.14: Ejemplo de binarizado.

Para este proceso, se utiliza el histograma de la imagen, y se determina el umbral del valor del píxel que diferencia los objetos de interés de los del entorno. En este caso no se está trabajando con imágenes en escala de grises, sino con un vídeo continuo que utiliza los fotogramas a color, como ya se ha comentado. Se utilizan los valores máximos y mínimos de la imagen para aplicar una función de binarizado que ya viene implementada en *OpenCV* que se encarga de recorrer la imagen comparando los valores de cada píxel con el umbral, si el valor del píxel es superior, este cambia el valor del píxel para que vea de color blanco, y si éste es inferior se representa como negro.

Una vez se ha realizado esta operación, se ve en la figura 2.14 que el único objeto que queda en la imagen es el objeto de interés.

2.5.3 Erosionado y dilatación de la imagen

Gracias al binarizado es posible aislar el objeto que se busca, sin embargo, para facilitar la detección del objeto debido a posibles ruidos que puedan generarse porque el color del objeto no se detecta de forma uniforme (por posibles reflejos), en este proyecto también se aplican operaciones de erosionado y dilatación de la imagen.

El erosionado consiste en reducir el contorno y el tamaño de los objetos que se pueden discernir después de la aplicación del binarizado a base de eliminar una serie píxeles del borde. Esto se utiliza sobre todo a la hora de detectar objetos, para separar objetos de interés que se encuentren en contacto. En este caso se utiliza para eliminar cualquier ruido que pueda existir, ya sea porque no se ha detectado correctamente el objeto o por algún tipo de reflejo que reproduzca el color sobre el entorno.

La dilatación es la operación contraria al erosionado, consiste en, una vez se tienen los objetos de interés diferenciados tras el binarizado, los píxeles que conforman el borde se aumentan y el tamaño del objeto es mayor. Esto puede resultar útil cuando el objeto que se busca se ha separado en dos debido a la superficie del objeto o a la iluminación de la sala.

Para el propósito de este proyecto se utilizarán ambas, una detrás de la otra con el fin de eliminar los posibles efectos del ruido para detectar el objeto nítidamente y posteriormente se dilatará para que el objeto aparezca como un elemento completo y su centroide se pueda detectar correctamente.

2.5.4 Momentos espaciales para la extracción de centroide

Una vez se ha detectado el objeto y mediante la segmentación se ha conseguido separar del resto de la imagen, lo siguiente es identificar en qué punto de la imagen se encuentra el centroide de dicho objeto. Para esto se utilizan los llamados momentos espaciales. Se definen como unas características geométricas de la imagen [47] y se tratan de parámetros ponderados calculados utilizando la intensidad de los píxeles [48]. Se utilizan sobre todo para el cálculo de propiedades con el fin de clasificar objetos.

Se calculan mediante la siguiente fórmula:

$$m_{pq} = \sum\sum f(i,j) * i^p * j^q$$

Ecuación 1 [47]: Fórmula momentos espaciales.

En esta ecuación *f(i,j)* representa el valor de la intensidad del píxel, i y j son el número de píxel recorriendo filas y columnas y el orden del momento viene dado por la suma de p+q.
Con esta fórmula y considerando p=0 y q=0, se tendría el cálculo del área del objeto detectado tras el binarizado en píxeles, puesto que se trata de la suma de los píxeles que conforman el objeto.

$$m_{00} = \sum\sum f(i,j)$$

Ecuación 2 [47]: Fórmula para el cálculo del área.

De forma adicional, si se combinan los momentos de orden 1 (cuando p=1, q=0 y p=0, q=1), es posible calcular el valor del centroide del objeto:

$$m_{10} = \sum\sum f(i,j) * i \quad ; m_{01} = \sum\sum f(i,j) * j$$

Ecuación 3 [47]: Momentos de primer orden.

$$\bar{\imath} = \frac{m_{10}}{m_{00}} = \frac{\sum\sum f(i,j) * i}{\sum\sum f(i,j)}; \; \bar{\jmath} = \frac{m_{01}}{m_{00}} = \frac{\sum\sum f(i,j) * j}{\sum\sum f(i,j)}$$

Ecuación 4 [47]: Cálculo del centroide basado en momentos.

Como puede verse, incorporar estos cálculos en el código puede resultar costoso, por lo que *OpenCV* incluye una funcionalidad que permite calcular todos los momentos hasta tercer grado.

2.5.5 Homografía

Por último, se requiere de situar los objetos en el sistema de coordenadas del robot. Para esto, y utilizando los datos de los centroides extraídos con los momentos espaciales, se puede calcular la matriz de homografía. Esta matriz representa la relación de transformación que existe entre los elementos de la imagen y los elementos en entorno real. Puesto que la forma de los objetos en este proyecto es irrelevante, sólo se precisará de conocer la relación de distancias.

El cálculo de la matriz de homografía se hace utilizando un patrón. Pueden utilizarse distintos tipos de patrón (de tablero de ajedrez, patrones asimétricos, etc…). Se busca identificar puntos en este patrón del que se conozca la distancia real, por ejemplo, en el caso de patrón asimétrico se detectan los centros de los círculos que conforman el patrón. El número de puntos que se utilizan puede variar en cada caso, el mínimo número que se utiliza es 4 puntos (a ser posible

que ocupen la mayor superficie y que no estén alineados entre sí), pero a mayor número de puntos mejor precisión. Sin embargo, el procedimiento de obtención de la matriz de homografía se hará con únicamente 4 puntos por simplificar.

Se parte de dos imágenes en las que aparecen 4 puntos. Supondremos que la figura de la izquierda se corresponde a la realidad (figura 2.15.a) y la figura de la izquierda es la transformación que se ha llevado a cabo (figura 2.15.b).

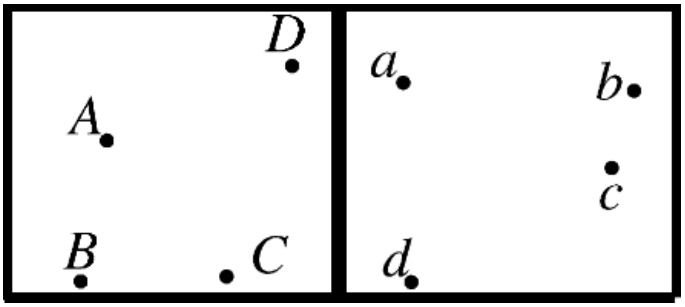

Figura 2.15 [10]: Imágenes para ejemplo de cálculo de la matriz de homografía

Las coordenadas de los puntos de la figura 2.15.a se denominarán como $[X_i \quad Y_i]^T$ las de los puntos de la figura 2.15.b como $[x_i \quad y_i]^T$ la relación que existe entre ellos puede expresarse de la siguiente forma:

$$s_i \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} * \begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix}; \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} = H_{3x3}$$

Ecuación 5 [51]: Relación entre coordenadas mediante matriz de homografía

Siendo H la matriz de homografía que se busca. Para el cálculo de ésta se tiene en cuenta que H está definida hasta factor escalar y es no singular.
Si se desarrolla el sistema de ecuaciones que se ha definido en la ecuación 5, queda lo siguiente:

$$s_i * x_i = X_i * h_{11} + Y_i * h_{12} + h_{13}$$

$$s_i * y_i = X_i * h_{21} + Y_i * h_{22} + h_{23}$$

$$s_i = X_i * h_{31} + Y_i * h_{32} + h_{33}$$

Ecuación 6 [51]: Sistema de ecuaciones desarrollado.

Si se sustituye la tercera ecuación del sistema por s_i, queda lo siguiente:

$$\begin{bmatrix} X_i & Y_i & 1 & 0 & 0 & 0 & -x_i X_i & -x_i Y_i & -x_i \\ 0 & 0 & 0 & X_i & Y_i & 1 & -y_i X_i & -y_i Y_i & -y_i \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Ecuación 7 [51]: Sistema de ecuaciones en forma matricial tras sutitución.

Donde queda una matriz de 8x9 que se conocerá como matriz A multiplicando a los componentes de la matriz de homología. Ahora se tienen dos opciones, o bien se calculan los parámetros de H utilizando métodos numéricos, o bien, si el rango de A es 8, se puede aplicar la descomposición en valores singulares (*SVD)* y los valores de H se encontrarán en la columna 9 asociada al menor valor singular (a ser posible 0) de la matriz VT.
Para este proyecto se ha optado por la segunda opción, puesto que gracias a la clase en la que se puede trabajar con matrices de *OpenCV* la aplicación de estos procedimientos se simplifican mucho.

Una vez se tiene la matriz H, para calcular las posiciones reales de los objetos detectados desde la cámara, simplemente hay que calcular la inversa de H y multiplicarla por las coordenadas del centroide en píxeles que se han obtenido mediante la aplicación de la función de momentos.

$$\begin{bmatrix} X_i \\ Y_i \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}^T * \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Ecuación 8: Cálculo de las coordenadas reales a partir de las coordenadas de la imagen.

El cálculo de esta inversa, de nuevo gracias a las funcionalidades de *OpenCV* se calcula de forma muy sencilla.

CAPÍTULO 3. DESARROLLO PRÁCTICO.

3.1 MATERIAL UTILIZADO.

En este apartado se hará una descripción detallada de los materiales utilizados para llevar a cabo el presente proyecto:

3.1.1 *UR3.*

El *UR3* se trata de un robot colaborativo de la marca *Universal Robots*. Es el robot de menor tamaño de entre todos los que ofrece esta marca pesando únicamente 11 kg [34] (sin herramienta colocada en su *end-effector*). Permite indicarle movimientos y ser programado de formas muy diversas, como ha podido verse en el apartado de esta memoria que habla de ello. Se trata de un robot que cuenta con 6 articulaciones de rotación. Ahora se entrará a hablar de las distintas especificaciones que ofrece.

1. Rendimiento:

Rendimiento	
Repetibilidad	±0,1 mm / ±0,0039 in (4 mil.)
Rango de temperatura ambiente	0–50°*
Consumo de energía	Mín. 90 W, estándar 125 W, máx. 250 W
Operación de colaboración	15 funciones avanzadas de seguridad regulables. Función de seguridad con certificación TÜV NORD Probado de acuerdo con las normas: EN ISO 13849:2008 PL d

Figura 3.1 [6]: Especificaciones de rendimiento del *UR3*

Hay que destacar aquí dos aspectos, el primero es que en caso de que las articulaciones se encuentren en uso continuado a elevadas velocidades, la temperatura de trabajo a la que el robot puede trabajar se reduce, y el segundo es que la norma *ISO* que cumple corresponde a su nivel de desempeño. Esto significa que las partes y elementos del robot que deben ser capaces de actuar en caso de una emergencia lo hacen bajo unas condiciones predecibles [35], sin embargo, puesto que este nivel se utiliza para reducir el riesgo en las funciones de seguridad, este estándar "requerido" ha de ser igual o superarse. Es por esto que existen distintos niveles de "superación" siendo *a* el más alto (menor riesgo) y *e* el más bajo. En el caso del *UR3* se puede ver que su nivel es PL d. Su probabilidad de fallos por hora es:

Tabla 2 [35]: Probabilidad de fallos peligrosos por hora en el caso de PL d.

Performance Level (PL)	Probabilidad de fallos peligrosos por hora
d	10-7y <10-6 (0.00001% al 0.0001%)

2. Especificaciones físicas

Especificación	
Carga útil	3 kg / 6,6 lb
Alcance	500 mm / 19,7 in
Grados de libertad	6 articulaciones giratorias

Figura 3.2: Especificaciones físicas del *UR3.*

Dentro de la carga útil se ha de considerar que el robot de por sí no cuenta con una herramienta. Por lo que, a la hora de diseñar las aplicaciones, si son de desplazar elementos dentro de un espacio, se ha de considerar el peso de los objetos y de la herramienta.

3. Especificaciones referentes al movimiento del *UR3*

Movimiento		
Movim. del eje del brazo robot.	Radio de acción	Velocidad máxima
Base	± 360°	± 180°/s
Hombro	± 360°	± 180°/s
Codo	± 360°	± 180°/s
Muñeca 1	± 360°	± 360°/s
Muñeca 2	± 360°	± 360°/s
Muñeca 3	Infinita	± 360°/s

Figura 3.3 [6]: Especificaciones del movimiento de las articulaciones del *UR3.*

En el caso de la muñeca 3, con el radio de acción infinito permite operaciones como atornillado o taladrado sin necesidad de añadir herramientas especiales para ello. Las velocidades que se indican son las máximas que permite el aparato, sin embargo, al ser demasiado alta ésta a la hora de realizar la aplicación se reducirá.

4. Funciones

Funciones		
Clasificación IP	IP64	
Clase ISO Sala limpia	5	
Ruido	70dB	
Montaje del robot	Todos	
Puertos de E/S en herramienta	Entrada digital	2
	Salida digital	2
	Entrada analógica	2
	Salida analógica	0
E/S de fuente de aliment. en herramienta	12 V / 24 V 600 mA en herramienta	

Figura 3.4 [6]: Funcionalidades que incluye el *UR3*

La clasificación IP viene dada por la UNE 20324 (derogada este año), y para el caso de este robot es 64, lo que quiere decir que el aparato es totalmente estanco al polvo (significado del 6) y que proyecciones de agua no tendrán efectos perjudiciales sobre el robot (significado del 4) [36]. El robot ha de estar en una sala limpia de clase 5 según la ISO 14644, y sus características son las siguientes: [37].

Tabla 3 [37]: Características de Sala Limpia clase 5.

	Límites máximos de concentración de partículas (p/m³ de aire) en m					
Tamaño partículas	>=0.1	>=0.2	>=0.3	>=0.5	>=1	>=5
ISO Clase 5	100000	23700	10200	3520	832	29

5. Características físicas

Características físicas	
Huella	Ø 128 mm
Materiales	Aluminio, plásticos de PP
Tipo de conector para herramientas	M8
Long. cable del brazo robótico	6 m / 236 in
Peso con cable	11 kg / 24,3 lb

Figura 3.5 [6]: Características físicas *UR3*

6. Características de los elementos complementarios al *UR3*.
Caja de control:

Funciones		
Clasificación IP	IP20	
Clase ISO Sala limpia	6	
Ruido	<65 dB (A)	
Puertos de E/S	Entrada digital	16
	Salida digital	16
	Entrada analógica	2
	Salida analógica	2
E/S de fuente de alimentación	24 V 2 A	
Comunicación	TCP/IP 100 Mbit, Modbus TCP, Profinet, EthernetIP	
Fuente de alimentación	100-240 V CA, 50-60 Hz	
Rango de temperatura ambiente	0–50°	
Características físicas		
Tamaño de la caja de control (an.×al.×la.)	475 × 423 × 268 mm / 18,7 × 16,7 × 10,6 in	
Peso	15 kg / 33,1 lb	
Materiales	Acero	

Figura 3.6 [6]: Características de la caja de control del *UR3*

3.1.2 Herramienta del robot.

La herramienta que se está utilizando para este proyecto está compuesta por una combinación de elementos. En primer lugar, está la *Wrist Camera* de *Robotiq* que se encuentra directamente acoplada a la muñeca 3 del robot. Sin embargo, para este proyecto puesto que no se puede utilizar *ROS* con este sensor, sólo interesan sus características físicas a la hora de definir el modelo de la herramienta [38]:

Specification	Value	
Maximum load	10 kg	40 Nm
Weight (without tool plate)	160 g	
Weight (with tool plate)	230 g	
Added height (without tool plate, for use with 2-Finger Gripper)	13.5 mm	
Global thickness (without tool plate)	22.4 mm	
Added height (with tool plate)	23.5 mm	
Global thickness (with tool plate)	29.5 mm	

Figura 3.7 [7]: Características físicas de la *Wrist Camera*

Además de estos parámetros, se han tenido en cuenta los valores de inercia que también se adjuntan en la información del catálogo del elemento y que pueden verse en la siguiente tabla.

Tabla 4: Parámetros de inercia de la *Wrist Camera*

Inertia Matrix			Metric value ($kg * mm^2$)		
Ixx	Ixy	Ixz	111	0	0
Iyx	Iyy	Iyz	0	70	3
Izx	Izy	Izz	0	3	165

Tras este elemento se ha adjuntado una pieza cilíndrica cuyo objetivo es alargar el alcance de la herramienta. Se asume, al no conocer más información que ha sido creada mediante impresión 3D y sus dimensiones son 60 mm de alto y 35 mm de radio.

Por último, como elemento manipulador se adhirieron las pinzas paralelas *Festo DHPS-20a*. De nuevo, de este elemento lo que más se ha utilizado han sido sus características físicas para la creación del modelo. En la figura 3.8 pueden verse representadas todas las dimensiones del elemento, sin embargo, las más importantes son las indicadas en la tabla a continuación.

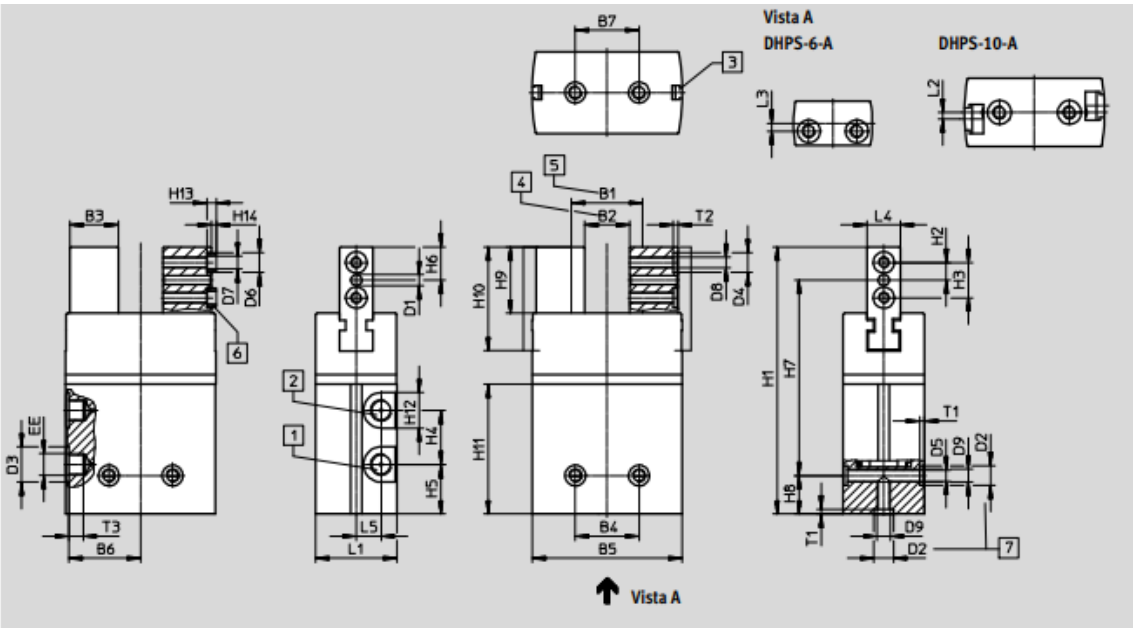

Figura 3.8 [8]: Planos pinzas *Festo DHPS.*

Tabla 5 [39]: Dimensiones de la pinza Festo DHPS-20a.

Dimensiones Festo DHPS-20a (en mm)	
B5	55,6
H1	101
L4	12
B2	17
H9	25
L1	30
B3	17,5
H10	39,5

3.1.3 Ordenadores

La generación de los programas se ha realizado desde un ordenador portátil Sony Vaio cuyas características son las siguientes:

Tabla 6 [40]: Características ordenador portátil.

Características Sony Vaio SVF1531B4E	
Fabricante del procesador	Intel
Tipo de procesador	Core i7 4500U
Velocidad del procesador	1.8 GHz
Toma del procesador	BGA1168
Número de procesadores	2
Capacidad de la memoria RAM	8 GB
Tecnología de la memoria	SO-DIMM, DDR3-SDRAM
Capacidad del disco duro	750 GB
Interfaz del disco duro	Serial ATA
Coprocesador gráfico	GeForce GT 740M
Descripción de la tarjeta gráfica	Nvidia GeForce GT 740M

De forma adicional se ha visto necesario, puesto que *ROS* sólo se encuentra disponible en sistemas operativos *Linux* instalar una partición de disco con *Ubuntu 16.04* y se ha trabajado en todo momento con la distribución *Kinetic* de *ROS.*

3.2 ENTORNO DEL LABORATORIO.

Es necesario realizar un breve comentario sobre el entorno en el que se encuentra el robot, puesto que parte del objetivo de este proyecto es definir dicho espacio en *Rviz* como objetos con los que puede existir colisión. Se supondrá que este entorno es fijo, es decir que no sufrirá cambios ni que los elementos que lo conforman se moverán de forma activa.

Como puede verse en la figura 3.9 que se adjunta a continuación, el robot se encuentra fijado en una plataforma elevadora y a su alcance hay dos objetos: una cinta transportadora y una mesa giratoria.

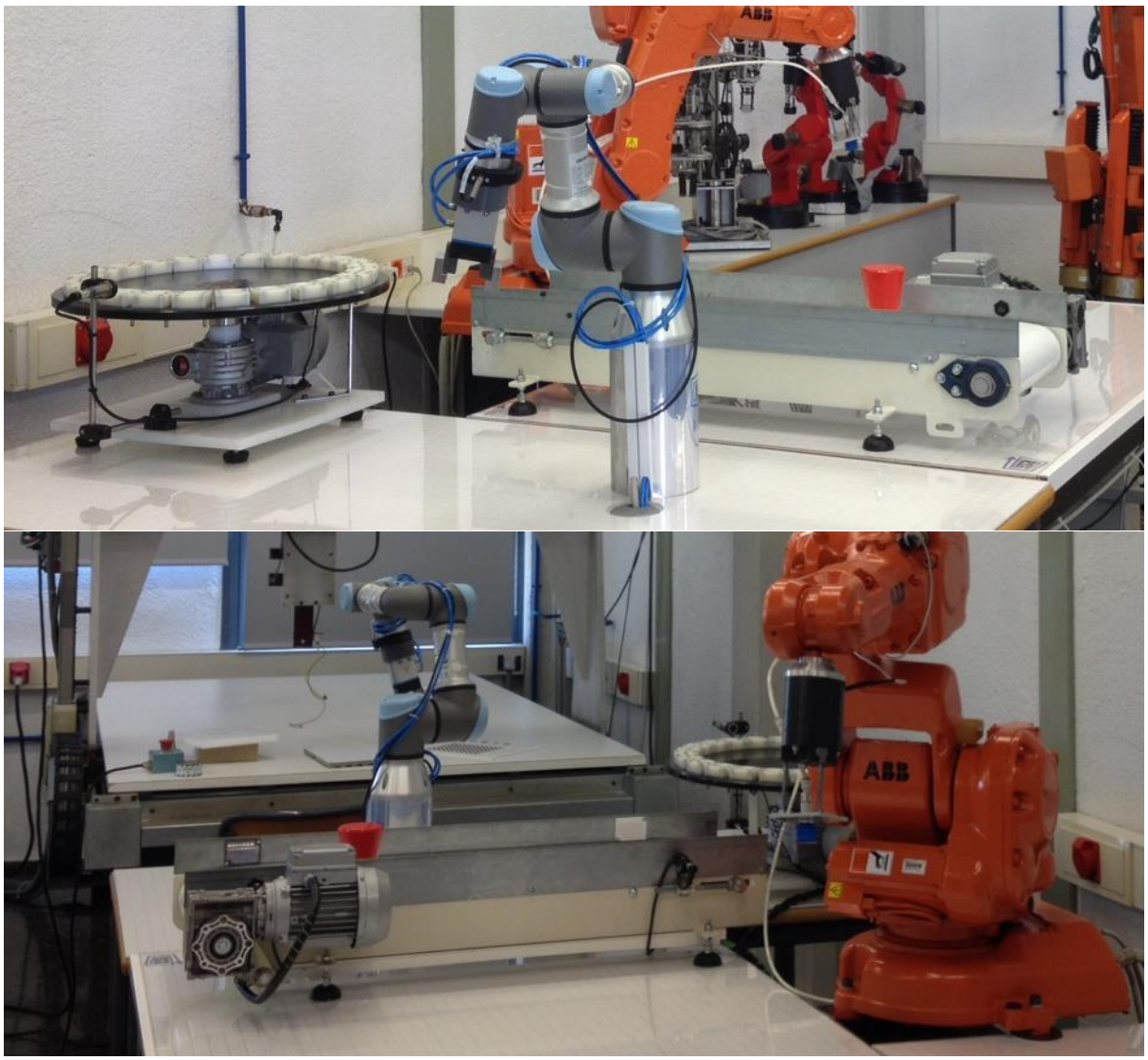

Figura 3.9: Entorno del laboratorio.

Las medidas con las que se trabaja en este proyecto son:

- Para el cilindro base: 128 mm de diámetro (coincidente con la huella del robot) y 212 mm de alto.
- Mesa giratoria: Se encuentra a aproximadamente 800 mm por detrás de la base del robot, tiene un diámetro de 550 mm y unos 270 mm de alto.
- Cinta transportadora: Se trata de una cinta de 270 mm de ancho, 1000 mm de largo, y la parte más elevada son 212 mm. Si se mide de centro de la base hasta el centro de la

cinta, ésta se encuentra a 450 mm a la derecha del robot, y se encuentra de forma que el punto medio a lo largo está en el mismo eje que el centro. Puede verse de forma esquemática en la figura 3.10.

Aquí se han tenido en cuenta otras medidas para darle al modelo el mayor realismo posible. Estas pueden verse en la figura 3.11 que corresponde con una representación esquemática de la vista frontal del entorno.

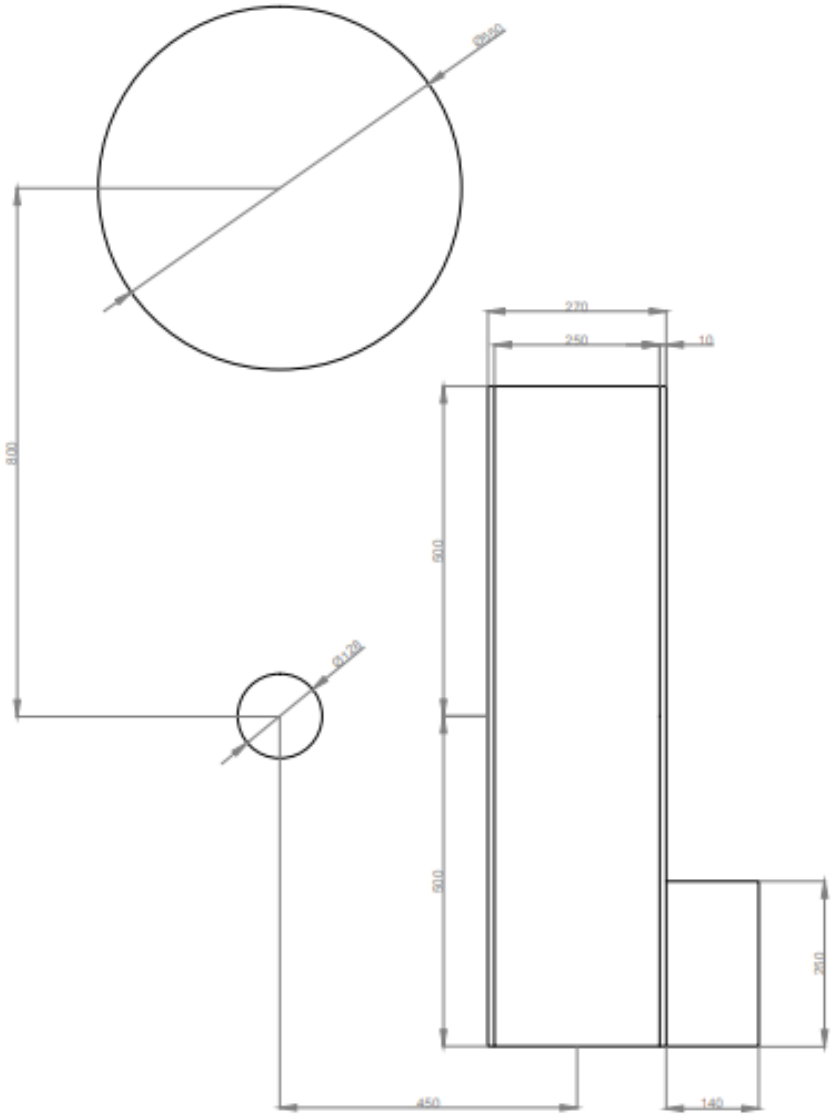

Figura 3.10: Esquema de la vista aérea con medidas del entorno.

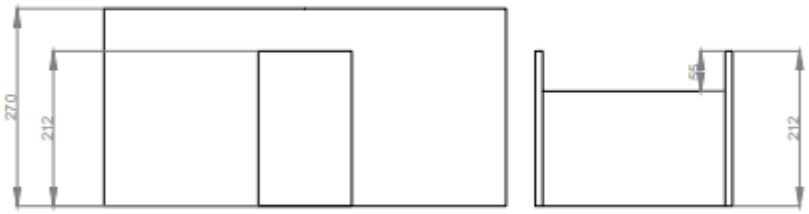

Figura 3.11: Esquema de la vista frontal con medidas del entorno.

3.3 PAQUETES DE ROS UTILIZADOS.

Como ya se ha comentado reiteradas veces a lo largo de este documento, una de las grandes ventajas que ofrece *ROS* es que se trata de un *software* colaborativo y libre, por lo que permite utilizar el trabajo previo de otra persona como punto de partida. Este hecho agiliza mucho el proceso de crear cualquier aplicación para un robot. Para la presente aplicación se han hecho uso de los siguientes paquetes ya creados:

- *Moveit!*
- *Universal_robots*
- *ur_modern_driver*
- *moveit_tutorials*

3.3.1 MoveIt!

En este paquete está basada completamente esta aplicación. Toda la información relevante respecto a él se encuentra desarrollada en el capítulo 2 de este documento.

3.3.2 Universal_robots

Este repositorio contiene toda la información relevante para poder controlar los robots desarrollados por *Universal Robots* en *ROS.* En él se pueden encontrar los distintos archivos de los modelos (se entrará a explicar más tarde en qué consisten y cómo funcionan) tanto del *UR3*, como del *UR5* y el *UR10* (*ur_description)*, y sus paquetes de configuración para poder ser controlados y simulados con *Moveit (urX_moveit_config)*. También aquí se encuentran los archivos necesarios para poder simular los robots antes mencionados en *Gazebo (ur_gazebo)*. A su vez, se pueden encontrar los paquetes con los *drivers* necesarios para conectarse y poder controlar el robot real desde *ROS* (*ur_driver* y *ur_bringup*), sin embargo, éstos sólo funcionan para versiones del software de *universal_robots* inferiores a la UR v3.x, que es justo el caso de este proyecto. Por último, este paquete incluye toda la información necesaria para que el robot pueda moverse como corresponde (*ur_kinematics)* y los complementos necesarios de *ROS* para la correcta comunicación entre los distintos nodos (*ur_msgs*).

3.3.3 *ur_modern_driver*.

Como se ha comentado antes, el paquete *universal_robots* sólo permite conectar con robots que cuentan con un *software* de versión inferior a la UR v3.x. El paquete *ur_modern_driver* sustituye al *ur_bringup* anteriormente mencionado y su función es precisamente que se pueda conectar con el robot real y controlarlo desde los programas de *ROS*. La versión de este paquete que existe actualmente es la correspondiente a la distribución *Indigo* de *ROS* y en un primer lugar no es compatible con la versión que se utiliza en este proyecto (*ROS Kinetic*). Sin embargo, si se realiza una modificación en uno de los nodos del paquete, este compila con *Kinetic* y funciona a la perfección. La modificación consiste en sustituir dentro del nodo *ur_hardware_interface.cpp* situado dentro de la carpeta *src* todos los "hardware_interface" que aparecen a lo largo del código sustituirlos por "claimed_resources.at(0).hardware_interface" [41].

3.3.4 moveit_tutorials

Este paquete contiene toda la información necesaria para aprender a utilizar todas las distintas funcionalidades de *MoveIt!* de forma sencilla. Se complementa con los distintos tutoriales que se pueden encontrar en: http://docs.ros.org/kinetic/api/moveit_tutorials/html/.
Sobre todo, en este proyecto se ha utilizado para, como se ha comentado, el aprendizaje de la herramienta y, posteriormente, para el desarrollo de los primeros códigos que generaban movimiento en el robot y la creación de los objetos del entorno, puesto que este paquete ya estaba configurado para acceder a todas las dependencias y paquetes externos necesarios para llevar a cabo estas funcionalidades. Una vez se consiguió la implementación exitosa dentro de este paquete, los nodos creados se movieron a sus correspondientes paquetes (se entrará en detalles sobre esto más adelante en el documento).

3.4 ACTUALIZACIÓN DEL MODELO

Como se ha explicado en apartados anteriores de este documento, es necesario tener un fichero que contenga el modelo del robot, o la información referente a este, y a su vez, como se ha comentado cuando se han explicado los paquetes que se van a utilizar, el modelo del *UR3* viene en el paquete *ur_description* dentro del compendio de *github* de *Universal_robots*. Pero debido a que en el entorno que se va a trabajar al robot se le ha añadido una herramienta, es necesario actualizar dicho modelo añadiendo todos los elementos que conforman este *end-effector.*

Partiendo del fichero *ur3.urdf.xacro* del paquete *ur_description*, se quiere añadir lo que se ha descrito en el apartado 3.1.2 como herramienta del robot. Para ello, en primer lugar, es necesario o bien realizar o bien encontrar un fichero de cad con los elementos descritos y las piezas que los componen que no se corresponden con formas geométricas primitivas, como es el caso de la *Wrist Camera* o la pinza hidráulica. Para el caso de la *Wrist Camera* como se trata de un elemento único sin piezas móviles, se puede trabajar con un único cad. Sin embargo para la pinza hidráulica se necesitan al menos dos archivos, puesto que existen partes móviles. Es por tanto que se tiene un fichero para el cuerpo macizo de la pinza hidráulica, y otro para representar los elementos móviles. Los modelos son los correspondientes a la figura 3.12 que sigue a continuación.

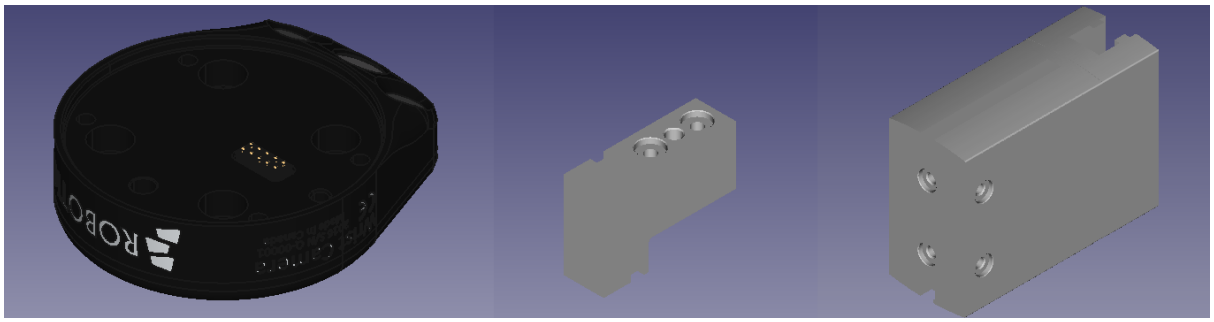

Figura 3. 12: Archivo CAD de los elementos a añadir al modelo del robot

Por último, para el cilindro que existe entre la *Wrist Camera* y la pinza se utilizará una figura primitiva cilíndrica.

Con esto, ya se entraría a complementar el modelo del robot dentro del fichero *ur3.urdf.xacro.* En primer lugar se elimina el *link* virtual que existe al final del código de nombre *${prefix}ee_link*, puesto que es en esta posición donde se van a añadir los nuevos elementos. Aplicando las etiquetas que se han explicado en el apartado 2.4.1 y las propiedades físicas que se encuentran en el apartado 3.1.2 que vienen en las especificaciones de los distintos aparatos y elementos que se añaden se puede definir la herramienta. Los elementos correspondientes a piezas se añaden de acuerdo con la siguiente sintaxis:

```
<link name="${prefix}camera_link">
        <visual>
        <geometry>
        <mesh filename="package://ur_grip/meshes/wristcam.stl" scale="0.01 0.01 0.01"/>
        </geometry>
        <material name="Black"/>
        </visual>
 <collision>
        <geometry>
        <mesh filename="package://ur_grip/meshes/wristcam.stl" scale="0.01 0.01 0.01"/>
        </geometry>
        </collision>
        <inertial>
        <mass value="0.23"/>
        <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>
        </inertial>
 </link>
```

En este caso, es interesante destacar lo siguiente: en primer lugar, que es necesario, por motivos de conversión, realizar un escalado de la pieza, y en segundo lugar llama la atención cómo se ha creado el nombre del elemento. Esto es debido a que gracias a las propiedades que ofrece el formato *xacro*, y gracias a *${prefix},* se puede utilizar el código de arriba para definir varios elementos en otros ficheros, en este caso puede ser utilizado también por *Gazebo* y habilita que el fichero *ur.transmission.xacro* encuentre la información. El código adicional que permite esto son las siguientes líneas:

```
 <xacro:ur_arm_transmission prefix="${prefix}" />
<xacro:ur_arm_gazebo prefix="${prefix}" />
```

Posteriormente, el código utilizado para la definición de las articulaciones del robot sigue la siguiente estructura:

```
joint name="${prefix}joint1" type="fixed">
        <parent link="${prefix}camera_link" />
        <child link = "${prefix}cylinder" />
        <origin xyz="0.0 0.0 0.0175" rpy="0.0 0.0 ${pi / 2.0}" />
</joint>
```

Sabiendo que ésta es la sintaxis que se va a utilizar, se añaden los 5 elementos adicionales que corresponden a la herramienta, que dentro del código se llaman *"${prefix}camera_link", "${prefix}cylinder", "${prefix}bloque", "${prefix}mano1"* y *"${prefix}mano2"* que se corresponden a la cámara, el cilindro alargador, el cuerpo de la pinza hidráulica y las dos manecillas que componen la pinza respectivamente. A su vez, se añaden las articulaciones y las uniones correspondientes que se conocen como *"${prefix}ee_link", "${prefix}joint1", "${prefix}joint2", "${prefix}joint3"* y *"${prefix}joint4"* que se corresponden a la articulación de la cámara con la muñeca del *UR3* (*ee_link*), la unión de la cámara con el cilindro (*joint1*), el cilindro con el cuerpo de la pinza hidráulica (*joint2*) y las articulaciones correspondientes al cuerpo de la pinza con las dos manecillas (*joint3* y *joint4*). SI se representa el árbol de transformadas del robot, se pueden observar todos los elementos que existían de forma previa, y cómo encajan los nuevos elementos añadidos dentro de éste. El árbol puede verse en la figura 3.13 adjunta a continuación:

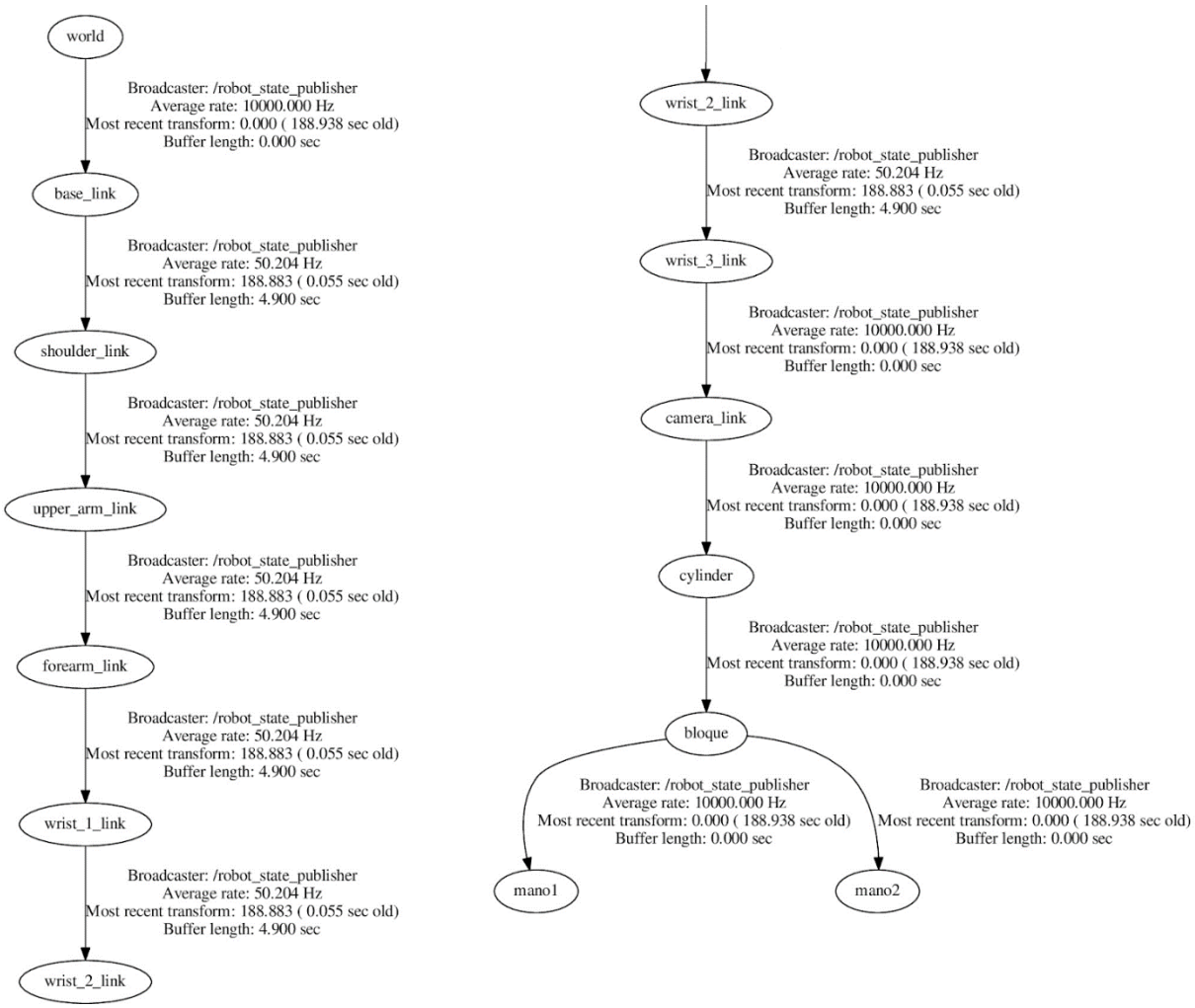

Figura 3.13: Árbol de transformadas del robot con herramienta

El aspecto final del modelo es el que se puede ver en la figura 3.14:

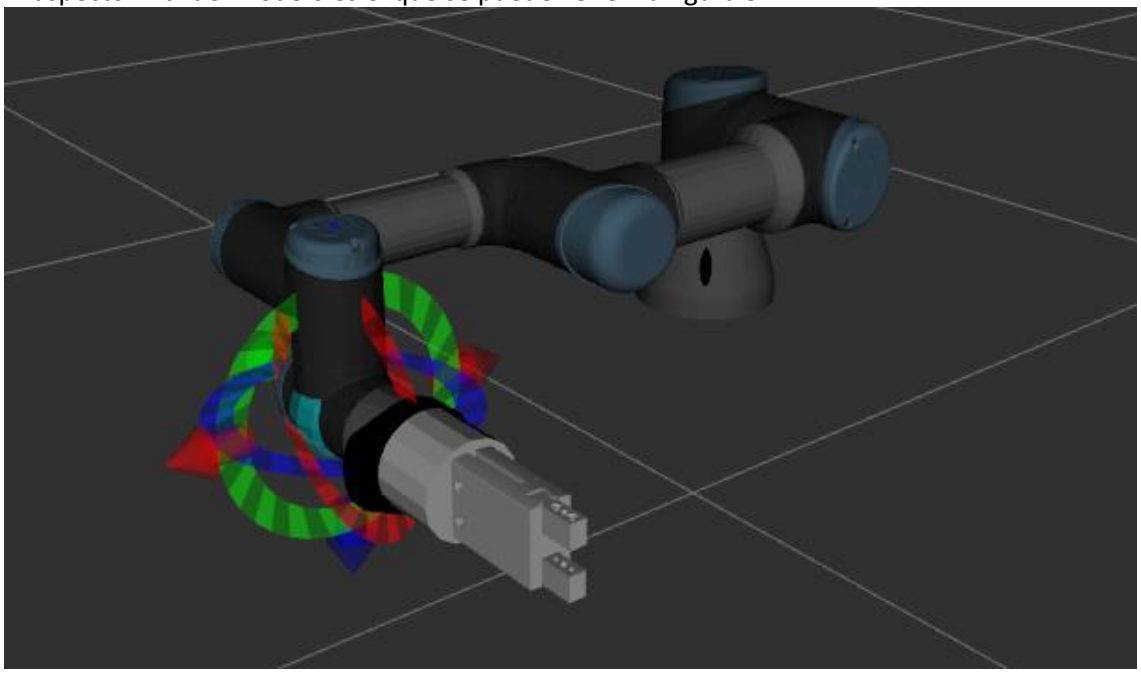

Figura 3.14: Modelo del *UR3* con la herramienta.

3.4.1 Creación del paquete que contiene el modelo

Se recomienda que estos ficheros que se han creado se almacenen todos dentro de un paquete de *ROS.* En este caso el nombre del paquete es *ur_description_mod* ya que se trata de una modificación del paquete *ur_description* ya existente.

En este directorio pueden encontrarse tres sub carpetas, el archivo *CMakeLists.txt* y el manifiesto del paquete *package.xml*. Se recomienda que dentro de la carpeta *meshes* se introduzcan en un directorio independiente (por ejemplo: *ur_grip)* los modelos CAD añadidos a los ficheros *urdf* modificados con el objetivo de que siempre puedan ser encontrados por el fichero a la hora de cargar el modelo. Por otra parte, en la carpeta *urdf* es donde se sitúan los ficheros *urdf* y *xacro* modificados, además de todos los ficheros complementarios que incluyen información adicional (como pueda ser *ur.transmission.xacro*). Puesto que se ha utilizado el paquete *ur_description* como base, dentro de estos ficheros habrá que cambiar todas las referencias que existan a dicho paquete por *ur_description_mod* para cargar el modelo adecuado. Y, por último, dentro del directorio *launch* se procede de igual manera, cambiando todos los argumentos que carguen elementos del paquete *ur_description* por *ur_description_mod*.

3.5 SIMULADORES

Dentro de *ROS* y en especial para el proyecto que se está llevando a cabo utilizando *MoveIt!* es necesario la utilización de simuladores que permitan realizar pruebas del software que se está creando, y en este caso que permitan hacer un diseño del entorno con el objetivo de que se pueda esquivar. Para el presente trabajo se han utilizado dos muy concretos en los que se entrará en detalle a continuación, que son *Gazebo* y *Rviz.*

3.5.1 *Rviz*.

Se trata de un visualizador de entornos 3D, su nombre proviene de *ROS visualization*. Como la mayoría de herramientas que provienen de *ROS* puede utilizarse con gran parte de los robots, y de no ser así el *set-up* para poder empezar a usarlo y la configuración concreta para la aplicación que se busca, se puede hacer y además existen numerosos tutoriales.

En este caso, *Rviz* se utiliza como interfaz gráfica donde se pueden ver los movimientos del robot en caso de que estos sean enviados mediante otro programa, el entorno que se ha definido, si se ha definido, y cómo interactúan dichos movimientos con el mencionado entorno. También permite el control manual de forma muy intuitiva de los movimientos del robot gracias a su *GUI.* Ésta permite mover manualmente el modelo del robot hasta una posición deseada mediante las flechas que se pueden ver en la figura 3.15 que se adjunta a continuación. Una vez se ha movido el robot hasta dicha posición, *MoveIt!* mediante los planificadores que se han incorporado, realiza un plan de la trayectoria, comprueba si ésta es posible o no, y la lleva a cabo. A su vez, es posible modificar una serie de parámetros, como inhabilitar la colisión con los objetos del entorno, o permitir que se realice una replanificación del movimiento o, incluso, elegir qué planificador utilizar de todos los que se han incluido a la hora de crear el paquete de *MoveIt!* de la descripción del robot (se entrará en esto más adelante).

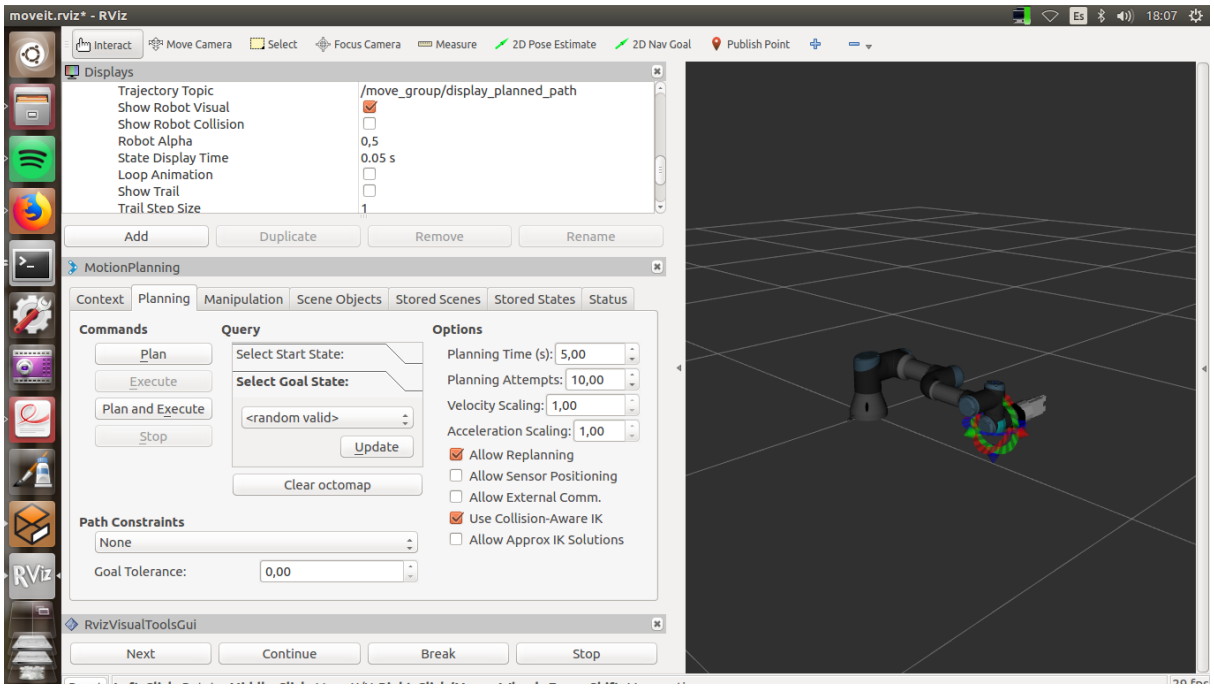

Figura 3.15: Entorno de la *GUI* de *Rviz.*

Rviz, además, contiene una serie de información relevante sobre todo a la hora de hacer simulaciones referente al origen de coordenadas de la representación del mundo, información sobre la posición de los distintos elementos que conforman el robot o distintas opciones de visualización (color del fondo, por ejemplo). Todos estos pueden encontrarse en la parte superior izquierda del menú del programa. De forma adicional, permite la posibilidad de añadir muchos tipos distintos de menús y opciones.

Sin embargo, a través de esta interfaz gráfica no pueden crearse los distintos objetos que pueden aparecer en el entorno, estos han de crearse de forma externa mediante programas en *C++* o *Python*, se hablará de ello más adelante en el documento.

Por último, hay que comentar que este programa se utiliza para enviar directamente los planes de movimiento que se han simulado y es capaz de enviar las posiciones finales. Por otra parte, el programa precisa de cargar la posición inicial y el modelo de alguna parte. Con este fin, *Rviz* se encuentra comunicado vía *topics* y *actions* con, o bien el robot real, o bien un modelo del robot que está siendo simulado en *Gazebo.*

3.5.2 Gazebo

Gazebo se trata de un simulador en el sentido más tradicional de la palabra, ya que lo único que permite es ver el robot en un entorno que puede ser personalizable, pero puesto que no influye en la trayectoria o el plan realizados, ya que el entorno de *Rviz* (que es donde se realizan los cálculos y las planificaciones) y el del propio programa no se pueden conectar. Sin embargo, sí que permite visualizar las trayectorias planificadas como si del robot real se tratara, y tiene, a nivel de comunicaciones un comportamiento muy similar al robot real. Su aspecto es el que puede verse en la figura 3.16 que se adjunta a continuación.

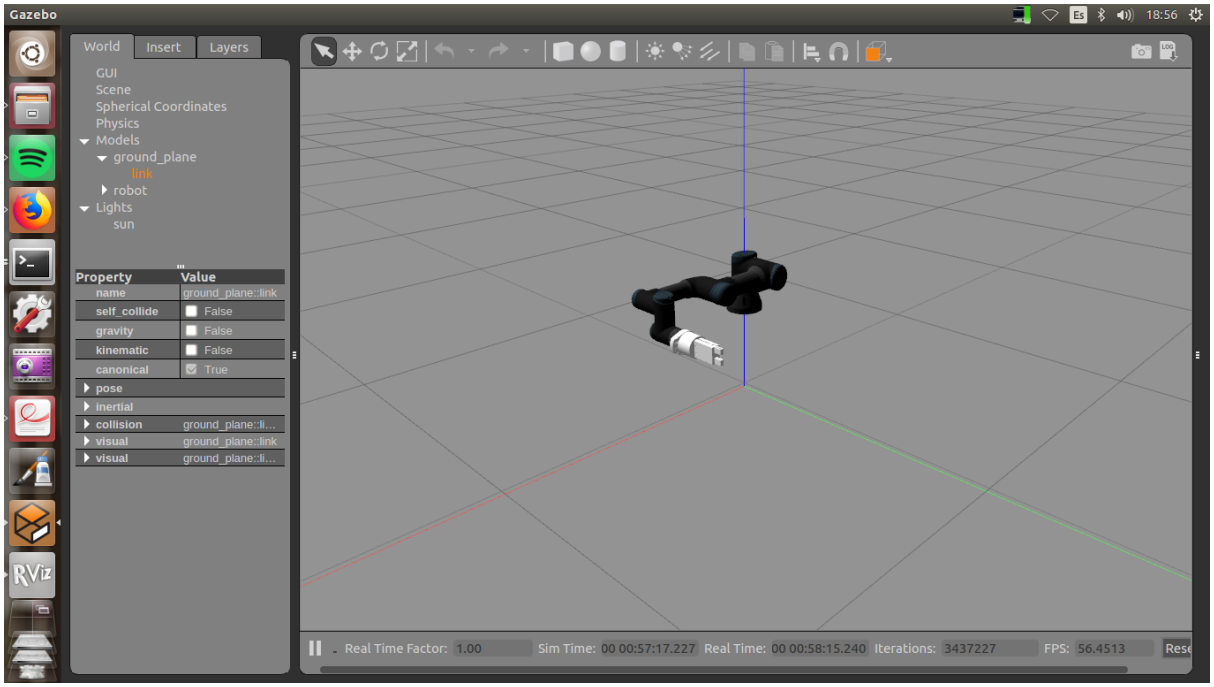

Figura 3. 16: Interfaz de *Gazebo*

Se trata, como *Rviz*, de una herramienta muy versátil, compatible con una gran cantidad de robots y fácilmente personalizable y configurable, puesto que permite el cambio de parámetros como puede ser la gravedad, las masas, inercias de los elementos que forman el robot, etc. Sin embargo, para los propósitos de este proyecto únicamente se ha utilizado como simulador para comprobar cuál era el comportamiento de los distintos programas que se han creado y cómo los movimientos del robot se adaptaban al entorno definido en *Rviz.*

3.6 CREACIÓN DEL PAQUETE DE *MOVEIT!* DEL NUEVO MODELO DEL ROBOT.

Tras la modificación del modelo original del *UR3* para añadir la herramienta del robot, es necesario crear un nuevo paquete de configuración de *MoveIt!* que permita realizar planes y controlar el robot con este *software*. Si se entra dentro de cualquier paquete de configuración al que se pueda acceder previamente, como puede ser *ur3_moveit_config* disponible dentro del compendio *universal_robots*, se puede ver que existen una gran cantidad de archivos de configuración y *launch* de *ROS*. Crear todos estos ficheros de forma manual a partir del modelo sería muy trabajoso, por suerte, *MoveIt!* cuenta con un asistente de *setup* muy similar a un instalador de programas de *Windows* que facilita mucho la tarea y que de forma adicional también crea el fichero *SRDF* que necesita *MoveIt!* para controlar el robot.

3.6.1 Asistente de *set up* de *MoveIt*

Si se lanza el siguiente comando en la terminal:

`roslaunch moveit_setup_assistant setup_assistant.launch`
se abriría el asistente con la pantalla que puede verse en la figura 3.17.

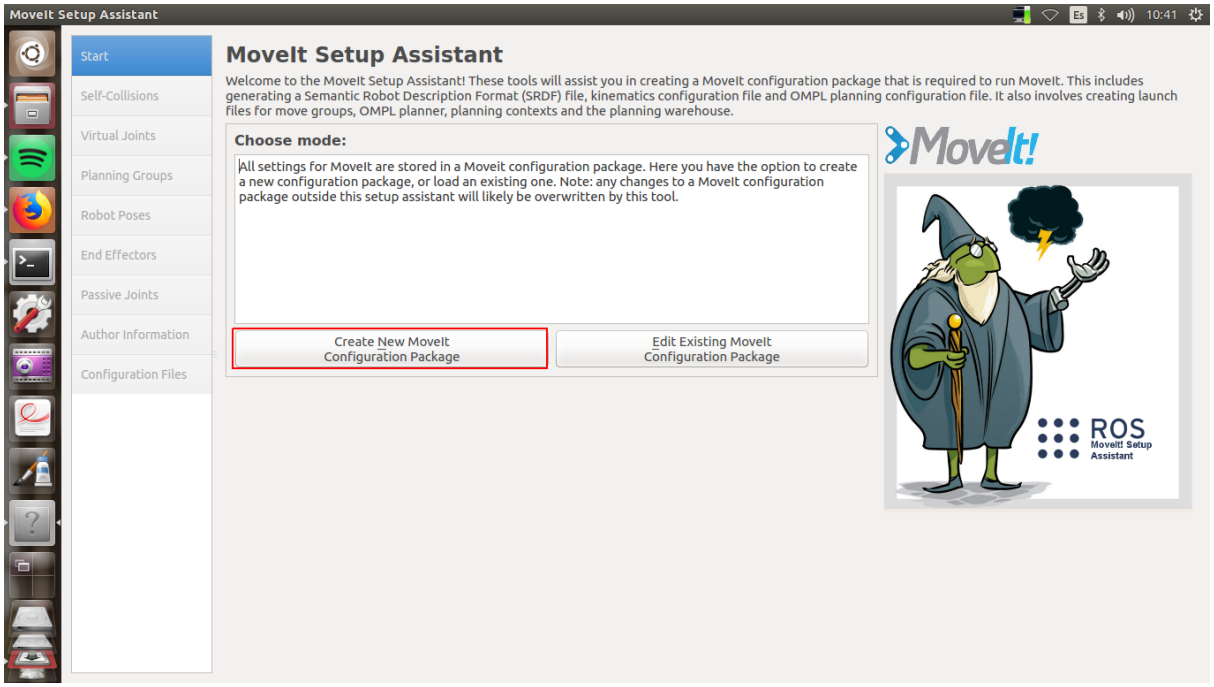

Figura 3.17: Pantalla de inicio del asistente de *setup* de *MoveIt!*

Aquí, como puede verse, existen dos opciones, o bien se crea un nuevo paquete para un modelo que se haya podido crear, o bien se actualiza un paquete ya existente. En este caso, se creará un paquete nuevo. Al seleccionar esta opción el programa pide que se seleccione el modelo del robot que se va a utilizar. En este caso, se utiliza el fichero *ur3_robot.urdf.xacro* ya que contiene toda la información del modelo del robot al estar referenciado el archivo *ur3.urdf.xacro* dentro de éste, e incluye información necesaria para el *setup* como una articulación virtual que conecta directamente la base del robot con el *frame* del mundo.

Una vez el modelo está subido al asistente, el panel que puede verse en la parte izquierda de la figura 3.17, se activa. Este menú permite configurar y definir los distintos aspectos del paquete

en cuestión. Si se selecciona la primera de todas las opciones: *self-collisions,* en la parte superior derecha se puede ver un botón en el que está escrito *Generate Collision Matrix* (recuadro azul en la figura 3.18). Esta es la matriz de la que se habló a la hora de explicar el funcionamiento de *MoveIt!* que indica qué partes del robot pueden generar colisión del robot consigo mismo. Al pulsar este botón, se genera la siguiente pantalla (recuadro rojo en la figura 3.18) en la que se puede ver la lista de los elementos que configuran la matriz junto con la justificación de por qué aparecen en la matriz.

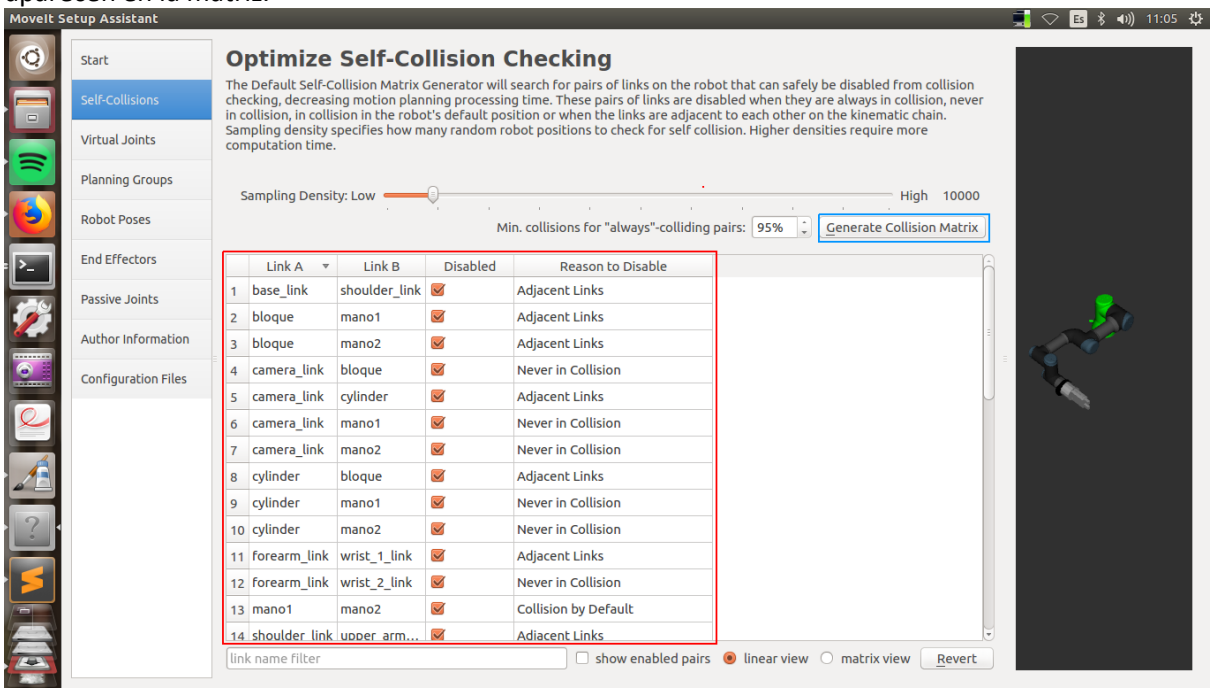

Figura 3.18: Pantalla de *Self-Collisions*.

Una vez se ha creado la matriz de colisiones, se puede pasar al siguiente menú. Se selecciona *Virtual Joints*. El objetivo de esta parte del *setup* es establecer una articulación fija virtual que conecte el mundo con el robot, así que con crear uno es suficiente. Para esto se selecciona el botón de la parte inferior izquierda en el que se puede leer *Add Virtual Joint*, y aparece el menú que puede verse en la figura 3.19. En este caso se crea una articulación entre el sistema de referencia del mundo (*world*) y el de la base del robot (*base_link*).

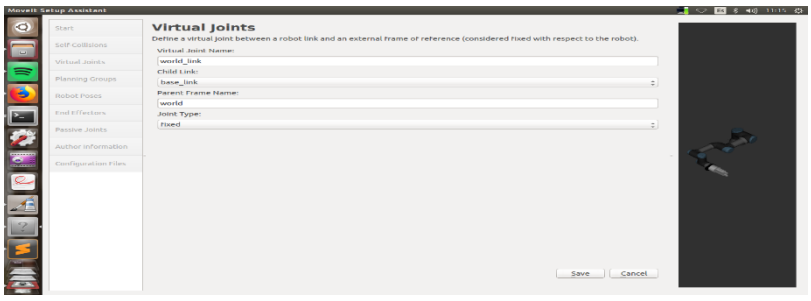

Figura 3.19: Menú *Add Virtual Joint*.

El siguiente paso, sería crear las agrupaciones de articulaciones que después se considerarán en los distintos planificadores de *MoveIt!*. Para el caso actual, se seleccionarán todas las articulaciones móviles que existen en este robot y se pondrán en un único grupo al que se llamará *manipulator*. Esto ayudará a que posteriormente se puedan reutilizar y modificar lo mínimo algunos de los ficheros de configuración del paquete *ur3_moveit_config.* Dentro del asistente, se seleccionaría la opción de *Planning Groups* y en la esquina inferior derecha se puede ver el botón *Add Group*. Una vez aquí, el menú pide darle un nombre al grupo, seleccionar

un *solver* de cinemática (se selecciona el *KDL*), el planificador por defecto dentro de todos los que están disponibles dentro del *OMPL* (este se deja el valor por defecto) y por último seleccionar las articulaciones del robot que formarán parte de este grupo. Aquí existen diversas opciones para seleccionar, se recomienda trabajar con cadenas cinemáticas puesto que, si éstas se han definido bien a la hora de realizar el *URDF*, es mucho más eficiente este *setup,* como puede verse en la figura 3.20. Se seleccionaría el *base_link* como primer eslabón de la cadena cinemática, y *wrist_3_link* como eslabón final. Se salva esta configuración, y ya se tendría el grupo configurado.

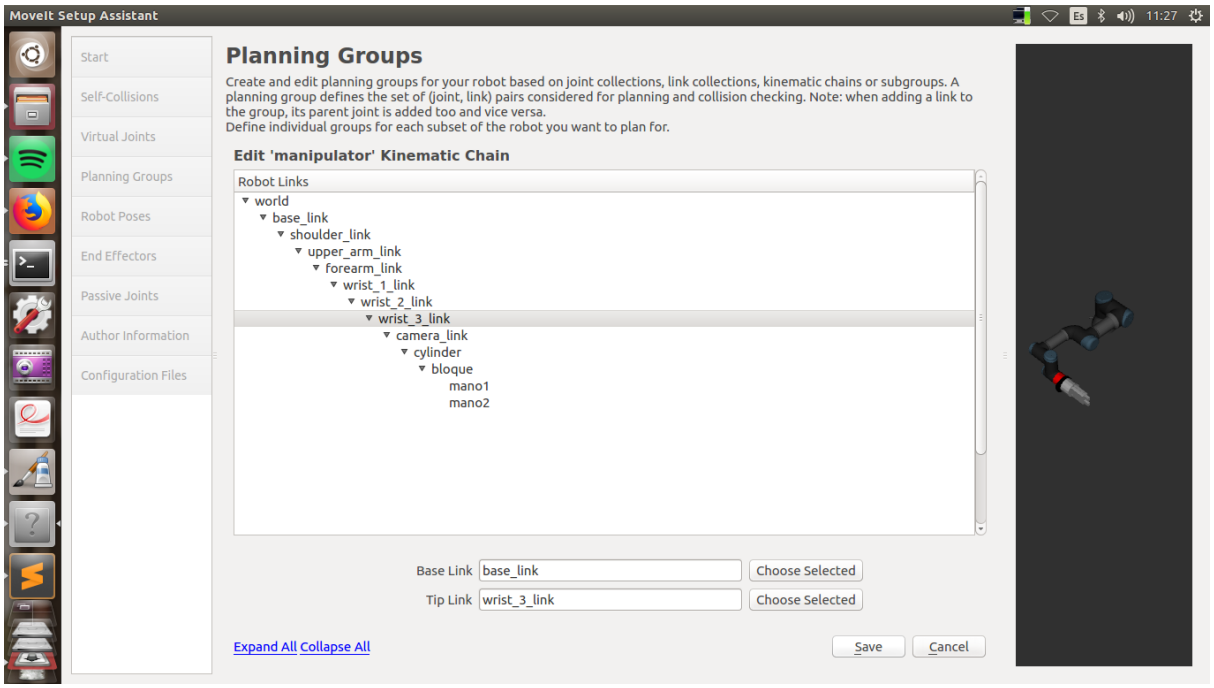

Figura 3. 20: Menú *Planning Groups*

De forma similar se seleccionan los 4 últimos joints en otro grupo que se corresponderá al *end-effector,* a pesar de que este no pueda moverse. Esto se hace porque cuando se llegue al apartado del menú correspondiente al *End-Effector* si no se selecciona un grupo que haga las veces de éste, el asistente no dejará finalizar la creación del paquete.

Una vez se tiene esto, se iría al siguiente menú, que corresponde a posiciones por defecto del robot. Aquí se puede mover el robot para guardar algunas posiciones básicas que luego pueden utilizarse en el menú gráfico de *Rviz.* Como puede verse en la figura 3.21, este menú permite darle un nombre a la posición, y modificar los valores de las articulaciones de forma manual con distintos *sliders*. También puede darse un valor determinado a dichas articulaciones a través de los distintos *displays*, e incluso permite cambiar de *Planning Group* entre los que han sido definidos en el apartado anterior. Aquí se recomienda guardar la posición de todas las articulaciones en valor 0 como *home* y un par de posiciones más en función de lo que se busque.

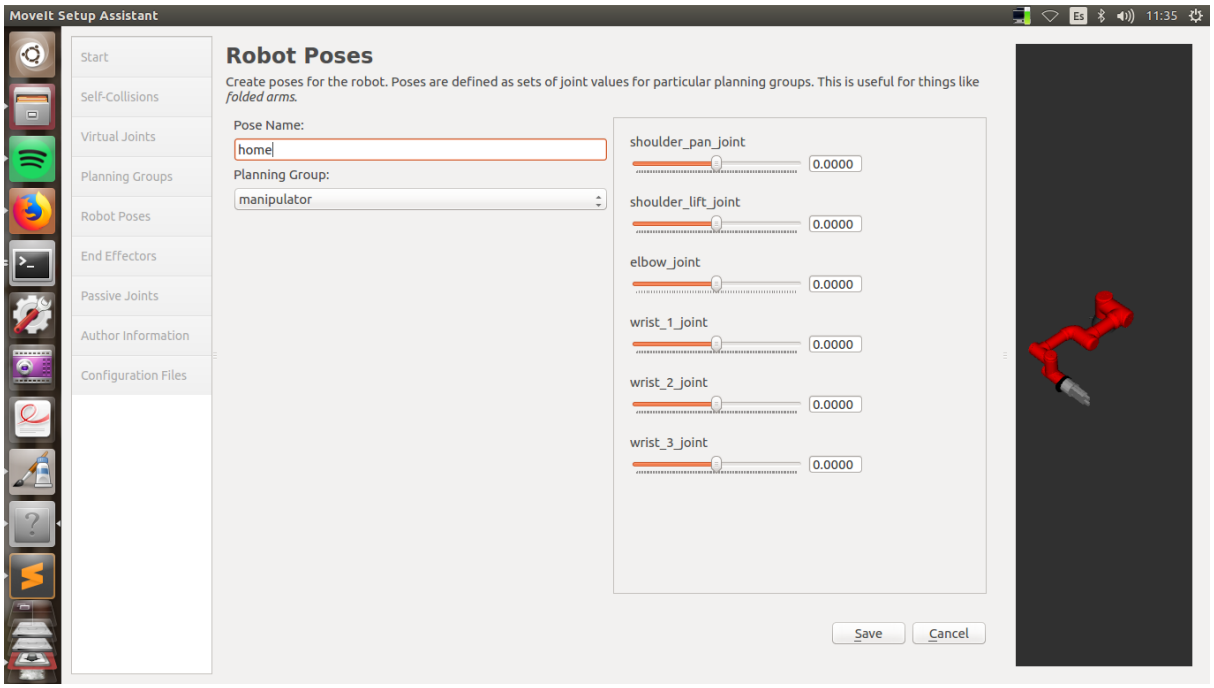

Figura 3.21: Menú *Robot Poses.*

Una vez se han seleccionado una serie de posiciones, se entra en el menú de *End-Effectors* y se añade la herramienta como *end-effector* del modelo. Aquí se tendría el menú que puede verse en la figura 3.22, donde simplemente se añade un nombre para el *end-effector* y se selecciona el *Planning Group* que corresponde a las piezas y articulaciones del robot que lo componen.

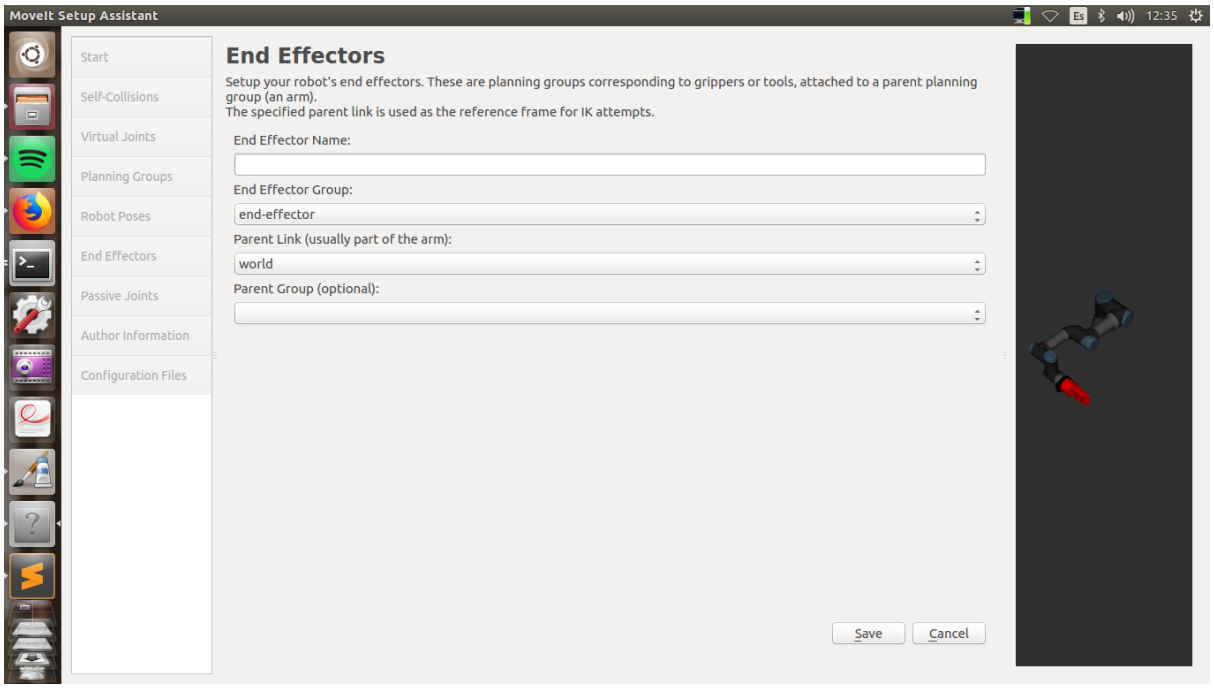

Figura 3.22: Menú *End Effectors.*

El menú correspondiente a los *Passive Joints* puede obviarse puesto que existe ninguna articulación que no pueda ser controlada por el robot y que por tanto deba ser omitido por *MoveIt!.* Así pues, sólo queda añadir la información del creador del paquete (nombre y e-mail) en la etiqueta *Author Information* y, por último, dentro de *Configuration FIles* seleccionar la carpeta en la que se desea crear el paquete (lo ideal sería dentro de la carpeta src del *workspace*

en el que se está trabajando) y seleccionar qué ficheros se busca crear en este caso todos los que pueden verse en la lista de la figura 3.23. Si se selecciona *Generate Package* el paquete se creará y con su finalización podrá cerrarse el asistente.

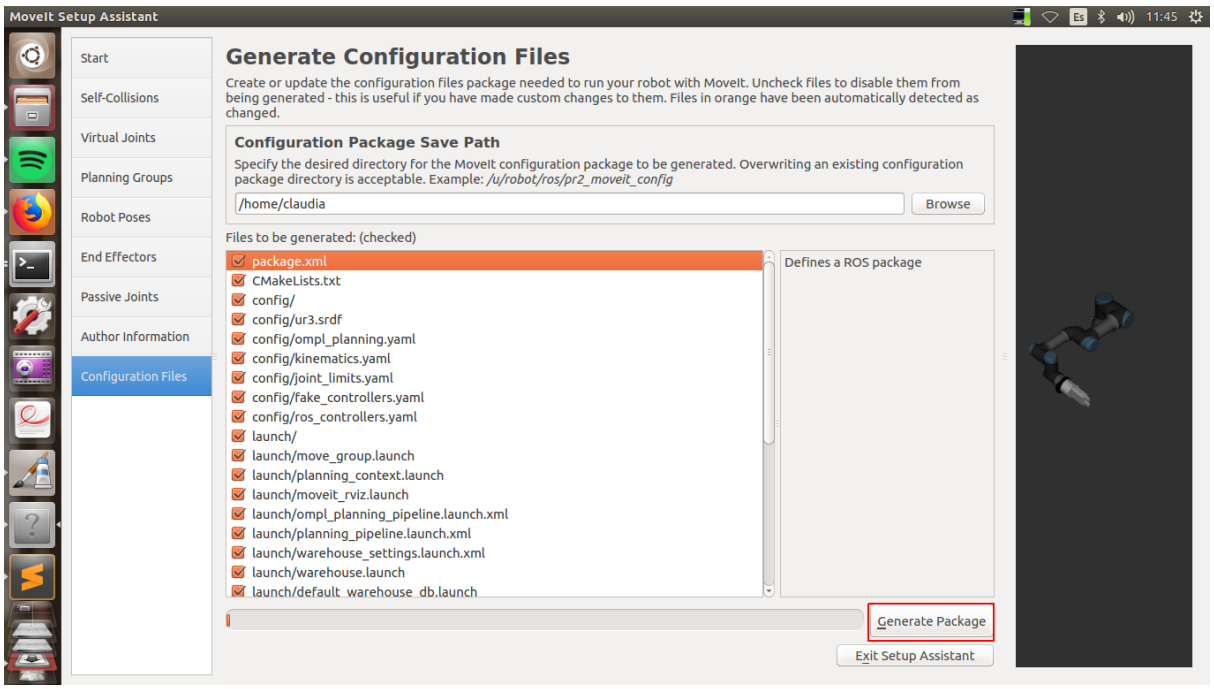

Figura 3.23: Menú *Configuration Files.*

3.6.2. Configurar el paquete para que pueda utilizarse con *Gazebo* y con el robot real.

Una vez se ha creado el paquete de *MoveIt!* es necesario hacer algunas modificaciones a algunos de los ficheros que se han creado con el fin de poder conectarlo tanto a *Gazebo* para realizar simulaciones, como al robot real para controlar directamente sus movimientos. Para esto se ha utilizado como guías y bases los paquetes ya mencionado anteriormente en este documento que venían incluidos dentro del repositorio *universal_robots: ur3_moveit_config* y *ur_gazebo*.

En primer lugar, se realizará una copia del paquete *ur_gazebo* y se ha modificado el nombre de este por *ur_gazebo_mod*. Para no tener problemas con el compilador por existir dos paquetes con el mismo nombre, también se ha modificado el fichero *CMakeLists.txt* y el manifiesto del paquete *package.xml*. Después, los archivos de configuración, que se encuentran en la carpeta *controller*, se dejarán tal cual están, todas las modificaciones se harán en la carpeta correspondiente a los *launch.*

Dentro de esta carpeta, las modificaciones se realizarán en: *ur3.launch, ur3_joint_limited.launch* y *controller_utils.launch.* La primera modificación consistirá en cambiar en los 3 ficheros cada vez que se quiere buscar un fichero dentro del paquete *ur_gazebo* sustituirlo por *ur_gazebo_mod*. Dentro del último *launch* de los listados no hay que hacer más modificaciones. En los otros dos, únicamente quedaría hacer referencia a que se carga el nuevo modelo creado y no el anterior. Es por tanto, que todas las instancias en las que se busca el paquete *ur_description* se sustituye por *ur_description_mod*. Con estas modificaciones, si se compila y se lanza en la terminal:
roslaunch ur_gazebo_mod ur3.launch

Se puede ver en la figura 3.24 que el modelo que se carga ya corresponde al que tiene la herramienta colocada.

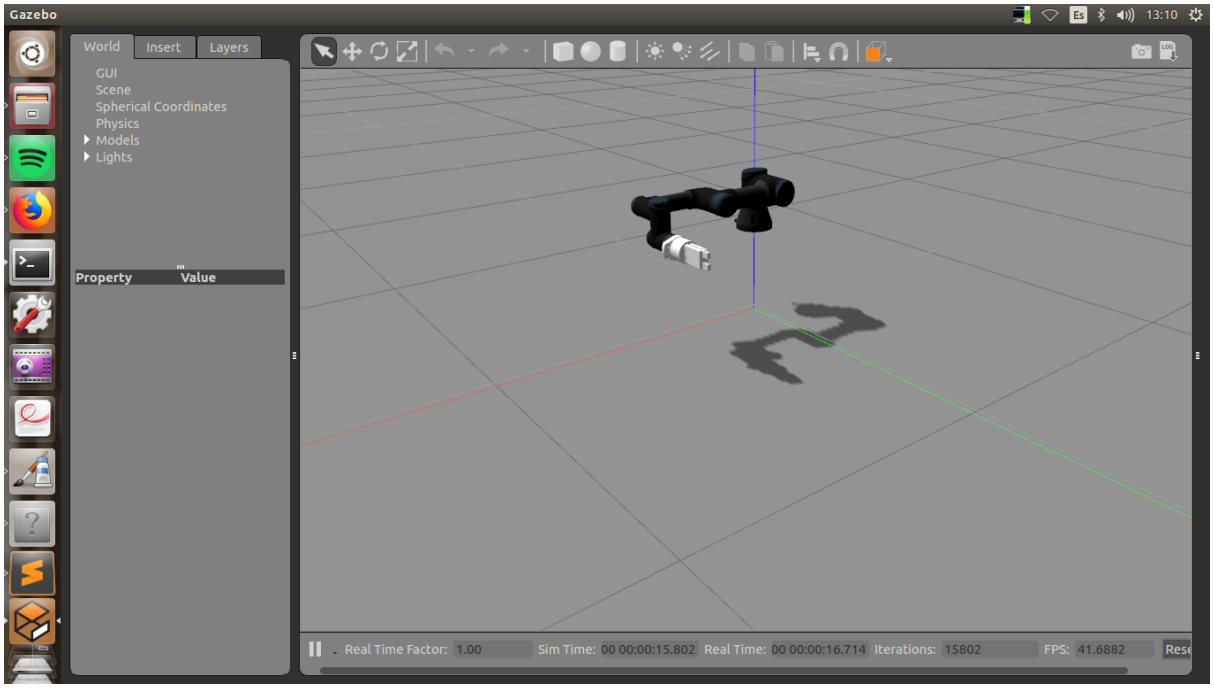

Figura 3. 24: Simulador *Gazebo* con el modelo del robot cargado.

Una vez se tiene esto, queda modificar algunos de los ficheros del paquete de configuración de *MoveIt!* que se ha creado para el nuevo modelo con el fin de que *MoveIt!* pueda enviar y recibir información de *Gazebo* para controlarlo y viceversa. Para conseguir esto es necesario modificar el fichero *move_group.launch*, los cambios que hay que llevar a cabo son:
Debajo de la primera línea que abre el fichero launch, añadir:

```
<arg name="limited" default="false"/>
```

Cerca del final del código, justo encima del siguiente código comentado:

```
<!-- Publish the planning scene of the physical robot so that rviz plugin can know actual robot -->
```

se añade lo siguiente:

```
       <!-- MoveGroup capabilities to load -->
      <param                                          name="capabilities"
value="move_group/MoveGroupCartesianPathService
            move_group/MoveGroupExecuteService
            move_group/MoveGroupKinematicsService
            move_group/MoveGroupMoveAction
            move_group/MoveGroupPickPlaceAction
            move_group/MoveGroupPlanService
            move_group/MoveGroupQueryPlannersService
            move_group/MoveGroupStateValidationService
            move_group/MoveGroupGetPlanningSceneService
            move_group/ClearOctomapService " />
```

Otro de los ficheros que debe modificarse es el que puede encontrarse bajo el nombre de: *ur3_moveit_controller_manager.launch.xml*. Si se abre para analizar el contenido tras la creación del paquete de configuración de *MoveIt!* puede verse que se trata de un launch vacío. Hay que añadir las siguientes líneas:

```
  <rosparam                                                file="$(find
ur3_grip_moveit_config)/config/controllers.yaml"/>
```

```
  <param name="use_controller_manager" value="false"/>
  <param     name="trajectory_execution/execution_duration_monitoring"
value="false"/>
  <param                                  name="moveit_controller_manager"
value="moveit_simple_controller_manager/MoveItSimpleControllerManager"
/>
```

Para terminar dentro de este directorio, es necesario crear un archivo *launch* con el nombre *ur3_grip_moveit_planning_execution.launch* con el siguiente contenido:

```
<launch>
  <arg name="sim" default="false" />
  <arg name="limited" default="false"/>
  <arg name="debug" default="false" />

  <!-- Remap follow_joint_trajectory -->
  <remap      if="$(arg      sim)"      from="/follow_joint_trajectory"
to="/arm_controller/follow_joint_trajectory"/>

  <!-- Launch moveit -->
  <include                                                    file="$(find
ur3_grip_moveit_config)/launch/move_group.launch">
      <arg name="limited" default="$(arg limited)"/>
      <arg name="debug" default="$(arg debug)" />
  </include>
</launch>
```

Por último, si se analizan las líneas que se han añadido en el fichero *ur3_moveit_controller_manager.launch.xml,* se puede ver que se carga un fichero de configuración llamado *controllers.yaml*. Sin embargo, este fichero no existe. Es, por tanto, imprescindible crear este fichero. El contenido del mismo para que cumpla su cometido ha de ser el siguiente:

```
      controller_list:
        - name: ""
            action_ns: follow_joint_trajectory
            type: FollowJointTrajectory
            joints:
            - shoulder_pan_joint
            - shoulder_lift_joint
            - elbow_joint
            - wrist_1_joint
            - wrist_2_joint
            - wrist_3_joint
```

Si se crean todos estos ficheros y se modifican como se indica los archivos ya existentes y se lanza el simulador o el robot real (cada uno con su comando) junto con las siguientes instrucciones (para el caso del simulador, por ejemplo):

roslaunch ur3_grip_moveit_config ur3_moveit_planning_execution.launch sim:=true

roslaunch ur3_grip_moveit_config moveit_rviz.launch config:=true

Se puede observar que es posible controlar tanto el robot como la simulación con la GUI de *Rviz.* En el anexo de instalación que se adjunta a esta memoria se pueden encontrar recomendaciones para reducir el número de archivos *launch* y argumentos a lanzar en cada caso.

3.7 DEFINICIÓN DEL ENTORNO DEL LABORATORIO EN *MOVEIT!*

Como se ha comentado en apartados anteriores, una de las características más destacables de *MoveIt!* es que puede planificar las trayectorias teniendo en cuenta su entorno. En otras palabras, permite que, si ésta se ha definido, los movimientos del robot se realicen sin colisionar con los objetos que definen su zona de trabajo. Esto resulta especialmente interesante dentro del marco de la robótica colaborativa, ya que permite que el entorno de trabajo compartido entre robots y personas resulte más seguro para los humanos.

El entorno del laboratorio en el que se encuentra el robot se encuentra descrito en este documento en el apartado 3.2, y lo que se busca es ser capaz de reproducir dicho espacio de una forma que *MoveIt!* pueda tener en cuenta. Como ya se comentó, también, a la hora de dar una breve introducción en las funcionalidades de *Rviz*, no es posible crear objetos desde la interfaz gráfica de *MoveIt!*, por lo tanto es necesario crear un programa en *C++* que sea capaz de representar los distintos objetos que configuran el laboratorio.

Pero antes, como puede verse en la figura 3.9 del apartado 3.2 de este documento, el robot se encuentra elevado respecto a la mesa en la que se encuentra. Esta mesa se considerará como el suelo de esta aplicación. Esto significa que el robot no debería encontrarse apoyado directamente aquí, sino que habría que situarlo por encima del plano que representa el suelo. Esta elevación, sin embargo, no puede llevarse a cabo en el mismo código que se utilizará para definir el entorno. Es por tanto que se debe recurrir a los ficheros de definición del modelo del robot. La modificación que habría de realizarse sería, dentro del fichero *ur3_robot.urdf.xacro* que existe dentro del paquete *ur_description_mod*, realizar un cambio en los parámetros de origen de la articulación virtual que se creó entre el robot y el mundo. Únicamente habría que modificar la coordenada *z* dándole un valor de 0.212 en lugar de 0. Con esto ya se puede entrar a definir el entorno.

El programa que define el entorno se trata de un código C++ incluido en un paquete de creación propia que está basado en el paquete creado para seguir la formación de los tutoriales que ofrece la página oficial de *MoveIt!.* Así pues, toda la configuración del paquete se ha realizado en función de las dependencias que necesitaba este paquete de tutoriales.

Antes de entrar como tal en el código que define el espacio, se realizará un breve comentario comentando las aproximaciones que se van a realizar con los objetos que se busca dibujar. Puesto que el código C++ con el que trabaja *MoveIt!* solo permite definir objetos de forma externa mediante formas primitivas, es necesario descomponer los objetos que se tienen y definirlos con cilindros, esferas y prismas de base rectangular. En primer lugar, la cinta transportadora, dada su forma y por simplificar, se definirá como 3 prismas de base rectangular (las dimensiones de todos los elementos están especificadas en el código que se explicará más adelante en el documento) que se encuentran juntos, la base del robot se trata de un elemento cilíndrico y, por último, el elemento que se encuentra detrás del robot se definirá por simplicidad como un gran cilindro.

El código que permite la definición del entorno fijo del laboratorio se adjunta a continuación, con incisos que explican qué funciones realiza cada parte del programa:

```
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
```

```
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>

#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

#include <moveit_visual_tools/moveit_visual_tools.h>
```

Esto son los distintos ficheros de tipo *header* que han de incluirse para que las funciones de *MoveIt!* que crean los objetos funcionen.

```
int main(int argc, char** argv)
{
  ros::init(argc, argv, "entorno_def");
  ros::NodeHandle node_handle;
  ros::AsyncSpinner spinner(1);
  spinner.start();
```

Declaración del *main* (este programa no consta de funciones auxiliares) y líneas de código necesarias para inicializar el programa en *ROS*: inicialización del nodo con su nombre y definición del elemento conocido como *Node Handler.* Se trata de un objeto que representa al nodo de *ROS* dentro del código, siempre ha de crearse uno al inicio de los nodos, y se utiliza en multitud de funcionalidades.

```
 static const std::string PLANNING_GROUP = "manipulator";
moveit::planning_interface::MoveGroupInterface
move_group(PLANNING_GROUP);
moveit::planning_interface::PlanningSceneInterface  planning_scene_int
erface;
  const robot_state::JointModelGroup* joint_model_group =
      move_group.getCurrentState()-
>getJointModelGroup(PLANNING_GROUP);
```

Aquí, en primer lugar, se define *"manipulator"* como un *string* llamado *PLANNING_GROUP* puesto que todas las operaciones de este código se harán respecto a este grupo de articulaciones, y en caso de querer reutilizar el código es más sencillo sólo cambiar este nombre. Posteriormente se construye el cliente de la *action* de *MoveGroup* para un determinado grupo de articulaciones y de forma análoga para escena con *PlanningSceneInterface*. Por último, aquí se almacena el estado en el que se encuentra de forma inicial el robot o el simulador para enviarlo al simulador y que este se inicie en dicha posición.

```
  namespace rvt = rviz_visual_tools;
  moveit_visual_tools::MoveItVisualTools visual_tools("/world");
  visual_tools.deleteAllMarkers();
  visual_tools.loadRemoteControl();
  visual_tools.trigger();
```

Esta parte del código se encarga de inicializar las llamadas *rviz_visual_tools*, que son las funcionalidades de *MoveIt!* gracias a las cuales se pueden definir distintos elementos en el entorno, como por ejemplo mensajes. Esta funcionalidad ayuda mucho a limpiar el código de errores. Se crea el objeto *visual_tools* asociado al frame *"/world".* Este es el sistema de referencia al que estarán asociados los elementos que se creen que pertenezcan a esta clase.

```
moveit_msgs::CollisionObject centro_cinta;
centro_cinta.header.frame_id = move_group.getPlanningFrame();
centro_cinta.id = "centrocinta";
geometry_msgs::Pose cinta1_pose;
shape_msgs::SolidPrimitive primitive_cinta;
 std::vector<moveit_msgs::CollisionObject> objetos_fijos;
```

Esta sería la primera definición de objeto que se encontraría en el código. Se refiere al prisma que definirá la parte correspondiente al centro de la cinta. Este procedimiento de definición se repetirá para todos los objetos que se creen. En primer lugar se define el elemento con la clase *moveit_msgs::CollisionObject*. Posteriormente, se asocia al mismo sistema de referencia al que se está haciendo la planificación del grupo de articulaciones que se está utilizando. A su vez, también se le da un nombre utilizando el argumento *id*. Lo siguiente que queda hacer es definir un objeto *Pose* en el que se almacenarán los datos de posición del elemento, y otro objeto de tipo *SolidPrimitive* en el que se le dice qué tipo de objeto es y sus dimensiones (esto se verá un poco más adelante).
Por último, en este fragmento de código se ve que se define un vector de *CollisionObject* llamado *objetos_fijos*, esto se debe a la forma de publicar estos objetos en *MoveIt!* (se entrará en detalles cuando se realice esta acción en el código).

```
moveit_msgs::CollisionObject cilindro_base;
cilindro_base.header.frame_id = move_group.getPlanningFrame();
cilindro_base.id = "cilindrobase";
geometry_msgs::Pose cylinder_pose;
shape_msgs::SolidPrimitive cylinder_primitive;

moveit_msgs::CollisionObject cilindro_detras;
cilindro_base.header.frame_id = move_group.getPlanningFrame();
cilindro_base.id = "cilindrodetras";
geometry_msgs::Pose cylinder_pose2;
shape_msgs::SolidPrimitive cylinder_primitive2;

moveit_msgs::CollisionObject derecha_cinta;
derecha_cinta.header.frame_id = move_group.getPlanningFrame();
derecha_cinta.id = "derechacinta";
geometry_msgs::Pose cinta2_pose;
shape_msgs::SolidPrimitive primitive_cinta2;

moveit_msgs::CollisionObject izquierda_cinta;
izquierda_cinta.header.frame_id = move_group.getPlanningFrame();
izquierda_cinta.id = "izquierdacinta";
geometry_msgs::Pose cinta3_pose;
shape_msgs::SolidPrimitive primitive_cinta3;
```

Como puede verse, aquí se ha realizado el mismo procedimiento, y prácticamente se trata del mismo código para el resto de objetos que definen el entorno. Se han definido en este orden: el cilindro de la base, el cilindro que se encuentra detrás del robot y los dos prismas complementarios que terminan de definir la cinta transportadora.

```
primitive_cinta.type = primitive_cinta.BOX;
primitive_cinta.dimensions.resize(3);
primitive_cinta.dimensions[0] = 1.0;
```

```
primitive_cinta.dimensions[1] = 0.25;
primitive_cinta.dimensions[2] = 0.155;

cinta1_pose.orientation.w = 1.0;
cinta1_pose.position.x = 0.0;
cinta1_pose.position.y = 0.45;
cinta1_pose.position.z = 0.0775;

centro_cinta.primitives.push_back(primitive_cinta);
centro_cinta.primitive_poses.push_back(cinta1_pose);
centro_cinta.operation = centro_cinta.ADD;
objetos_fijos.push_back(centro_cinta);
```

Aquí es donde ya se realiza la definición completa de los elementos que se han creado anteriormente. Como puede verse se coge el objeto de *SolidPrimitive* que se creó para el centro de la cinta y se define en la primera línea como un prisma (*BOX*). Después, *dimensions* se trata de un vector en el que se le dan las dimensiones en *x, y, z*. Con esto ya se ha definido el objeto. Ahora se define la posición a través del objeto *Pose* creado junto con el objeto. Aquí se le dan las coordenadas en *x, y, z* que posicionan el centro geométrico del objeto, y la orientación que este tendrá respecto de los ejes.
Una vez se han definido estos, hay que asociar estos objetos del código al objeto que se ha creado, porque hasta ahora se han definido de forma independiente. Para esto lo que se hace es, se trata el objeto *centro_cinta* que se definió como un vector en *C++* y las objetos de forma y posición que se han creado como parámetros que quieren añadirse al final del mismo. Una vez se ha hecho esto, al objeto se le aplica la operación de *ADD* y se añade al vector de objetos que es el que se utilizará para añadir los elementos al entorno.

```
cylinder_primitive.type = cylinder_primitive.CYLINDER;
cylinder_primitive.dimensions.resize(3);
cylinder_primitive.dimensions[0] = 0.212;
cylinder_primitive.dimensions[1] = 0.064;

cylinder_pose.orientation.w = 1.0;
cylinder_pose.position.x = 0.0;
cylinder_pose.position.y = 0.0;
cylinder_pose.position.z = 0.106;

cilindro_base.primitives.push_back(cylinder_primitive);
cilindro_base.primitive_poses.push_back(cylinder_pose);
cilindro_base.operation = cilindro_base.ADD;
objetos_fijos.push_back(cilindro_base);
```

Análogo al elemento anterior, pero en este caso se define el cilindro que corresponde a la base del robot.

```
//Tamaño y posición cilindro detrás
cylinder_primitive2.type = cylinder_primitive2.CYLINDER;
cylinder_primitive2.dimensions.resize(3);
cylinder_primitive2.dimensions[0] = 0.27;
cylinder_primitive2.dimensions[1] = 0.225;

cylinder_pose2.orientation.w = 1.0;
cylinder_pose2.position.x = -0.8;
```

```
  cylinder_pose2.position.y = 0.0;
  cylinder_pose2.position.z = 0.135;

  cilindro_detras.primitives.push_back(cylinder_primitive2);
  cilindro_detras.primitive_poses.push_back(cylinder_pose2);
  cilindro_detras.operation = cilindro_detras.ADD;
  objetos_fijos.push_back(cilindro_detras);
```

Igual que los anteriores, pero en este caso es el cilindro que se encuentra en la parte posterior del entorno de trabajo del robot.

```
  primitive_cinta2.type = primitive_cinta2.BOX;
  primitive_cinta2.dimensions.resize(3);
  primitive_cinta2.dimensions[0] = 1.0;
  primitive_cinta2.dimensions[1] = 0.01;
  primitive_cinta2.dimensions[2] = 0.21;

  cinta2_pose.orientation.w = 1.0;
  cinta2_pose.position.x = 0.0;
  cinta2_pose.position.y = 0.32;
  cinta2_pose.position.z = 0.105;

  derecha_cinta.primitives.push_back(primitive_cinta2);
  derecha_cinta.primitive_poses.push_back(cinta2_pose);
  derecha_cinta.operation = derecha_cinta.ADD;
  objetos_fijos.push_back(derecha_cinta);

// Tamaño y posición lado izquierdo cinta
  primitive_cinta3.type = primitive_cinta2.BOX;
  primitive_cinta3.dimensions.resize(3);
  primitive_cinta3.dimensions[0] = 1.0;
  primitive_cinta3.dimensions[1] = 0.01;
  primitive_cinta3.dimensions[2] = 0.21;

  cinta3_pose.orientation.w = 1.0;
  cinta3_pose.position.x = 0.0;
  cinta3_pose.position.y = 0.58;
  cinta3_pose.position.z = 0.105;

  izquierda_cinta.primitives.push_back(primitive_cinta3);
  izquierda_cinta.primitive_poses.push_back(cinta3_pose);
  izquierda_cinta.operation = izquierda_cinta.ADD;
  objetos_fijos.push_back(izquierda_cinta);
```

Procedimiento similar a los anteriores, pero en este caso se han definido los dos extremos de la cinta transportadora.

```
  planning_scene_interface.addCollisionObjects(objetos_fijos);
  ros::shutdown();
  return 0;
}
```

Por último, dentro de lo que es el programa, se tiene el comando que toma el vector de *CollisionObjetc*s que se ha ido rellenando según se han ido creando los distintos objetos del entorno y se publica en el *Planning Scene Interface*.
Posteriormente, se cerrarían las funcionalidades de *ROS* en el código y se cierra el *main* del programa.

Con todo esto, el resultado de la definición del entorno del robot es la que puede verse en las figuras 3.25 y 3.26:

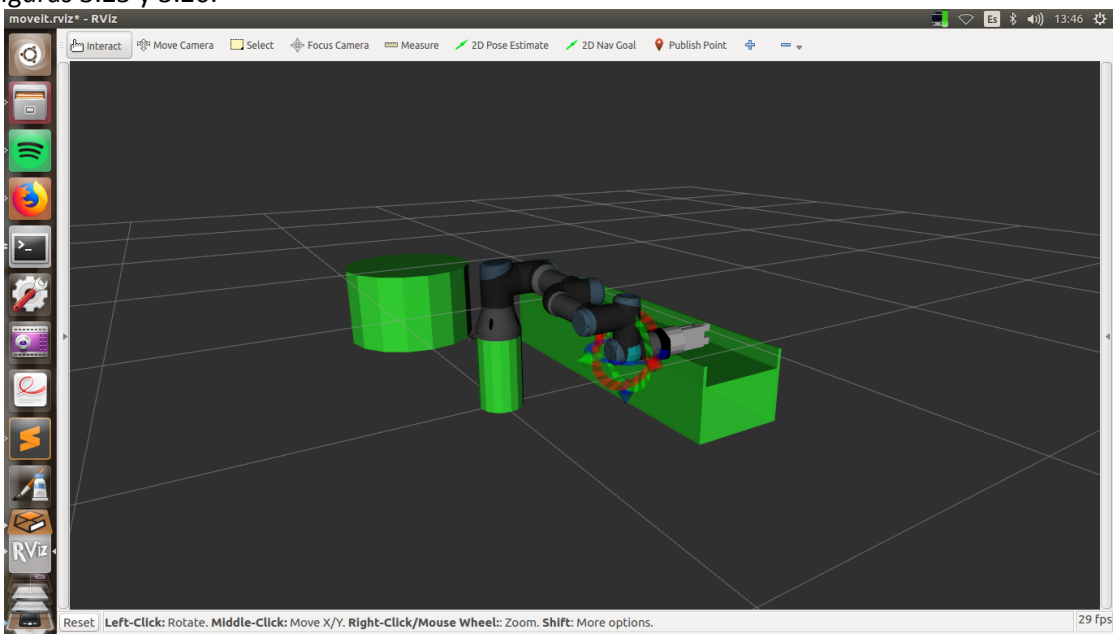

Figura 3. 25: Vista lateral del entorno virtual.

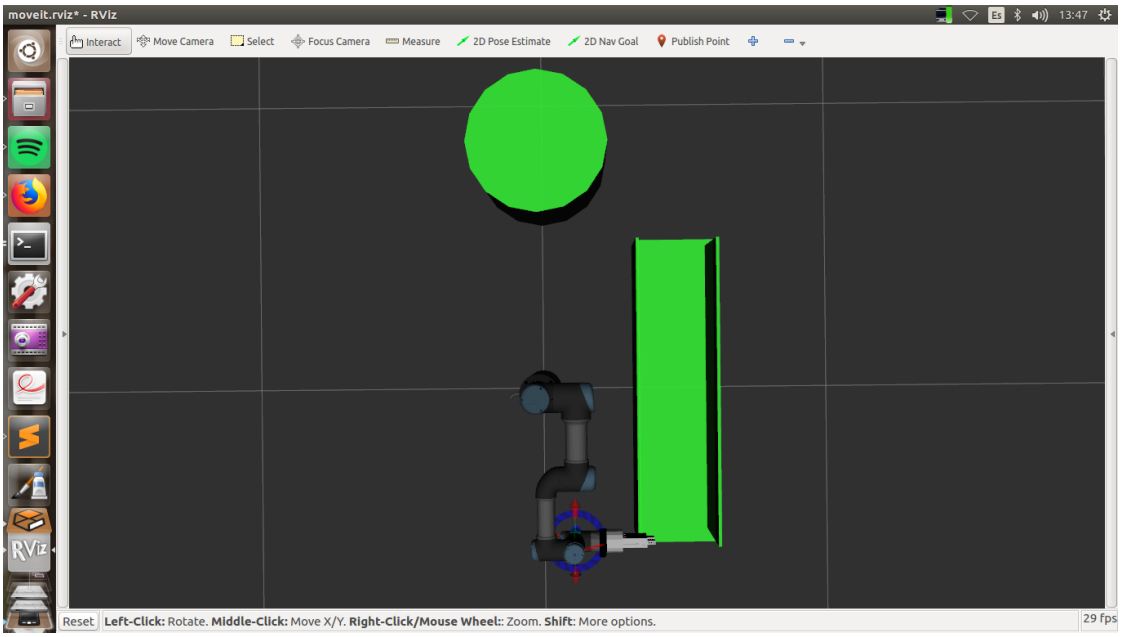

Figura 3. 26: Vista superior

3.8 PROGRAMAS DE SEGUIMIENTO DE OBSTÁCULOS

Ya se ha visto cómo definir el entorno de trabajo del robot lo que facilita que éste pueda moverse por este entorno evitando estos objetos que conforman su espacio. Ahora, de forma adicional, se han realizado dos programas que permiten que una cámara realice el seguimiento de un objeto, permitiendo así que el robot pueda esquivar objetos en movimiento en tiempo real.

El primer programa realiza los ajustes de los parámetros para el filtro de la cámara, esto es, se selecciona qué objeto se va a seguir en función de su color, y la calibración de la misma. La segunda aplicación es la que realiza el seguimiento del objeto y envía al programa que crea el objeto en el espacio las coordenadas (ya convertidas) del centroide del objeto que se ha detectado.

Para la creación de estos programas se han utilizado librerías tanto de *ROS* como de *OpenCV* ya que se va a trabajar, como se ha dicho con una cámara a tiempo real. *OpenCV* se trata de un conjunto de librerías de visión artificial. Se trata de bibliotecas de código libre desarrolladas en primer lugar por *Intel* en 1999 y que posteriormente fueron ampliadas y desarrolladas por los usuarios [49]. Actualmente permite la toma y procesado de imágenes (fotografías) y puede utilizarse con vídeos, tanto offline como vídeo a tiempo real.

Esta biblioteca está disponible en un gran número de lenguajes de programación siendo los más utilizados *C++ o Python*, sin embargo, también es compatible con *Java* e incluso con *Matlab* y *Octave.* Puede utilizarse en sistemas operativos con base *Windows, Linux* y *Mac*, además de en sistemas de dispositivos móviles como puede ser *iOS y Android* [50], permitiendo así tener sus funcionalidades en teléfonos móviles o en ordenadores de placa reducida como pueda ser *Raspberry PI.*

Provee códigos y funciones optimizados para realizar las funciones más básicas de procesado de imágenes, como por ejemplo binarizados o detección de objetos. A su vez, también implementa una forma de trabajar con las imágenes gracias a su clase *cv::Mat* que permite almacenar los valores de los píxeles en la forma de una matriz, y permite trabajar con estos elementos de forma muy sencilla e intuitiva. Las distintas funcionalidades y clases de *OpenCV* que se utilizarán en este trabajo se explicarán a la hora de ver el código de las aplicaciones realizadas más adelante en el documento.

La adición de este conjunto de librerías que provee *Openc* supone varios cambios respecto al resto de paquetes que se han creado para este proyecto, puesto que hay que añadir al archivo *CMakeLists.txt*, dentro del apartado correspondiente a cada nodo que se quiere compilar y que utiliza funciones de la librería de *OpenCV* el siguiente trozo de código en su apartado *target_link_libraries: ${OpenCV_LIBRARIES}*, para que así el compilador tenga en cuenta esta librería a la hora de compilar los programas.

3.8.1 Programa de ajustes de la cámara

Si se utilizan estos programas por primera vez o si hay cambios en la posición de la cámara o los niveles de iluminación del entorno de la misma, lo recomendable sería lanzar previamente al uso del programa de seguimiento, la aplicación que corresponde a la configuración y calibración de la cámara. Esto se hace mediante el siguiente comando en la terminal:

```
rosrun camara_objeto configuracion_camara
```

De nuevo se supone que hay un *roscore* funcionando de forma previa. Al ejecutar dicha línea, se puede ver un menú que pregunta al usuario si se quiere configurar los niveles del filtro, o si bien

se quiere realizar la calibración. Si se elige la primera de las opciones aparecen 3 pantallas, dos de ellas conectadas a la cámara y una que contiene *sliders* como puede verse en la figura 3.27:

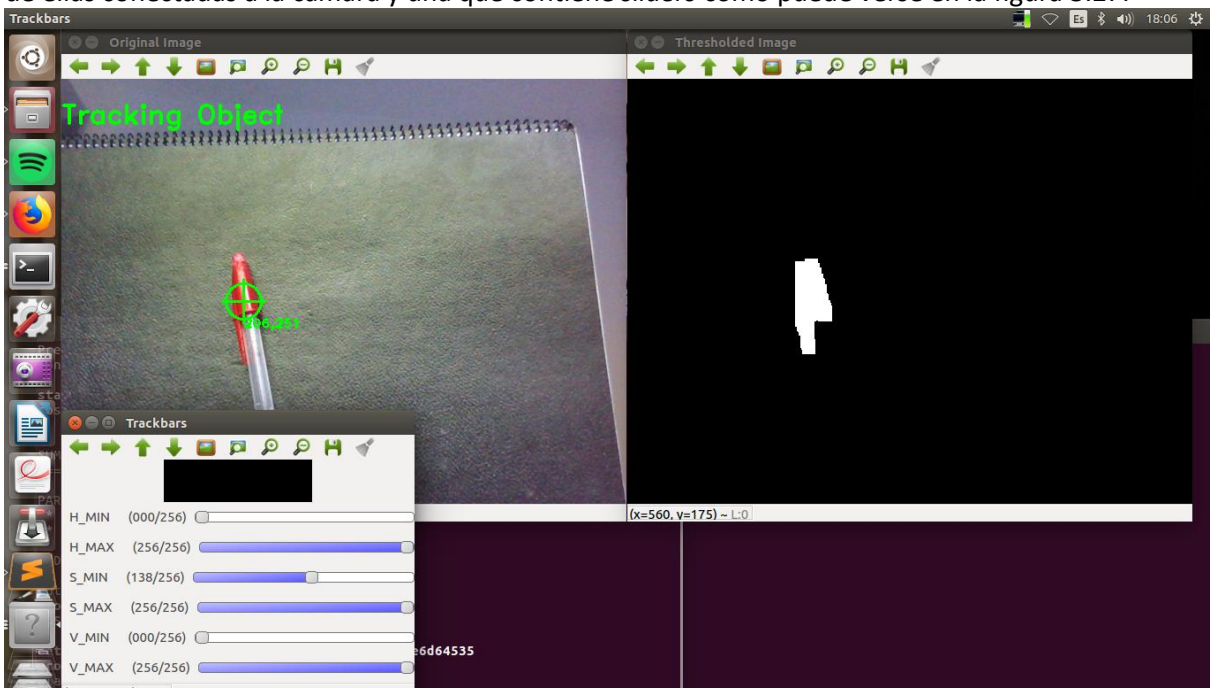

Figura 3.27: Pantalla de ajuste del filtro.

En la figura 3.27, en el lado superior izquierdo puede verse la imagen de la cámara directamente. La imagen de su lado corresponde a la misma imagen de la cámara, pero con un filtro *Threshold* que permite filtrar los objetos según su color ajustando los valores *H_MAX, H_MIN, S_MAX, S_MIN, V_MAX, V_MIN* mediante el menú de *slidebars* que aparece en la parte inferior derecha de la figura. La imagen de la derecha en un primer lugar aparece en blanco, pero si se ajustan los valores pueden aislarse colores de forma muy efectiva. En el caso de la ya mencionada figura, se ha filtrado un objeto rojo.

Una vez se ha ajustado el nivel de forma que se ha podido aislar el objeto deseado, si se cierra el programa, los valores de los filtros que quedaron al cierre de este son almacenados en un fichero de texto llamado *HSV.txt* que será cargado por el segundo programa que se explicará más adelante en el documento.

La segunda opción que ofrece este programa es cargar la imagen de un patrón asimétrico, concretamente el de la configuración que puede verse en la figura 3.28, para calcular la matriz de homografía.

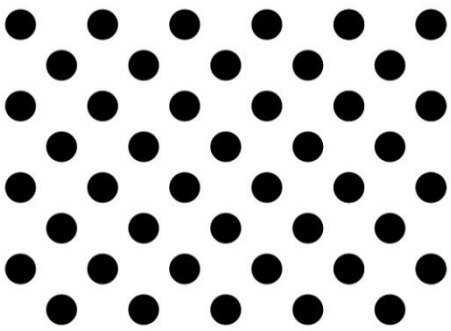

Figura 3.28: Patrón asimétrico utilizado en la aplicación.

Una vez se realiza el cálculo de dicha matriz (cómo se calcula se entrará más adelante cuando se explique el código de la aplicación), se almacenan los valores de forma similar a los parámetros del filtro, en un fichero de texto. Éste tiene el nombre de *H.txt.*

Cuando ya se tienen los valores apropiados para la aplicación almacenados en estos dos ficheros ya podría lanzarse el segundo de los programas que precisan de cámara para funcionar. Sin embargo, antes de entrar en éste se hará un comentario sobre el código del primero de ellos para explicar su funcionamiento completamente.

```
#include <sstream>
#include <string>
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
#include <cv.h>
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgcodecs/imgcodecs.hpp>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <fstream>
#include <cstdio>
#include "msgs_pruebas/Pose_pers.h"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/calib3d/calib3d.hpp"
```

Esto es muy similar a todas las aplicaciones explicadas anteriormente en este documento, pero con la diferencia de que para este caso hacen falta muchas más librerías que añaden las funcionalidades necesarias de *OpenCV*.

```
using namespace ros;
using namespace std;
using namespace cv;
```

Aquí se cargan las bibliotecas, tanto la de *ros,* como la de *OpenCV,* como la *std*. Esto permite que reducir el código y llamar las funciones sin necesidad de especificar qué biblioteca es la que las tiene, aunque en muchos de los casos como se verá en este código se añade. Esto es una especie de seguro que permite que, si en alguno de los casos se olvida, el compilador encuentre la función de todos modos.

```
int H_MIN = 0;
int H_MAX = 256;
int S_MIN = 0;
int S_MAX = 256;
int V_MIN = 0;
int V_MAX = 256;
```

Inicialización de los valores que se cambiarán posteriormente con los *sliders* se corresponden a los valores máximos y mínimos de tono, saturación y valor (*Hue, Saturation* y *Value)*, mediante el ajuste de estos máximos y mínimos es posible el filtrado de colores, y por tanto de objetos que se caractericen por ser de un color distinto del entorno.

```
const int FRAME_WIDTH = 640;
const int FRAME_HEIGHT = 480;

const int MAX_NUM_OBJECTS=50;

const int MIN_OBJECT_AREA = 20*20;
const int MAX_OBJECT_AREA = FRAME_HEIGHT*FRAME_WIDTH/1.5;

const string windowName = "Original Image";
const string windowName2 = "Thresholded Image";
const string trackbarWindowName = "Trackbars";
```

En este segmento se configuran algunos aspectos de las ventanas emergentes y del programa, como por ejemplo el área de trabajo de la cámara de 640x480 píxels, el número máximo de objetos que se pueden detectar como 50, el área mínima de los objetos a detectar (en píxels), el área máxima y el nombre de las ventanas emergentes.

```
cv::Mat segmentacion_auto(cv::Mat cort){
      cv::Mat tcort;
      cv::threshold(cort,tcort,0,255,cv::THRESH_OTSU);
      cv::imwrite ("imagen_seg_auto_2.jpg",tcort);
return  tcort;
}
```

Función que recibe una imagen almacenada en un *cv::Mat*, le aplica un filtro de umbral para binarizar la imagen que llega y la retorna.

```
void on_trackbar( int, void* )
{
}
```

Función a la que se llama cada vez que se realiza un movimiento de la *slidebar*.

```
string intToString(int number){
    std::stringstream ss;
    ss << number;
    return ss.str();
}
```

Función auxiliar que permite convertir los números en cadenas de caracteres, esto se utilizará más adelante en la función que permite dibujar el objeto para mostrar las coordenadas del centroide en la imagen de la cámara.

```
void createTrackbars(){
      namedWindow(trackbarWindowName,0);
    char TrackbarName[50];
    sprintf( TrackbarName, "H_MIN", H_MIN);
    sprintf( TrackbarName, "H_MAX", H_MAX);
    sprintf( TrackbarName, "S_MIN", S_MIN);
    sprintf( TrackbarName, "S_MAX", S_MAX);
    sprintf( TrackbarName, "V_MIN", V_MIN);
    sprintf( TrackbarName, "V_MAX", V_MAX);
      createTrackbar( "H_MIN", trackbarWindowName, &H_MIN, H_MAX, on_trackbar
);
```

```
        createTrackbar( "H_MAX", trackbarWindowName, &H_MAX, H_MAX, on_trackbar
);
        createTrackbar( "S_MIN", trackbarWindowName, &S_MIN, S_MAX, on_trackbar
);
        createTrackbar( "S_MAX", trackbarWindowName, &S_MAX, S_MAX, on_trackbar
);
        createTrackbar( "V_MIN", trackbarWindowName, &V_MIN, V_MAX, on_trackbar
);
        createTrackbar( "V_MAX", trackbarWindowName, &V_MAX, V_MAX, on_trackbar
);
}
```

Esta función es la que crea los distintos *slidebars* que pueden verse en la sección inferior izquierda de la figura 3.28. Los *createTrackbar* como puede adivinarse, son los objetos que permiten que los *sliders* existan, el primer parámetro es el nombre de la barra, el segundo el nombre de la ventana en la que se creará, el tercero indica la variable que se variará con el uso de esta barra entre cero y el cuarto parámetro que se indica en esta función, y por último se llama a la función vacía auxiliar de *callback* que se ha comentado previamente.

```
void drawObject(float x, float y,Mat &frame){

    circle(frame,Point(x,y),20,Scalar(0,255,0),2);
      if(y-25>0)
      line(frame,Point(x,y),Point(x,y-25),Scalar(0,255,0),2);
      else line(frame,Point(x,y),Point(x,0),Scalar(0,255,0),2);
      if(y+25<FRAME_HEIGHT)
      line(frame,Point(x,y),Point(x,y+25),Scalar(0,255,0),2);
      else line(frame,Point(x,y),Point(x,FRAME_HEIGHT),Scalar(0,255,0),2);
      if(x-25>0)
      line(frame,Point(x,y),Point(x-25,y),Scalar(0,255,0),2);
      else line(frame,Point(x,y),Point(0,y),Scalar(0,255,0),2);
      if(x+25<FRAME_WIDTH)
      line(frame,Point(x,y),Point(x+25,y),Scalar(0,255,0),2);
      else line(frame,Point(x,y),Point(FRAME_WIDTH,y),Scalar(0,255,0),2);
putText(frame,intToString(x)+","+intToString(y),Point(x,y+30),1,1,Scalar(0,25
5,0),2);
}
```

Esta es la función que, una vez se ha detectado el objeto, dibuja la cruz que indica en la imagen de la izquierda dónde se encuentra el centoide del objeto dibujado, así como sus coordenadas.

```
void morphOps(Mat &thresh){
    Mat erodeElement = getStructuringElement( MORPH_RECT,Size(3,3));
    Mat dilateElement = getStructuringElement( MORPH_RECT,Size(8,8));
    erode(thresh,thresh,erodeElement);
    erode(thresh,thresh,erodeElement);
    dilate(thresh,thresh,dilateElement);
    dilate(thresh,thresh,dilateElement);
}
```

Esta sección del código se trata de una función auxiliar que erosiona y dilata los elementos para eliminar el posible ruido que pueda existir en la detección de un objeto, puesto que aunque un objeto sea de un color, su situación espacial o la iluminación pueden causar que se distingan como dos objetos distintos (a efectos del programa) o que parte del objeto se pierda. Puesto que la forma que tiene el objeto no es demasiado relevante, porque lo que más importa es dónde se encuentre el centroide del mismo, estas operaciones pueden realizarse con cierta

agresividad. Sin embargo, en la figura 3.28 puede observarse que la forma del objeto se respeta en la medida de lo posible.

```
void trackFilteredObject(float &x, float &y, Mat threshold, Mat &cameraFeed){

    Mat temp;
    threshold.copyTo(temp);
    vector< vector<Point> > contours;
    vector<Vec4i> hierarchy;

    findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE
);

    double refArea = 0;
    bool objectFound = false;
    if (hierarchy.size() > 0) {
        int numObjects = hierarchy.size();

       if(numObjects<MAX_NUM_OBJECTS){
              for (int index = 0; index >= 0; index = hierarchy[index][0]) {

                    Moments moment = moments((cv::Mat)contours[index]);
                    double area = moment.m00;

             if(area>MIN_OBJECT_AREA && area<MAX_OBJECT_AREA && area>refArea){
                          x = moment.m10/area; //Coordenada del centroide en
X
                          y = moment.m01/area; //Coordenada del centroide en
Y
                          objectFound = true;
                          refArea = area;
                    }else objectFound = false;
              }
              if(objectFound ==true){
        putText(cameraFeed,"Tracking
Object",Point(0,50),2,1,Scalar(0,255,0),2);
        drawObject(x,y,cameraFeed);
              }
        }else      putText(cameraFeed,"TOO      MUCH      NOISE!      ADJUST
FILTER",Point(0,50),1,2,Scalar(0,0,255),2);
    }
}
```

Esta es la función que hace posible el *tracking* como tal del objeto, en primer lugar recibe los centroides del objeto en forma de puntero, la imagen tras la aplicación del filtro y la imagen correspondiente a la cámara. Con esta información se aplica la función *findContours* que realiza un análisis de la fotografía que se le da y detecta el número de objetos que encuentra en ella (una vez se ha binarizado esta previamente). Este número de objetos se encuentra en el tamaño del vector *hierarchy*. Si el valor de este objeto es superior a 0 y menor al número máximo que se eligió al principio del código, entonces se procede al cálculo de distintas propiedades mediante la función *moments*. Esta permite calcular todos los momentos espaciales de los objetos detectados, los más significativos son m00 (área), m10 y m01, puesto que con ellos se puede calcular el centroide de los objetos (que se almacena en los punteros que se han enviado a la función al principio). A su vez, se indica que se ha encontrado un objeto dándole el valor *true* a *objectFound*, y permitiendo la publicación del dibujo de la cruz que sigue el objeto que se quiere seguir.

```
void configfile(int H_MAX, int S_MAX,int V_MAX,int H_MIN,int S_MIN,int V_MIN){
    std::ofstream myfile;
    myfile.open("/home/claudia/HSV.txt",         std::ofstream::out         |
std::ofstream::trunc);
    myfile << H_MAX << " " << S_MAX<< " " << V_MAX << " " << H_MIN << " " <<
S_MIN << " " << V_MIN << "\n";
    myfile.close();

}
```

Esta parte del código simplemente guarda los valores que se han seleccionado para el filtrado del objeto en el fichero de configuración de texto *HSV.txt.*

```
cv::Mat homografia_auto(){

      cv::Mat imagen;
      cv::Mat image;
      cv::Mat gris;
      imagen=cv::imread("/home/claudia/Patron_1.jpg", CV_LOAD_IMAGE_COLOR);
      gris=cv::imread("/home/claudia/Patron_1.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      printf("Imagen de patron leída\n");
      image=segmentacion_auto(gris);

      //printf("dgfldfihgwl\n");
      //cv::imwrite("Patron.jpg",image);
      vector<vector<cv::Point> >  conto;
      vector<cv::Vec4i> hc;
      cv::findContours(image, conto, hc, CV_RETR_LIST, CV_CHAIN_APPROX_NONE);
      int numContours2 = conto.size()-1;

      vector<cv::Moments> m_c(numContours2);
      vector<cv::Point2f> centroide_c(numContours2);

      for(int i=0; i<numContours2;i++){
      m_c[i]=(cv::moments(conto [i],true));

      centroide_c[i]=(cv::Point2f(static_cast<float>(m_c[i].m10/m_c[i].m00),
static_cast<float>(m_c[i].m01/m_c[i].m00)));
      }

      cv::Size imSize;
      cv::Size bSize; bSize.height = 4; bSize.width = 11;
      vector<cv::Point2f> dPoints;
      cv::SimpleBlobDetector::Params params;
      params.minThreshold = 0; params.maxThreshold = 150;
      params.filterByArea = true; params.minArea = 100; params.maxArea = 600;
      params.filterByCircularity = false; params.filterByConvexity = false;
      params.filterByInertia = false;
      cv::Ptr<cv::SimpleBlobDetector>                                    bD
=cv::SimpleBlobDetector::create(params);

      cv::Scalar rosa = cv::Scalar(255, 0, 255);

      cv::drawChessboardCorners(imagen, bSize, cv::Mat(centroide_c), true);
      cv::circle(imagen, centroide_c[43], 5, rosa, 2, 8, 0); //AQUÍ SE CAMBIA
EL CENTRO.
      cv::imwrite("Punto.jpg",imagen);
```

```
	cv::Mat Homografia(numContours2*2,9, cv::DataType<float>::type);

	int X_auto[numContours2], Y_auto[numContours2];
	float x_auto[numContours2],y_auto[numContours2];

	for(int i=0; i<numContours2; i++){
	x_auto[i]=centroide_c[i].x;
	y_auto[i]=centroide_c[i].y;
	}
	int fila=0;
	int d=55;
	int i=0;
	for (fila=0; fila<8; fila++){
	if(fila==0||fila==2||fila==4||fila==6){
		for(int j=0; j<5; j++){
		X_auto[i]=(4-j)*55.0+55.0/2.0;
		Y_auto[i]=(3-fila/2)*55.0+55.0/2.0;
		i++;
		}
	}
	if(fila==1||fila==3||fila==5||fila==7){
		for(int j=0; j<6; j++){
		X_auto[i]=(5-j)*55;
		Y_auto[i]=(3-fila/2)*55;
		i++;
		}
	}
	}

	for (int i=0; i<numContours2*2;i=i+2){
		int j=i+1;
		Homografia.at<float>(i, 0)=X_auto[i/2];
		Homografia.at<float>(i, 1)=Y_auto[i/2];
		Homografia.at<float>(i, 2)=1;
		Homografia.at<float>(i, 3)=0;
		Homografia.at<float>(i, 4)=0;
		Homografia.at<float>(i, 5)=0;
		Homografia.at<float>(i, 6)=-x_auto[i/2]*X_auto[i/2];
		Homografia.at<float>(i, 7)=-x_auto[i/2]*Y_auto[i/2];
		Homografia.at<float>(i, 8)=-x_auto[i/2];
		Homografia.at<float>(j, 0)=0;
		Homografia.at<float>(j, 1)=0;
		Homografia.at<float>(j, 2)=0;
		Homografia.at<float>(j, 3)=X_auto[i/2];
		Homografia.at<float>(j, 4)=Y_auto[i/2];
		Homografia.at<float>(j, 5)=1;
		Homografia.at<float>(j, 6)=-y_auto[i/2]*X_auto[i/2];
		Homografia.at<float>(j, 7)=-y_auto[i/2]*Y_auto[i/2];
		Homografia.at<float>(j, 8)=-y_auto[i/2];
	}

	cv::Mat u_auto, v_auto,s_auto, vt_auto;
	cv::SVD::compute(Homografia,	s_auto,	u_auto,	vt_auto,
cv::SVD::FULL_UV);
	v_auto=vt_auto.t();
	cv::Mat H_auto(3,3, cv::DataType<float>::type);
	float v_9_auto[9];
	for(int i=0; i<9; i++){
```

```
		v_9_auto[i]=v_auto.at<float>(i,8);

		}
		int cont_auto=0;
		for(int i=0; i<3; i++){
			for(int j=0; j<3; j++){
			H_auto.at<float>(i,j)=v_9_auto[cont_auto];
			cont_auto++;
			}
		}

		return H_auto;
}
```

Aquí se carga la imagen del entorno con el patrón en posición. Se conoce el valor de la distancia entre los centros de los círculos que conforman el patrón, y con este valor y la distancia entre los objetos que se detecta gracias al programa, se puede realizar el cálculo de la matriz de homografía. En este caso se ha calculado utilizando la última columna de la matriz V de la descomposición en valores singulares, tal como se ha explicado en el apartado 2.5.5, sólo que en este caso, en lugar de utilizar 4 puntos se han utilizado todos los del patrón.

```
void configfile2(cv::Mat H){
		std::ofstream myfile;
		myfile.open("/home/claudia/H.txt", std::ofstream::out |
std::ofstream::trunc);
		myfile << H.at<float>(0,0) << " " << H.at<float>(0,1)<< " " <<
H.at<float>(0,2) << " "<< H.at<float>(1,0) << " " << H.at<float>(1,1)<< " "
<< H.at<float>(1,2) << " " << H.at<float>(2,0) << " " << H.at<float>(2,1)<< "
" << H.at<float>(2,2) << "\n";
		myfile.close();
}
```

De forma similar a la función *configfile*, este segmento de código se utiliza para guardar los valores que conforman la matriz de homografía que se ha obtenido en la función anterior.

```
int main(int argc, char* argv[])
{
    ros::init(argc, argv, "configuracion_camara");
    ros::NodeHandle n;
    ros::AsyncSpinner spinner(1);
      spinner.start();
    Mat cameraFeed;
    Mat HSV;
    Mat threshold;
    float x=0, y=0;

    VideoCapture capture;
    capture.open(0);
    int teclado=0;
    capture.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
    capture.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);

    printf("Menú:\n");
    printf("1. Determinar valores del filtro\n");
    printf("2. Calibración cámara\n");
    scanf("%d", &teclado);
```

```
    switch(teclado){
        case 1: {
    while(ros::ok()){
        createTrackbars();

        capture.read(cameraFeed);

        cvtColor(cameraFeed,HSV,COLOR_BGR2HSV);


inRange(HSV,Scalar(H_MIN,S_MIN,V_MIN),Scalar(H_MAX,S_MAX,V_MAX),threshold);
        configfile(H_MAX,S_MAX,V_MAX,H_MIN,S_MIN,V_MIN);
        //if(useMorphOps)
        morphOps(threshold);
        //if(trackObjects)
        trackFilteredObject(x,y,threshold,cameraFeed);
        imshow(windowName2,threshold);
        imshow(windowName,cameraFeed);

        waitKey(30);
    }
    break; }
        case 2:{
               cv::Mat H;
               H=homografia_auto();
               configfile2(H);
               break; }
}
    ros::shutdown();
    return 0;
}
```

Por último, en este programa se tiene el *main* en el que, como puede verse y como en el resto de los programas de este documento, se realiza la inicialización de *ROS*, además en este caso se llama a la cámara mediante las funciones propias para ello que facilita *OpenCV*. También es aquí donde se alberga el código que genera parte el menú que le aparece al usuario cuando inicializa el programa. Cuando selecciona cualquiera de las dos opciones, en un *switch* se llaman a las correspondientes funciones que llevan a cabo todo lo que se ha descrito a lo largo del código.

3.8.2 Programa de seguimiento y envío de información.

El siguiente programa a utilizar es aquel que, con la configuración que se ha realizado previamente mediante la ejecución de la aplicación anterior, permite el seguimiento como tal del objeto y el envío de las coordenadas del objeto en unas unidades que permita representarlo en *Rviz*, es decir, realiza la transformación de coordenadas y cuando se realiza este cálculo se envían a través de un mensaje de *ROS* a un tercer programa que es el que representa el objeto en función del valor del centroide que se detecta aquí.

Este programa no tiene ningún menú, simplemente se ejecuta con el siguiente comando:
rosrun camara_objeto seguimiento_objeto
Y éste carga los valores de los dos ficheros de configuración, y si se detecta el objeto en cuestión, realiza los cálculos pertinentes utilizando la matriz de homografía y envía la información.

Su código es muy similar al anterior, es por tanto que sólo se comentarán las secciones que representen un cambio significativo, como por ejemplo que las funciones que llaman y generan las *slidebars* no existen. Otra variación es la no existencia de las funciones que generan la matriz de homografía ni las que generan los ficheros de configuración, por otra parte sí se tienen dos funciones que leen el contenido de los ficheros y lo almacenan:

```
cv::Mat HSVsetup(){
    cv::Mat HSV_Values(6 ,1, cv::DataType<float>::type);;
    std::ifstream infile("/home/claudia/HSV.txt");
      std::string line;
      while (std::getline(infile, line))
  {
      std::istringstream iss(line);
      iss   >>   HSV_Values.at<int>(0,0)   >>   HSV_Values.at<int>(1,0)
>>HSV_Values.at<int>(2,0)      >>      HSV_Values.at<int>(3,0)      >>
HSV_Values.at<int>(4,0) >> HSV_Values.at<int>(5,0);
  }
    infile.close();
    return HSV_Values;
}
```

Como se puede ver en el caso de esta primera función, lee los valores del filtro almacenados en el fichero *HSV.txt* y los almacena en un vector creado como matriz de *OpenCV* de 6 filas y una columna, para facilitar su retorno a la parte del programa que invoque este servicio.

```
cv::Mat Hsetup(){
    cv::Mat H_Values(3 ,3, cv::DataType<float>::type);;
    std::ifstream infile("/home/claudia/H.txt");
      std::string line;
      double h1, h2, h3, h4, h5, h6, h7, h8, h9;

      while (std::getline(infile, line))
  {
      std::istringstream iss(line);
      iss >> h1 >> h2 >> h3 >> h4 >> h5 >> h6 >> h7 >> h8 >> h9;
}
    H_Values.at<float>(0,0)=h1;
    H_Values.at<float>(0,1)=h2;
    H_Values.at<float>(0,2)=h3;
    H_Values.at<float>(1,0)=h4;
    H_Values.at<float>(1,1)=h5;
    H_Values.at<float>(1,2)=h6;
    H_Values.at<float>(2,0)=h7;
    H_Values.at<float>(2,1)=h8;
    H_Values.at<float>(2,2)=h9;
    infile.close();
    return H_Values;
}
```

Por otra parte, esta segunda función es muy similar, pero la diferencia es que se utiliza una matriz de *OpenCV* en este caso de 3x3, para de nuevo facilitar el retorno, y de forma adicional, simplificar la forma de realizar las operaciones, porque parte de las facilidades que ofrece

OpenCV es que las operaciones matriciales se realizan sin necesidad de aplicar operadores complicados, como podrá verse más adelante en el documento.

```
int main(int argc, char* argv[])
{
    ros::init(argc, argv, "seguir_objeto");
    ros::NodeHandle n;
    ros::AsyncSpinner spinner(1);
      spinner.start();


      cv::Mat HSV_setup;
      HSV_setup=HSVsetup();
      H_MIN = HSV_setup.at<int>(3,0);
    H_MAX = HSV_setup.at<int>(0,0);
    S_MIN = HSV_setup.at<int>(4,0);
    S_MAX = HSV_setup.at<int>(1,0);
    V_MIN = HSV_setup.at<int>(5,0);
    V_MAX = HSV_setup.at<int>(2,0);
```

Aquí se realiza la llamada a la función de lectura del fichero de configuración del filtro y se almacena cada valor del vector en su correspondiente variable, para facilitar la aplicación del filtro un poco más adelante en el código.

```
    cv::Mat H;
    H=Hsetup();

    cv::Mat Hinv;
    Hinv=H.inv();
```

De forma similiar, se llama a la función del fichero en el que se encuentran los valores de la matriz de homografía, pero en este caso se crea otra matriz para almacenar el valor de su inversa. De nuevo, gracias a *OpenCV* el cálculo de esta matriz se simplifica considerablemente gracias al operador *.inv()*.

```
    Mat cameraFeed;

    Mat HSV;

    Mat threshold;
    float x=0, y=0;

    VideoCapture capture;
    capture.open(0);

    capture.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
    capture.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);
```

Configuración de la cámara, muy similar al programa anterior.

```
ros::Publisher
chatter_pub=n.advertise<msgs_pruebas::Pose_pers>("chatter", 1000);
msgs_pruebas::Pose_pers msg;
```

Este segmento supone uno de los grandes cambios respecto al programa anterior. Gracias a estas líneas de código se pueden publicar los mensajes que recibirá posteriormente el tercer programa que se explicará a continuación y que contienen el valor del centroide del objeto que se ha detectado. Básicamente lo que se hace es, crear el objeto de tipo *Publisher* con el nombre *chatter_pub* y en él se almacena el objeto *node_handler* creado en la inicialización del programa tras aplicarle el operador *advertise*, aquí se indica qué tipo de mensajes se utilizarán, y posteriormente, el nombre del *topic* en el que se publicarán y el tamaño máximo de la cola de mensajes que creará. Por último se crea un objeto de tipo mensaje que será lo que se enviará a través del *topic* que acaba de crearse gracias a la operación anterior.

```
    while(ros::ok()){

       capture.read(cameraFeed);

       cvtColor(cameraFeed,HSV,COLOR_BGR2HSV);


inRange(HSV,Scalar(H_MIN,S_MIN,V_MIN),Scalar(H_MAX,S_MAX,V_MAX),thresh
old);

       morphOps(threshold);
       trackFilteredObject(x,y,threshold,cameraFeed);

      cv::Mat posicion_obj(1 ,3, cv::DataType<float>::type);
      cv::Mat aux(3 ,1, cv::DataType<float>::type);
      cv::Mat cent(3 ,1, cv::DataType<float>::type);
            cent.at<float>(0,0)=x;
            cent.at<float>(1,0)=y;
            cent.at<float>(2,0)=1;
            aux=Hinv*cent;
            aux=aux/aux.at<float>(2,0);
            posicion_obj.at<float>(0,0)=aux.at<float>(0,0);
            posicion_obj.at<float>(0,1)=aux.at<float>(1,0);
            posicion_obj.at<float>(0,2)=1;
       msg.x = x/1000;//posicion_obj.at<float>(0,0);
       msg.y= y/1000;//posicion_obj.at<float>(0,1);
       chatter_pub.publish(msg);

       imshow(windowName2,threshold);
       imshow(windowName,cameraFeed);
       waitKey(30);
    }

    ros::shutdown();
    return 0;
}
```

Por último, se tiene dentro de un bucle que se encuentra funcionando hasta que se detiene manualmente el programa la localización y seguimiento del objeto, la conversión de las coordenadas del centroide de píxels a centímetros y por último, se puede ver que el resultado de la conversión se almacena en los objetos del mensaje que se ha creado anteriormente, y se envía utilizando la línea `chatter_pub.publish(msg)`. Esto permite que se pueda estar enviando la información continuamente al encontrarse dentro del bucle.

3.9. PROGRAMA DEL OBJETO EN MOVIMIENTO EN EL ENTORNO DEL ROBOT

Para finalizar la definición del entorno se ha creado un último nodo de *ROS* que, como se ha comentado brevemente anteriormente, recibe la información del centroide del objeto que se ha detectado y crea un objeto que se mueve siguiendo este centroide. Para recibir la información, se hace mediante el sistema, ya explicado, de mensajes de *ROS* vía *topic.*

El objeto que se crea en este caso tiene la forma de un prisma de dimensiones 0.1 cm x 0.11 cm x 0.25 cm y se encuentra elevado 0.155 cm respecto al plano que representa el suelo del entorno. Esto es debido a que se asume que el objeto se encontrará sobre la cinta transportadora, sin embargo si de detectara fuera de la misma, también se dibujaría. El objeto junto con el entorno pueden verse en la figura 3.29:

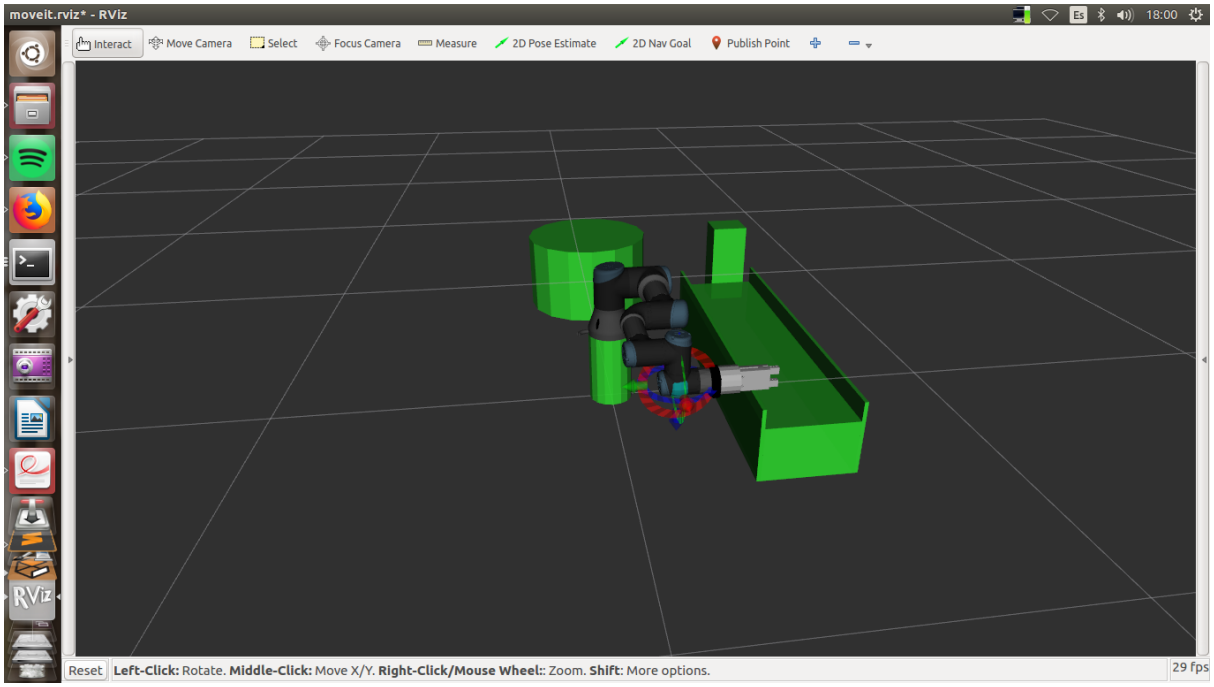

Figura 3.29: Entorno de trabajo del robot con objeto incorporado.

El objeto se encontrará en movimiento siempre y cuando se registre un objeto en programa de seguimiento, y en caso de desaparecer del alcance de la cámara se almacena el último valor guardado y el objeto dibujado se quedará en esa posición. La actualización entre movimientos se realiza cada número de segundos que indique el usuario por pantalla al inicio del programa.

Si se analiza el código de este programa se tiene lo siguiente:

```
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>
#include <moveit_visual_tools/moveit_visual_tools.h>
#include "msgs_pruebas/Pose_pers.h"
#include "msgs_pruebas/Pose_pers.h"
#include <iostream>
#include <ctime>
```

```
#include <cstdlib>
#include <fstream>
#include <sstream>
#include <cstdio>
#include <sstream>
#include <string>
#include <iostream>

using namespace ros;
using namespace std;
```

Similar a lo realizado en otros programas, añadir los distintos archivos *header* necesarios para llevar a cabo las funcionalidades buscadas, y las librerías donde se encuentran las distintas funciones.

```
class Listener{

  public:
      double x;
      double y;

      void   chatterCallback(const   msgs_pruebas::Pose_pers::ConstPtr&
msg);
};
```

Aquí se ha creado una clase especial para poder utilizar los valores que se reciban desde el *topic* mediante el cual se reciben los valores del centroide del objeto detectado. Sólo se necesitan dos variables que corresponden a las coordenadas de dicho centroide. De forma adicional en la clase se declara la función de llamada al mensaje para asegurar su correcto funcionamiento.

```
void                                    Listener::chatterCallback(const
msgs_pruebas::Pose_pers::ConstPtr& msg)
{

   x=msg->x;
   y=msg->y;
}
```

Esta es la función con la que se lleva a cabo el recibo de la información del *topic* y gracias a la cual se pueden utilizar posteriormente esos datos para la creación del objeto.

```
int main(int argc, char** argv)
{

  ros::init(argc, argv, "objeto_movimiento");
  ros::NodeHandle node_handle;
  ros::AsyncSpinner spinner(10);
  spinner.start();
  Listener oyente;
```

Similar a todos los programas, inicialización de *ROS* y en este caso, declaración de una variable de tipo *Listener*, la clase que se ha definido de forma previa.

```
  ROS_INFO("Indique tiempo de espera entre movimientos del objeto");
      float espera;
      scanf("%f",&espera);
```

En este segmento de código se imprime por pantalla el mensaje para que el usuario indique el tiempo en segundos que habrá entre dos actualizaciones de la posición del objeto cuando este se encuentre en movimiento.

```
ros::Subscriber sub=node_handle.subscribe("chatter", 1000,
&Listener::chatterCallback, &oyente);
  ros::Rate loop_rate(30);
```

Aquí se realiza la creación del objeto de tipo suscriptor. Una vez se ha creado este objeto, el nodo que lo contiene recibe la información del *topic* al que se ha suscrito de forma continua. Se puede ver a la hora de crear este objeto, que de forma muy similar al objeto de tipo *Publisher* comentado en el apartado 3.10.2 de este documento, hay que aplicar el operador *suscribe* al objeto de tipo *node handle* y dentro de este se indican tanto el *topic,* como el número de objetos en cola máximo como la función que lleva a cabo la *escucha* como tal, y por último el objeto de la clase *Listener* que se ha creado al inicio del *main* y que es donde se almacenarán los valores recibidos.

```
  int control=0;

  ROS_INFO("Esperando posicion del objeto");

  while(control==0){
  ros::spinOnce();
  if(sub.getNumPublishers()){
      if(oyente.x!=0)
      control=1;
  }
   loop_rate.sleep();
  }
```

El siguiente bucle se trata de un punto de control que lo que hace es esperar a que exista un nodo publicando información sobre la situación del objeto. Hasta que el presente nodo no detecta que existe un *publisher* en el *topic* y hasta que la información que recibe no es distinta de 0 para x, las operaciones del nodo no continuan. Esto se utiliza para asegurar que el objeto existe y puede seguirse.

```
  static const std::string PLANNING_GROUP = "manipulator";
  moveit::planning_interface::MoveGroupInterface
move_group(PLANNING_GROUP);
  moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;
  const robot_state::JointModelGroup* joint_model_group =
      move_group.getCurrentState()-
>getJointModelGroup(PLANNING_GROUP);

  namespace rvt = rviz_visual_tools;
  moveit_visual_tools::MoveItVisualTools visual_tools("world");
```

```
  visual_tools.deleteAllMarkers();
  visual_tools.loadRemoteControl();

  Eigen::Affine3d text_pose = Eigen::Affine3d::Identity();
  text_pose.translation().z() = 1.75;//esto publica mensajes en la
pantallita del rviz
  visual_tools.trigger();


  //AÑADIMOS EL OBJETO
  moveit_msgs::CollisionObject collision_object;
  collision_object.header.frame_id = move_group.getPlanningFrame();
  collision_object.id = "box1";
  geometry_msgs::Pose box_pose;
  shape_msgs::SolidPrimitive primitive;
  std::vector<moveit_msgs::CollisionObject> collision_objects;
  std::vector<moveit_msgs::CollisionObject> objetos_fijos;
```

Esto, como puede observarse es clónico al código utilizado en la definición del entorno fijo del robot, y es normal puesto que se trata de las mismas funcionalidades de código.

```
  float altura=0.25;

// Tamaño y forma del objeto
  primitive.type = primitive.BOX;
  primitive.dimensions.resize(3);
  primitive.dimensions[0] = 0.1;
  primitive.dimensions[1] = 0.11;
  primitive.dimensions[2] = altura;

  // Posición del objeto
  box_pose.orientation.w = 1.0;
  box_pose.position.x = oyente.y;
  box_pose.position.y = oyente.x;
  box_pose.position.z = 0.155+altura/2;

  collision_object.primitives.push_back(primitive);
  collision_object.primitive_poses.push_back(box_pose);
  collision_object.operation = collision_object.ADD;
  collision_objects.push_back(collision_object);
```

Esta es la definición de características de tamaño y posición del objeto en movimiento. De nuevo, se trata de una operación muy similar a la llevada a cabo en la definición del espacio de trabajo, pero esta vez puede verse que la posición se encuentra en función de los datos recibidos vía *topic* de ahí que se utilice el objeto *oyente*. Las coordenadas debido a que los ejes de coordenadas de *OpenCV* y de *ROS* se encuentran rotados 180º, y puesto que el eje z no influye en estas operaciones, con sustituir una con la otra es suficiente para que el seguimiento del objeto se corresponda con el movimiento real.

```
  planning_scene_interface.addCollisionObjects(collision_objects);
  ros::Duration(espera).sleep();
```

Esta es la publicación del objeto en el entorno de *MoveIt!*

```
while(ros::ok()){
  moveit_msgs::CollisionObject move_object;
  move_object.id="box1";
  move_object.operation = move_object.MOVE;
  move_object.header.frame_id = move_group.getPlanningFrame();
  box_pose.position.x = oyente.y;
  box_pose.position.y=oyente.x;
  move_object.primitive_poses.clear();
  move_object.primitive_poses.push_back(box_pose);
  collision_objects.clear();
  collision_objects.push_back(move_object);
  planning_scene_interface.addCollisionObjects(collision_objects);
  ros::Duration(espera).sleep();
}
  ros::shutdown();
  return 0;
}
```

Por último, dentro del bucle se encuentra la mayor peculiaridad que puede encontrarse en este programa, puesto que es la sección de código que permite el movimiento del objeto. Para que esto sea posible no es suficiente con cambiar las coordenadas previamente definidas del objeto, sino que hay que crear una variable de tipo *CollisionObject (move_object)* que se asociará al objeto anteriormente creado gracias a darles la misma identificación mediante el operador *id*. Tras asociar ambos objetos, se indica que se realizará el movimiento con el operador *.MOVE*, posteriormente se indica el cambio de coordenadas en el objeto de tipo *pose* que se había creado anteriormente, y se asocia a *move_object*. Cuando este objeto se publica como se ha hecho anteriormente, es cuando es visible el movimiento. Este bucle se ejecuta hasta que el programa se detiene, y cada iteración se lleva a cabo después de la espera de los segundos que se hayan indicado por pantalla, así las actualizaciones se llevan a cabo después del tiempo estipulado.

3.10 PROGRAMAS DE MOVIMIENTO

Los programas que se han creado para definir el entorno y los posibles objetos que pueden existir dentro del mismo (incluso aquellos que se mueven) son suficiente para que cualquier operario utilice el robot de forma segura. Sin embargo, la programación en el entorno de *ROS* puede resultar dificultosa y poco práctica para estos casos, es por tanto que se añaden dos programas complementarios que facilitan la programación manual. El primero de ellos se utiliza para definir las posiciones a las que se quiere llevar el robot, y el segundo para que éste las ejecute.

3.10.1 Programa de definición de posiciones.

Este programa, como se ha comentado brevemente, es el que se utilizaría para definir las posiciones a las que se quiere llevar el robot en el siguiente programa. Se encuentra en el paquete *movimiento_ur3* y que como es el caso del anterior, ha tomado las dependencias del paquete *moveit_tutorials* para que ninguna de las herramientas que se utilizan en él falte, y así

asegurarse el correcto funcionamiento del mismo. Este programa se lanzaría escribiendo en la terminal el siguiente comando:

rosrun movimiento_ur3 generar_fichero

Se ha optado en este caso por un *rosrun* puesto que se asume que el ordenador se encuentra conectado al robot, por lo que ya se ha lanzado previamente un *launch* que ha inicializado el *master* (ya que para que se pueda ejecutar un programa mediante *rosrun* éste debe estar operativo).

Cuando se ejecuta esto, puede verse en la terminal el siguiente menú (figura 3.30) si el robot o el simulador está conectado, si no está conectado podrá verse en la pantalla un mensaje de error:

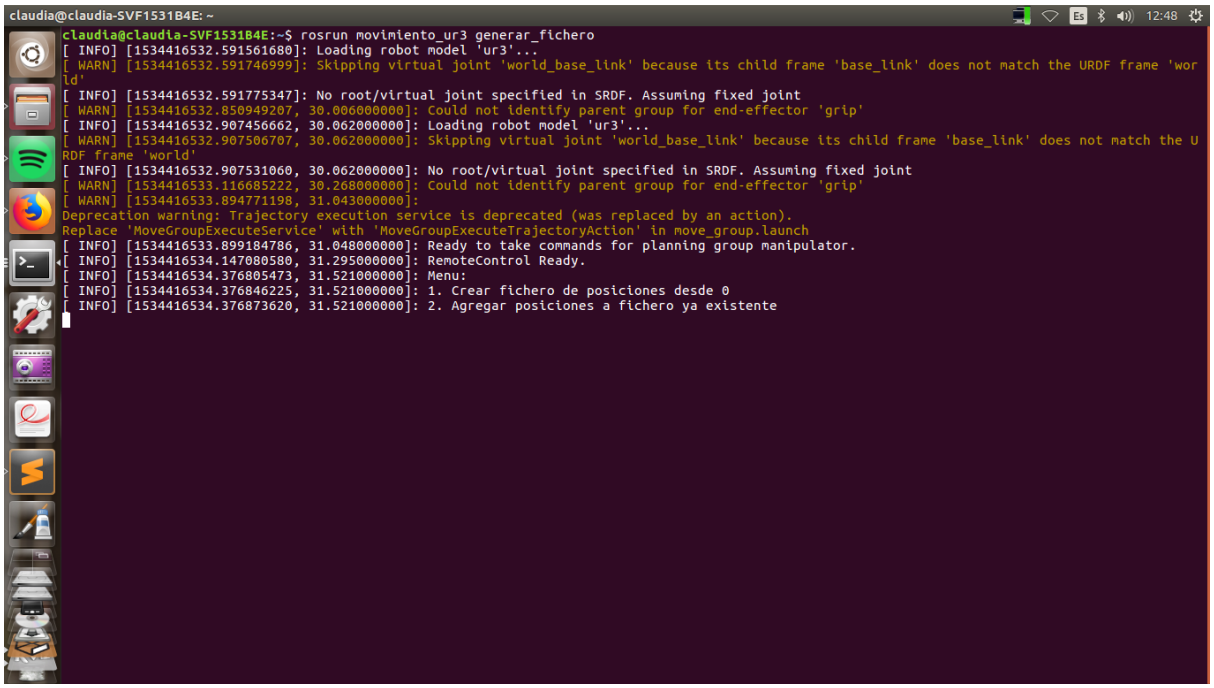

Figura 3.30: Menú de inicio del programa *generar_fichero.*

Como puede verse, al inicializar existen dos opciones, o bien crear la secuencia de movimientos desde cero, o bien tomar la secuencia de movimientos que se ha utilizado anteriormente y añadir nuevas posiciones.

Esto es posible gracias a que las posiciones se almacenan en un fichero llamado *joints.txt* que se encuentra localizado en la carpeta personal. Al seleccionar la primera opción, este fichero se borra y se crea de nuevo de cero para almacenar los nuevos puntos. Si es la segunda opción la que se marca, simplemente, el archivo existente se abre y se guarda nueva información a continuación de la ya existente.

Una vez se ha seleccionado cualquiera de las dos opciones, lo que puede verse en la terminal en la que se está ejecutando el programa es lo que muestra la figura 3.31:

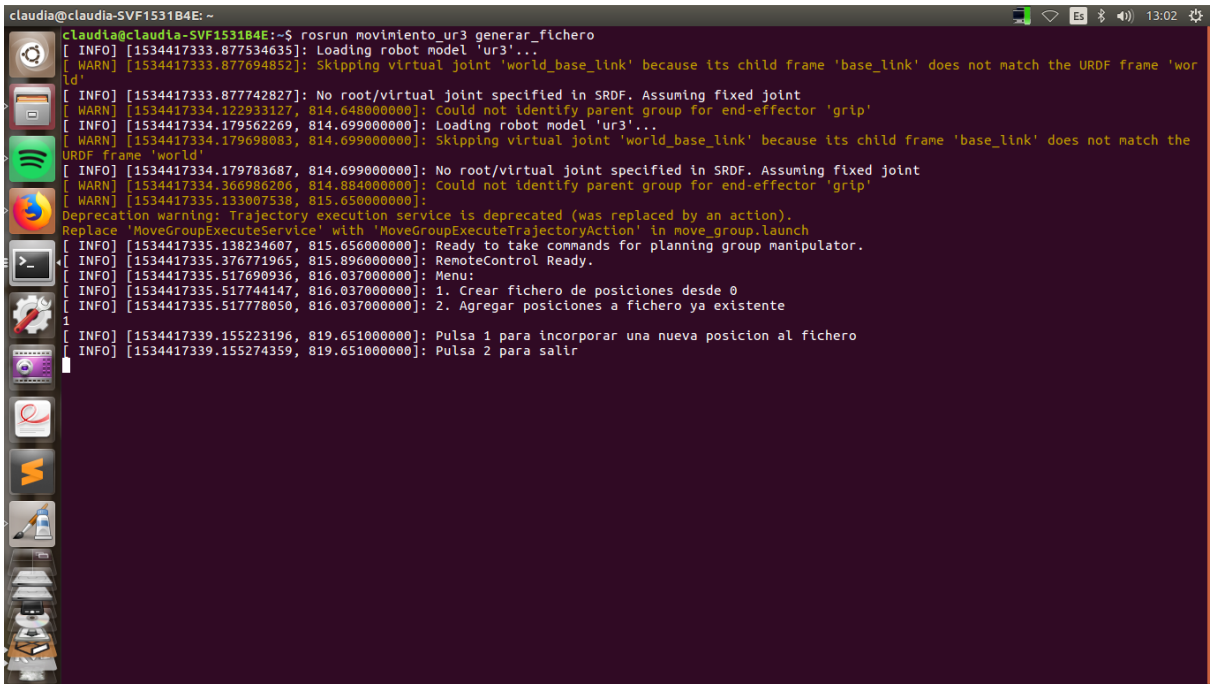

Figura 3.31: Segundo menú programa *generar_fichero.*

Aquí lo único que hay que hacer es llevar el robot a la posición deseada, o bien de forma manual con el botón que facilita el *UR3* para liberar los frenos y poder moverlo, o bien con los diferentes métodos que ofrecen *ROS y MoveIt!* (la *GUI* de *Rviz* o el programa que se puede encontrar en el paquete *moveit_commander*). Una vez se encuentra el robot en la posición deseada, se pulsa 1 y el programa almacena en el fichero .txt en cada uno del valor de las articulaciones. Cómo se guardaría viene en la figura 3.32 adjuntada a continuación:

```
1  0.587202 -1.18068 1.64134 -1.9905 -1.52375 0.664447
2  0.999087 -0.581351 0.812125 -2.11211 -1.4202 1.09026
3  1.64877 -0.477927 0.584233 -1.97016 -1.52013 1.62056
4  1.64982 -1.34041 1.85452 -2.06991 -1.52561 1.61711
```

Figura 3.32: Ejemplo de formato del fichero *joints.txt*

En este caso, como se tienen 6 articulaciones, se puede ver que en cada línea hay 6 valores, y se tendrían en total cuatro posiciones ya que sólo hay 4 líneas en este fichero. Una vez el robot se ha llevado a todas las posiciones que se buscan y se han almacenado en el fichero, se pulsa la tecla 2 para finalizar el programa.

Ahora se entrará en el código *C++* de este programa para analizar cómo funciona internamente el programa:

```
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>

#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

#include <moveit_visual_tools/moveit_visual_tools.h>
#include <fstream>
```

```
#include <string>
```

Esto, de forma similar al programa del entorno, define los *headers* necesarios para poder llevar a cabo las funciones que se buscan en el programa. Se añaden dos librerías estándar de *C++* ya que este programa sí precisa de funciones como *scanf* o *switch.*

```
int main(int argc, char** argv)
{
  //Inicializacióon de ROS
  ros::init(argc, argv, "generar_fichero");
  ros::NodeHandle node_handle;
  ros::AsyncSpinner spinner(1);
  spinner.start();

  static const std::string PLANNING_GROUP = "manipulator";
  moveit::planning_interface::MoveGroupInterface
move_group(PLANNING_GROUP);
  moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;
  const robot_state::JointModelGroup* joint_model_group =
      move_group.getCurrentState()-
>getJointModelGroup(PLANNING_GROUP);

  namespace rvt = rviz_visual_tools;
  moveit_visual_tools::MoveItVisualTools visual_tools("base_link");
  visual_tools.deleteAllMarkers(); //Borramos marcadores
  visual_tools.loadRemoteControl();

  Eigen::Affine3d text_pose = Eigen::Affine3d::Identity();
  text_pose.translation().z() = 1.75;
  visual_tools.publishText(text_pose, "Inicio del programa",
rvt::WHITE, rvt::XLARGE);
  visual_tools.trigger();
```

Igualmente a los *headers* esto es muy similar a los programas utilizados para el entorno, se inicializa *ROS*, su *handler* y se inicializan las herramientas visuales de *Rviz.*

```
  int teclado;
  int acabar;
  std::vector<double> joint_group_positions;
  std::ofstream myfile;
```

Definición de las variables que se van a utilizar: *teclado* guardará el valor que se lea por pantalla, *acabar* determinará el final del programa*, joint_group_positions* almacenará el valor de las articulaciones del robot cuando se indique, *myfile* es el nombre de la variable que corresponde al fichero .txt que se utilizará.

```
ROS_INFO("Menu:");
ROS_INFO("1. Crear fichero de posiciones desde 0");
ROS_INFO("2. Agregar posiciones a fichero ya existente");
  scanf("%d", &teclado);
  acabar=0;
```

Esta parte del código muestra el menú que puede verse en la figura 3.30. Aquí también se leería el *input* por teclado para entrar en el menú que puede verse en la siguiente parte del código.

```
switch (teclado){
  case 1:
  {
      myfile.open("/home/claudia/joints.txt", std::ofstream::out |
std::ofstream::trunc);
      teclado=0;
      do{
      ROS_INFO("Pulsa 1 para incorporar una nueva posicion al
fichero");
      ROS_INFO("Pulsa 2 para salir");
      scanf("%d", &teclado);
      switch(teclado){
            case 1:{
            moveit::core::RobotStatePtr current_state =
move_group.getCurrentState();
            std::vector<double> joint_group_positions2;
            current_state->copyJointGroupPositions(joint_model_group,
joint_group_positions2);
            myfile << joint_group_positions2[0] << " " <<
joint_group_positions2[1] << " " << joint_group_positions2[2] << " "
<< joint_group_positions2[3] << " " << joint_group_positions2[4] << "
" << joint_group_positions2[5] << "\n";
            break;
            }
            case 2:
            myfile.close();
            acabar=1;
            break;
      }
      }while(acabar==0);
      break;
      }
```

Aquí, dentro del primer caso del *switch*, se crea el fichero desde 0, para después salir el segundo menú (figura 3.31) con el que se entra en el segundo *switch.* Dentro de éste, en el caso 1 se puede encontrar el código que lee la posición en la que se encuentra el robot en el momento en el que se pulsa el botón gracias a las funciones que vienen dadas por los *headers* de *MoveIt!.* En este caso dentro del objeto *move_group* existe la posibilidad de leer el valor de las posiciones de sus articulaciones con el operador *getCurrentState().* Este se almacena en otro vector y se copia en el fichero *joints.txt*. El caso 2 cambia de valor la variable *acabar* y finalizaría el programa.

```
      case 2:
      {
      myfile.open("/home/claudia/joints.txt", std::ios::app);
      teclado=0;
      do{
      ROS_INFO("Pulsa 1 para incorporar una nueva posicion al
fichero");
      ROS_INFO("Pulsa 2 para salir");
```

```
      scanf("%d", &teclado);
      switch(teclado){
            case 1:{
            moveit::core::RobotStatePtr current_state =
move_group.getCurrentState();
            std::vector<double> joint_group_positions2;
            current_state->copyJointGroupPositions(joint_model_group,
joint_group_positions2);
            myfile << joint_group_positions2[0] << " " <<
joint_group_positions2[1] << " " << joint_group_positions2[2] << " "
<< joint_group_positions2[3] << " " << joint_group_positions2[4] << "
" << joint_group_positions2[5] << "\n";
            break;
            }
            case 2:
            myfile.close();
            acabar=1;
            break;
      }
      }while(acabar==0);
      break;
      }
```

Muy similar al caso anterior, sin embargo, en este tramo de código no se borraría el contenido del archivo *joints.txt* de forma previa, sino que está configurado para que añada información a la ya existente.

```
}

  ros::shutdown();
  return 0;
}
```

Y por último, como en el caso de los programas del entorno, se cerrarían las funciones de *ROS* y se haría el *return 0* para finalizar el programa.

3.10.2 Programa que lleva a cabo los movimientos

Este segundo programa lleva el robot a las posiciones que se han definido previamente en el programa explicado anteriormente. Para ejecutar este programa hay que lanzar el siguiente comando en la terminal:
rosrun movimiento_ur3 programa_movimientos_fichero
De nuevo, como en el caso anterior, se supone que ya hay lanzados otros programas de *ROS* (y por tanto un *master*) que están conectados, o bien con el robot o con la simulación en *Gazebo*, ya que sino el programa mostará errores por pantalla.

Cuando se ejecuta el programa, lo primero que puede verse es una indicación por pantalla que le pide al usuario que indique cuál es el tiempo de espera entre los movimientos, en segundos. Cuando se indica esto, lo siguiente que aparece es el menú que puede verse en la figura 3.33 adjunta a continuación:

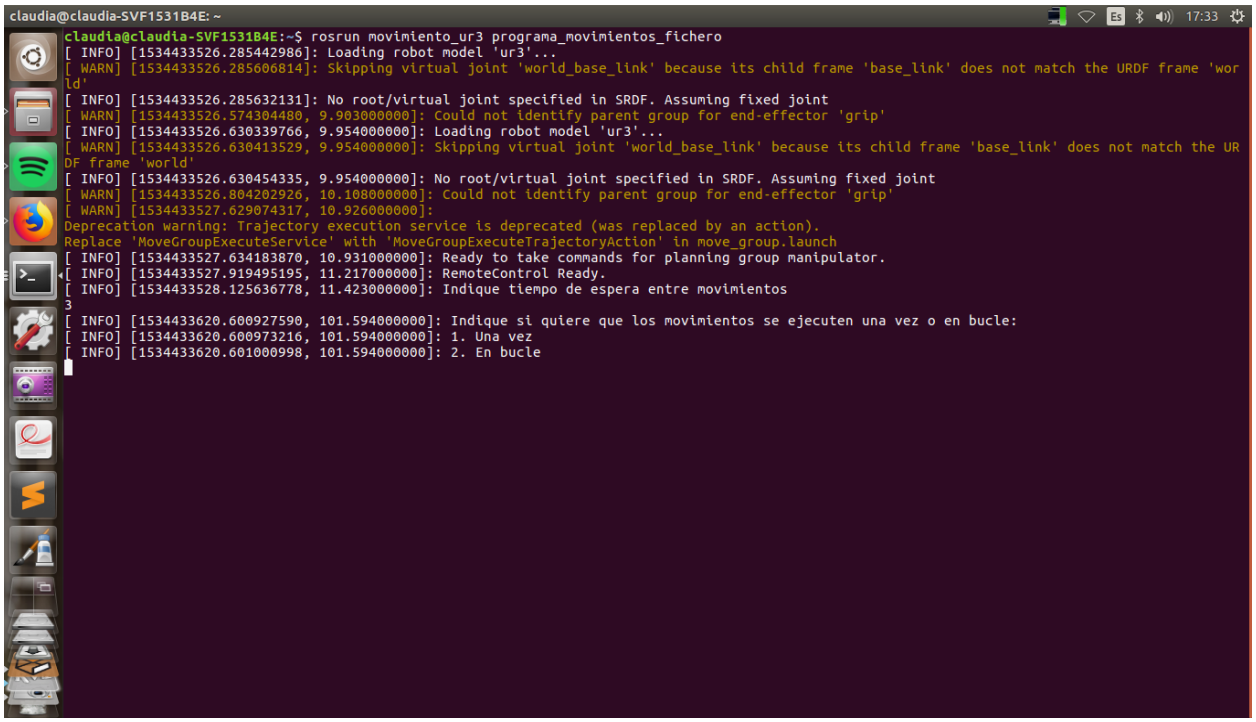

Figura 3.33: Menú de *programa_movimientos_fichero.*

Puede observarse que se le da la opción al usuario de ejecutar el movimiento o bien una vez, o bien de forma cíclica formando un bucle. La primera opción se ha colocado para pruebas y ver que la ejecución de ese bucle no supondrá ningún problema, y la segunda ya es el modo operativo que suele buscarse en este tipo de aplicaciones.

Cuando se selecciona cualquiera de los dos modos, se indica por pantalla a qué número de posición se va a mover el robot, y en caso de que por algún motivo no pueda ejecutarse el movimiento, o bien errores en la planificación, o bien uno de los objetos se ha desplazado al punto en final de la trayectoria, aparece un error indicando que no ha sido posible alcanzar el punto final objetivo indicado.

Ahora, como en el caso anterior, se adjuntará el código explicado a continuación para entender su funcionamiento:

```
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>

#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

#include <moveit_visual_tools/moveit_visual_tools.h>
#include <fstream>
#include <string>

int main(int argc, char** argv)
{
```

```
  ros::init(argc, argv, "programa_movimientos_fichero");
  ros::NodeHandle node_handle;
  ros::AsyncSpinner spinner(1);
  spinner.start();

  static const std::string PLANNING_GROUP = "manipulator";
  moveit::planning_interface::MoveGroupInterface
move_group(PLANNING_GROUP);
  moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;
  const robot_state::JointModelGroup* joint_model_group =
      move_group.getCurrentState()-
>getJointModelGroup(PLANNING_GROUP);

  namespace rvt = rviz_visual_tools;
  moveit_visual_tools::MoveItVisualTools visual_tools("base_link");
  visual_tools.deleteAllMarkers(); //Borramos marcadores
  visual_tools.loadRemoteControl();

  Eigen::Affine3d text_pose = Eigen::Affine3d::Identity();
  text_pose.translation().z() = 1.75;
  visual_tools.publishText(text_pose,    "Inicio    del    programa",
rvt::WHITE, rvt::XLARGE);   visual_tools.trigger();
```

Como puede verse, el inicio el código resulta muy similar a los explicados anteriormente, por lo que no se entrará en detalle sobre sus funciones.

```
  ROS_INFO("Indique tiempo de espera entre movimientos");
  int teclado;
  scanf("%d", &teclado);

  std::vector<std::vector<double>> file_read;
  std::vector<double> joint_group_positions;

  std::ifstream infile("/home/claudia/joints.txt");
  std::string line;
```

En este fragmento del código crea el mensaje que pide al usuario que introduzca el valor del tiempo (en segundos) de espera entre movimientos y lee dicho *input*. Posteriormente, crea el objeto en el que se va a guardar el fichero, el objeto que se corresponderá a una línea del fichero y el vector donde se almacenarán los valores de las posiciones. De forma adicional se crea una matriz que almacenará valores de tipo *double* que se utilizará para ordenar los valores leídos en el fichero *joints.txt*.

```
  while (std::getline(infile, line))
  {
      std::istringstream iss(line);
      double j0, j1, j2, j3, j4, j5, j6;

      iss >> j0 >> j1 >> j2 >> j3 >> j4 >> j5;
      joint_group_positions.clear();
      joint_group_positions.push_back(j0);
      joint_group_positions.push_back(j1);
```

```
      joint_group_positions.push_back(j2);
      joint_group_positions.push_back(j3);
      joint_group_positions.push_back(j4);
      joint_group_positions.push_back(j5);

      file_read.push_back(joint_group_positions);
  }
```

En este *while* se lleva a cabo la lectura del fichero línea a línea, de forma que cada valor de articulación que se encuentra se guarda en las variables de la *j0* a *j5*. Estas, posteriormente se guardan en el vector *joint_group_positions* que se formatea en cada iteración del bucle, para al final, guardarse como filas de la matriz *file_read* creada inicialmente. Esto permite que el programa se ajuste a un número indeterminado de posiciones, ya que no hay forma de saber cuántas posiciones se han definido de forma previa, maximizando, así, la flexibilidad del programa.

```
 int size;
 size=file_read.size();
 infile.close();
```

Aquí, con el fin de conocer el número de posiciones que existen, al terminar de leer el fichero, se cierra el archivo *.txt* y se almacena en la variable *size* el tamaño del vector.

```
 ROS_INFO("Indique si quiere que los movimientos se ejecuten una vez o
en bucle:");
 ROS_INFO("1. Una vez");
 ROS_INFO("2. En bucle");
 int teclado2;
 scanf("%d", &teclado2);
```

De forma similar al caso anterior, aquí se tiene el menú que aparece al usuario preguntando si busca ejecutar este programa una vez o de forma cíclica, además del *scanf* que permite almacenar en el programa la selección que se haya realizado.

```
 for(int i=0; i<size; i++){
       ROS_INFO("Posicion: %d", i+1);

  moveit::core::RobotStatePtr              current_state              =
move_group.getCurrentState();
  std::vector<double> joint_group_positions2;
  current_state->copyJointGroupPositions(joint_model_group,
joint_group_positions2);
  joint_group_positions2[0] = file_read.at(i).at(0);
  joint_group_positions2[1] = file_read.at(i).at(1);
  joint_group_positions2[2] = file_read.at(i).at(2);
  joint_group_positions2[3] = file_read.at(i).at(3);
  joint_group_positions2[4] = file_read.at(i).at(4);
  joint_group_positions2[5] = file_read.at(i).at(5);
  move_group.setJointValueTarget(joint_group_positions2);
```

En esta sección del código puede verse que es el inicio del bucle *for* que recorrerá la matriz *file_read* en la que se encuentran las posiciones. Se sabe que cada fila de la matriz es una posición. Por lo que se indica por pantalla a qué posición se va a ir, para posteriormente

almacenar esta posición dentro del vector *joint_group_positions2.* Cuando se han almacenado los seis valores de las articulaciones, se establece esta posición como *target* con la operación *setJointValueTarget* para el objeto *move_group,* que como ya se ha comentado antes, ya tiene en cuenta que se está trabajando con el grupo de articulaciones "*manipulator*").

```
  moveit::planning_interface::MoveGroupInterface::Plan my_plan;

  bool success = (move_group.plan(my_plan) ==
moveit::planning_interface::MoveItErrorCode::SUCCESS);
  if(success)
  move_group.move() ;
  else
  ROS_INFO("Imposible alcanzar la posicion");
```

Posteriormente, se realiza la planificación del movimiento con el operador *Plan.* Si este sale como exitoso, es decir, el planificador ha determinado que es posible alcanzar la posición que se ha pedido, almacena *true* en la variable *success*, en caso contrario almacena *false.* Si el resultado ha sido positivo, el condicional que se ha adjuntado después de la planificación lanza la orden a *MoveIt!* de mover el robot a la posición indicada. Si, por el contrario, el resultado de la planificación ha sido negativo, aparece un mensaje en la pantalla del programa que indica que dicha posición no puede alcanzarse.

```
  visual_tools.deleteAllMarkers();
visual_tools.publishTrajectoryLine(my_plan.trajectory_,
joint_model_group);
  ros::Duration(teclado).sleep();
```

En este fragmento, las dos primeras líneas se utilizan para que pueda verse en la *GUI* el resultado de la planificación y el movimiento del robot*,* y por último, la última línea de que define este trozo del código es la espera entre movimientos, que como puede verse está en función del valor que se ha indicado al principio del programa que se almacenó en la variable *teclado.*

```
 if(teclado2==2 && i==size-1){
      i=-1;
  }
  }
```

La función de este condicional es el que permite que el programa pueda ejecutarse de forma cíclica, ya que establece el contador del *for* en -1 cuando este está a punto de acabarse y cuando se ha seleccionado 2 en el menú inicial.

```
  ros::shutdown();
  return 0;
}
```

Por último, como en todos los programas, se cerraría la funcionalidad de *ROS* y el programa.

CAPÍTULO 4. CONCLUSIONES Y TRABAJOS FUTUROS.

4.1 RESUMEN DEL TRABAJO LLEVADO A CABO

Para concluir este documento de memoria y como resumen del trabajo llevado a cabo se repasarán los objetivos establecidos en la introducción del mismo para justificar si éstos se han cumplido:

- Se ha comprendido y utilizado correctamente tanto el entorno de *Ubuntu*, como *ROS* y las distintas herramientas que éste ofrece, puesto que el trabajo se ha podido llevar a cabo. Lo mismo ocurre con el *software* de *MoveIt!*.
- Se ha creado de forma fidedigna el modelo que corresponde al robot que se encuentra en el laboratorio respetando las dimensiones y las formas de los elementos añadidos.
- El paquete de configuración de *MoveIt!* se ha generado correctamente, pudiendo así controlar el robot con el mismo, tanto a nivel de simulación como con la instalación real.
- Se ha definido de forma sencilla el entorno de trabajo del robot y se ha generado a su vez un objeto que se mueve dentro de este entorno. A su vez, los movimientos del robot se ha probado que detectan los objetos y los tienen en cuenta puesto que se evitan.
- Se ha conseguido mediante la utilización, tanto de *ROS* como de *OpenCV* un programa que realiza la configuración de una cámara y tras esto es capaz de procesar un vídeo que se está captando a tiempo real de forma que siga un objeto de un determinado color, extraiga las coordenadas de su centroide y lo envíe vía *topic* al programa que genera el objeto en movimiento para que ambos se correspondan.
- Se ha creado una pareja de programas que permiten de forma muy sencilla, mover el robot a posiciones concretas de forma manual, y posteriormente que el robot las lleve a cabo de forma automática y cíclica.

Todos estos programas, si se utilizan de forma conjunta permiten lo que se buscaba en los objetivos generales, es decir, buscar una forma sencilla de programar el robot que permita que los movimientos del mismo tengan en cuenta el entorno y esquiven los elementos físicos que le rodean. Se puede decir, por tanto que se han cumplido todos los objetivos establecidos.

De forma adicional, cabe añadir, que gracias a la forma en la que están diseñados, tanto *MoveIt!* como *ROS*, si se colocara otro robot que fuera compatible con *ROS*, todos los programas que se han generado con el objetivo de este proyecto también serían válidos para el mismo con las modificaciones mínimas.

4.2 DIFICULTADES ENCONTRADAS DURANTE LA REALIZACIÓN DEL PROYECTO.

También es interesante hacer cierta reflexión sobre las distintas dificultades encontradas a lo largo de la realización de este proyecto:

- Es necesaria la familiarización con el entorno de *Ubuntu*, si es la primera vez que se trabaja con él, puede resultar complicado y puede consumir mucho tiempo encontrar todos los elementos y sobre todo, el aprendizaje del uso de la terminal.
- El aprendizaje de *ROS* y sus distintos elementos puede resultar muy complicada, puesto que, a pesar de que existen numerosos tutoriales, la información que tienen de los distintos conceptos de *ROS* no está tan completa como debiera, y para poder comprender completamente todo lo relacionado con lo que se quiere utilizar hay que recurrir a distintas páginas e incluso libros.
- En un primer lugar puede parecer que con conocer lenguajes de programación como *C++* o *Python* es suficiente, pero para desarrollar el modelo fue necesario el aprendizaje de otro lenguaje (*URDF*), que supuso un retraso también en el desarrollo del proyecto.
- La modificación que hubo que realizarse en el paquete de *ur_modern_driver* para que funcionase correctamente con la versión *Kinetic* de *ROS*, puesto que la que existía sólo era compatible con *Indigo*.

4.3 POSIBLES TRABAJOS FUTUROS.

Por último, a partir del desarrollo de este proyecto, es posible realizar trabajos adicionales que aumenten las funcionalidades de la aplicación realizada, por ejemplo, se puede realizar un programa que funcione con la cámara que realice un análisis más exhaustivo del objeto que se ha localizado, incluso realizando una clasificación. Con esta clasificación es posible generar el objeto en movimiento con unas características u otras, y con unos comportamientos u otros, es decir, distinguir si lo que se ha detectado es un objeto o una persona y definirlos de forma distinta a la hora de dibujarlos en el entorno.

Otra posibilidad es realizar un estudio de los distintos planificadores, sus diferencias a la hora de ejecutar el mismo camino, e incluso intentar implementar otros tipos de planificador que se encuentran en fase de pruebas en *MoveIt!* pero que pueden utilizarse, como pueden ser los planificadores de tipo *CHOMP* o *STOMP*, o incluso los de tipo controlador dentro de los *OMPL.*

CAPÍTULO 5. BIBLIOGRAFÍA.

5.1 DOCUMENTACIÓN

[1] Colaboradores de Wikipedia. Cobot [en línea] https://en.wikipedia.org/wiki/Cobot
[2] Robotiq. Getting Started With Colaborative Robots. [en línea] https://blog.robotiq.com/hubfs/eBooks/Getting-Started-with-Collaborative-Robots.pdf?hsLang=en-ca&t=1532968970867
[3] Universal Robots. Beneficios de los robots colaborativos. [en línea] https://www.universal-robots.com/es/acerca-de-universal-robots/beneficios-de-los-robots-colaborativos/
[4] Mathieu Bélanger-Barrette. What Does Collaborative Robot Mean? Robotiq blog. [en línea] https://blog.robotiq.com/what-does-collaborative-robot-mean
[5] Patti Robotics. 4 Types of Collaborative Robots to Increase Productivity. [en línea] http://pattiengineering.com/blog/4-types-collaborative-robots/
[6] International Standard Organization. Robots and robotic devices -- Safety requirements for industrial robots -- Part 1: Robots. [en línea] https://www.iso.org/standard/51330.html
[7] Ángel Valera. Apuntes de Robótica Industrial para el máster de Ingeniería Industrial, tema 4: Programación y simulación de robots industriales. 2ª parte: Robot colaborativo UR3, curso 2017-2018.
[8] Universal Robots. Manual de programación de Polyscope. [en línea, disponible para descarga] https://www.universal-robots.com/download/?option=40148#section40113
[9] Universal Robots. The URScript Programming Language. [en línea, disponible para descarga] https://www.universal-robots.com/download/?option=40154#section39925
[10] Robot Operating System. About ROS. [en línea] https://www.ros.org/about-ros/
[11] WikiROS. Introduction to ROS. [en línea] http://wiki.ros.org/ROS/Introduction
[12] WikiROS. Concepts [en línea] http://wiki.ros.org/ROS/Concepts
[13] WikiROS. Topics [en línea] http://wiki.ros.org/Topics
[14] WikiROS. Services [en línea] http://wiki.ros.org/Services
[15] WikiROS. Actions [en línea] http://wiki.ros.org/actionlib
[16] WikiROS. roscore [en línea] http://wiki.ros.org/roscore
[17] WikiROS. Master [en línea] http://wiki.ros.org/Master
[18] WikiROS. rosout [en línea] http://wiki.ros.org/rosout
[19] MoveIt! Motion Planning Framework [en línea] https://moveit.ros.org/
[20] MoveIt! Concepts [en línea] http://moveit.ros.org/documentation/concepts/
[21] Morgan Quigley, Brian Gerkey & William D. Smart. *Programming robots with ROS.* O'Reilly Media 2013.
[22] WikiROS. Parameter Server [en línea] http://wiki.ros.org/Parameter%20Server
[23] WikiROS. URDF [en línea] http://wiki.ros.org/urdf
[24] WikiROS. SRDF [en línea] http://wiki.ros.org/srdf
[25] WikiROS. robot_state_publisher [en línea] http://wiki.ros.org/robot_state_publisher

[26] WikiROS. Using the robot state publisher on your own robot [en línea] http://wiki.ros.org/robot_state_publisher/Tutorials/Using%20the%20robot%20state%20publisher%20on%20your%20own%20robot
[27] WikiROS. tf [en línea] http://wiki.ros.org/tf
[28] Aaron Romero, Enrique Fernández. Learning ROS for Robotics Programming - Second Edition. Packt Publishing 2015.
[29] Octomap. Octomap. http://octomap.github.io/
[30] The Open Motion Planning Library. The Open Motion Planning Library [en línea] http://ompl.kavrakilab.org/
[31] The Open Motion Planning Library. Available Planners [en línea] ompl.kavrakilab.org/planners.html
[32] Colaboradores de Wikipedia. Rapidly-exploring random tree [en línea] https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree
[33] The Open Motion Planning Library. ompl::geometric::EST Class Reference [en línea] http://ompl.kavrakilab.org/classompl_1_1geometric_1_1EST.html#gEST
[34] Universal Robots. Especificaciones del UR3 [en línea] https://www.universal-robots.com/media/1801283/esp_semea_199912_ur3_tech_spec_web_a4.pdf
[35] KEYENCE America. PL (Performance Level) https://www.keyence.com/ss/products/safetyknowledge/performance/level/
[36] f2i2.net. Significado y explicación de los códigos IP, IK. http://www.f2i2.net/documentos/lsi/rbt/guias/guia_bt_anexo_1_sep03R1.pdf
[37] Grupo Europeo de Ingenieria Agroalimentaria y Ambiental. Clasificación de las salas blancas [en línea]. http://www.gei-2a.com/rcs/GEI-2A_clasificacion_salas_blancas.pdf
[38] Robotiq. Wrist Camera Specifications [en línea]. https://assets.robotiq.com/production/support_documents/document/online/Vision_System_HTML5_20180509.zip/Vision_System_HTML5/Default.htm?_ga=2.52990952.1133663452.1533639420-555183071.1529320580#Vision_System-Instructions-Manual-06-2017/7.%20Specifications.htm#7.1._Mechanical_Specifications_of_Wrist_Camera%3FTocPath%3DSpecifications%7C7.%2520Specifications%7C7.1.%2520Mechanical%2520Specifications%2520of%2520Wrist%2520Camera%7C_____0
[39] Festo. Especificaciones pinzas Festo DHPS [en línea] https://www.festo.com/cat/en-gb_gb/data/doc_ES/PDF/ES/DHPS_ES.PDF
[40] Amazon.es. Sony Vaio SVF1531B4E [en línea] https://www.amazon.es/Sony-Vaio-SVF1531B4E-Port%C3%A1til-GeForce/dp/B00FYIY1RG#productDetails
[41] Colaboradores de github. Branch for Kinetic? Issue #58 [en línea] https://github.com/ThomasTimm/ur_modern_driver/issues/58
[42] WikiROS. xacro [en línea]. https://wiki.ros.org/xacro
[43] WikiROS. urdf/ XML/ link [en línea] http://wiki.ros.org/urdf/XML/link
[44] WikiROS. urdf/ XML/ joint [en línea] http://wiki.ros.org/urdf/XML/joint
[45] Colaboradores de Wikipedia. HSV. https://en.wikipedia.org/wiki/HSL_and_HSV
[46] Quora. What are the differences between RGB, HSV and CIE-Lab? https://www.quora.com/What-are-the-differences-between-RGB-HSV-and-CIE-Lab
[47] Antonio-José Sánchez-Salmerón. Tema: Extracción de características, apuntes de la asignatura de visión artificial curso 2017-2018.
[48] Colaboradores de Wikipedia. Image Moments. https://en.wikipedia.org/wiki/Image_moment
[49] Colaboradores de Wikipedia. OpenCV. https://en.wikipedia.org/wiki/OpenCV
[50] OpenCV. OpenCV Library. https://opencv.org/
[51] Antonio-José Sánchez-Salmerón. Tema: Reconstrucción tridimensional, apuntes de la asignatura de visión artificial curso 2017-2018.

5.2 IMÁGENES

[1] Búsqueda google images "UR3" http://www.directindustry.es/prod/universal-robots-s/product-101499-1628337.html
[2] Björn Matthias. Industrial Safety Requirements for Collaborative Robots and Applications. [en línea] https://www.roboticsbusinessreview.com/wp-content/uploads/2016/05/Industrial_HRC_-_ERF2014.pdf
[3] Ángel Valera. Apuntes de Robótica Industrial para el máster de Ingeniería Industrial, tema 4: Programación y simulación de robots industriales. 2ª parte: Robot colaborativo UR3, curso 2017-2018.
[4] Universal Robots. Manual de programación de Polyscope. [en línea, disponible para descarga] https://www.universal-robots.com/download/?option=40148#section40113
[5] Moveit! Concepts. http://moveit.ros.org/documentation/concepts/
[6] Universal Robots. Especificaciones del *UR3* https://www.universal-robots.com/media/1801283/esp_semea_199912_ur3_tech_spec_web_a4.pdf
[7] Robotiq. Wrist Camera Specifications. https://assets.robotiq.com/production/support_documents/document/online/Vision_System_HTML5_20180509.zip/Vision_System_HTML5/Default.htm?_ga=2.52990952.1133663452.1533639420-555183071.1529320580#Vision_System-Instructions-Manual-06-2017/7.%20Specifications.htm#7.1._Mechanical_Specifications_of_Wrist_Camera%3FTocPath%3DSpecifications%7C7.%2520Specifications%7C7.1.%2520Mechanical%2520Specifications%2520of%2520Wrist%2520Camera%7C_____0
[8] Festo. Especificaciones pinzas Festo DHPS. https://www.festo.com/cat/en-gb_gb/data/doc_ES/PDF/ES/DHPS_ES.PDF
[9] Búsqueda google imágenes "HSV color" https://en.wikipedia.org/wiki/HSL_and_HSV
[10] Antonio-José Sánchez-Salmerón. Tema: Reconstrucción tridimensional, apuntes de la asignatura de visión artificial curso 2017-2018.

ANEXO I: MANUAL DE INSTALACIÓN. INSTRUCCIONES Y RECOMENDACIONES DE USO.

1.MANUAL DE INSTALACIÓN.

Se supone en este manual que se parte de un ordenador con Ubuntu 16.04 y *ROS Kinetic* instalado y operativo. De no ser así, se puede instalar de forma muy sencilla siguiendo los pasos indicados en el siguiente enlace:

http://wiki.ros.org/kinetic/Installation/Ubuntu

En primer lugar, es necesario crear el directorio en el que se guardarán los distintos paquetes, ya sean nuevos o descargados. Para esto, es necesario generar una nueva carpeta, a la que se llamará *catkin_ws*. Dentro de ésta, se crea otro directorio: *src* en el que se guardarán los paquetes que contienen los distintos programas que se utilizarán en esta aplicación. Para convertir esta carpeta en un *workspace* de *ROS* propiamente dicho, es necesario aplicarle el comando *catkin_make*. Esto se hace vía comandos a través de la terminal de *Ubuntu*. Habría que situarse en la carpeta (cd ~/catkin_ws) y ejecutar *catkin_make.* Este comando también se tendrá que ejecutar de la misma forma cada vez que se quieran compilar los distintos códigos de los paquetes.

El siguiente paso es más bien una recomendación, ya que para que *ROS* encuentre los distintos archivos y *workspaces* es necesario cargar el fichero *setup.bash*, por lo que si sólo se tiene una carpeta en la que se ha aplicado *catkin_make* lo que se recomienda es añadir la línea: "source ~/catkin_ws/devel/setup.bash" al fichero *.bashrc*. Para esto basta con lanzar el siguiente comando en la terminal:

```
echo "source ~/catkin_ws/devel/setup.bash" >> /.bashrc.
```

Si se opta por no hacer esto, habría que lanzar cada vez que se abre una nueva terminal:

```
source ~/catkin_ws/devel/setup.bash.
```

Una vez se tiene el *workspace* configurado, ya se puede empezar a usar paquetes que lleven a cabo las funcionalidades buscadas. Estos paquetes pueden, o bien descargarse desde internet, o bien pueden crearse desde cero. Si se descargan desde internet (cosa que ocurrirá para la mayoría de ellos), existen dos opciones, o bien se descargan sus ficheros y se compilan desde el ordenador, o bien se instalan directamente como si de un programa se tratara.

Ambos casos ofrecen ventajas e inconvenientes, ya que el primer caso permite que se pueda modificar el código antes de compilar ya que se tiene acceso a él, sin embargo, es posible que a la hora de realizar esto devuelva algún tipo de error debido a que faltan dependencias. Estas dependencias pueden instalarse de forma automática con el comando *rosdep*, se utilizaría de la siguiente forma:

"rosdep install nombre-del-paquete".

También existe la posibilidad de buscar e instalar las dependencias de todo el *workspace.* Esto se haría con el siguiente comando en la terminal:

"rosdep install --from-paths src --ignore-src -r -y".

Si se descarga como un programa normal, las dependencias que tendría este paquete se instalan automáticamente, pero no se puede acceder al código. Para realizar este tipo de instalación basta con escribir en la terminal:

"sudo apt-get install ros-kinetic-nombre-del-paquete".

Es recomendable que el usuario considere qué tipo de instalación le interesa más antes de realizarla.

Para la instalación de esta aplicación se recomienda lo siguiente: los paquetes más básicos como puede ser el *Moveit!* que se instalen de la segunda forma y que paquetes como los de *Universal Robots* sean instalados desde el código base.

Así pues, para instalar *Moveit!* se ejecuta el siguiente comando:

"sudo apt-get install ros-kinetic-moveit".

Los paquetes relacionados con el *UR3* son el paquete *Universal_Robots* y el paquete *Ur_modern_driver.* Habría que descargarlos con el primer método, puesto que están alojados en *github.* Por lo tanto, es recomendable antes instalar *git* para *Ubuntu* (*sudo apt-get install git-core*). Primero se clonarían los repositorios en la carpeta *src* del *workspace* antes creado, esto es, se situaría la terminal en *src* (*cd catkin_ws/src)* y se copiarían con los siguientes comando:

"git clone https://github.com/ros-industrial/universal_robot.git"

"git clone https://github.com/ThomasTimm/ur_modern_driver.git"

En este momento, antes de lanzar el comando *catkin_make* para realizar la compilación, se recomienda comprobar que todas las dependencias están instaladas (aquí se lanzaría en la terminal: "*rosdep install --from-paths src --ignore-src -r -y*"). Además, si se está utilizando *ROS Kinetic* es necesario llevar a cabo una modificación en el código fuente de uno de los nodos de *ur_modern_driver* para que éste pueda compilar ya que está diseñado para *ROS Indigo*. Sería dentro del nodo: hardware_interface.cpp, sustituir cada "hardware_interface" que aparece en el código, por "claimed_resources.at(0).hardware_interface".

De forma adicional, también se añade cómo realizar la instalación de *OpenCV* en *Ubuntu 16.04* ya que puede resultar interesante para trabajos futuros, y algunos de los paquetes que se han desarrollado, aunque en menor medida, precisan de algunas de las librerías que ofrece. Previamente a la descarga de *OpenCV* como tal, ha sido necesario instalar algunas librerías a través de la terminal:

sudo apt-get install build-essential checkinstall cmake pkg-config yasm

sudo apt-get install build-essential

```
sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev
libavformat-dev libswscale-dev

sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev
libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev

sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev
libv4l-dev

sudo apt-get install --assume-yes build-essential cmake git

sudo apt-get install --assume-yes pkg-config unzip ffmpeg qtbase5-dev
python-dev python3-dev python-numpy python3-numpy

sudo apt-get install --assume-yes pkg-config unzip ffmpeg qtbase5-dev
python-dev python3-dev python-numpy python3-numpy

sudo apt-get install --assume-yes pkg-config unzip ffmpeg qtbase5-dev
python-dev python3-dev python-numpy python3-numpy

sudo apt-get install --assume-yes libopencv-dev libgtk-3-dev
libdc1394-22 libdc1394-22-dev libjpeg-dev libpng12-dev libtiff5-dev
libjasper-dev

sudo apt-get install --assume-yes libavcodec-dev libavformat-dev
libswscale-dev libxine2-dev libgstreamer0.10-dev libgstreamer-plugins-
base0.10-dev

sudo apt-get install --assume-yes libv4l-dev libtbb-dev libfaac-dev
libmp3lame-dev libopencore-amrnb-dev libopencore-amrwb-dev libtheora-
dev

sudo apt-get install --assume-yes libvorbis-dev libxvidcore-dev v4l-
utils vtk6

sudo apt-get install --assume-yes liblapacke-dev libopenblas-dev
libgdal-dev checkinstall
```

Una vez tenemos estas librerías, sería necesario descargar el siguiente fichero:

https://github.com/opencv/opencv/archive/3.3.0.zip

Una vez se ha obtenido, éste se descomprimiría en la carpeta personal, y se lanzarían los siguientes comandos en la terminal:

```
cd opencv-3.3.0/

mkdir build

cd build/

cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -
D FORCE_VTK=ON -D WITH_TBB=ON -D WITH_V4L=ON -D WITH_QT=ON -D
WITH_OPENGL=ON -D WITH_CUBLAS=ON -D CUDA_NVCC_FLAGS="-D_FORCE_INLINES"
-D WITH_GDAL=ON -D WITH_XINE=ON -D BUILD_EXAMPLES=ON ..
```

```
make -j $(($(nproc) + 1))

sudo make install

sudo /bin/bash -c 'echo "/usr/local/lib" >
/etc/ld.so.conf.d/opencv.conf'

sudo ldconfig

sudo apt-get update
```

Con esto, ya se tendrían las librerías de *OpenCV* disponibles. Por último, para que puedan ser integradas a los programas desarrollados con *ROS* es necesario descargar el paquete *CVBridge*. Se descarga de la misma manera que los paquetes de *Universal Robots*:

"git clone https://github.com/ros-perception/vision_opencv.git"

Por último, habría que instalar los paquetes que se han creado como objetivo de este trabajo. Son los siguientes: *entornos, movimientos_ur3, ur3_grip_moveit_config, ur_description_mod, ur_gazebo_mod* y *vision_pruebas*. Puesto que existe una comunicación creada a través de un *message* que requiere de un archivo de tipo *header* que no existe hasta que no se compila por primera vez, es necesario realizar los siguientes pasos:

1. Dentro del paquete *vision_pruebas*, en el fichero *CMakeLists.txt*, habría que comentar (añadiendo # antes de cada línea) la siguiente parte del texto para que el compilador no busque estos archivos:
   ```
   add_executable(talker src/talker.cpp)
   target_link_libraries(talker ${catkin_LIBRARIES}
   ${OpenCV_LIBRARIES})
   add_dependencies(talker
   beginner_tutorials_generate_messages_cpp)
   ```

2. De forma similar, en el paquete *entornos* dentro de *CMakeLists.txt* se comentaría la parte del fichero correspondiente a la compilación del nodo *objeto_movimiento.cpp* que requiere de este *header* también:
   ```
   add_executable(objeto_movimiento src/objeto_movimiento.cpp)
   target_link_libraries(objeto_movimiento ${catkin_LIBRARIES}
   ${Boost_LIBRARIES})
   install(TARGETS objeto_movimiento DESTINATION
   ${CATKIN_PACKAGE_BIN_DESTINATION})
   add_dependencies(objeto_movimiento
   ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
   ```

3. Una vez está esto comentado, se realizaría la creación del fichero. Para esto, se sitúa la terminal en el *workspace* y en lugar de realizar *catkin_make*, se lanza *catkin_build*. Esto crea el fichero .h que se necesita a partir del mensaje que está creado ya en la carpeta *vision_pruebas*.

4. Cuando ha terminado el *build* se revierten los comentarios realizados en ambos *CMakeLists.txt*.

5. Se compila como es habitual con *catkin_make.*

Con estos paquetes, ya se tendría el robot funcionando correctamente.

2.INSTRUCCIONES DE USO.

Modo simulación:

Para simular el robot, se trabaja con *Gazebo* y *Rviz.* En primer lugar tendría que lanzarse:

```
roslaunch ur_gazebo_mod ur3.launch.
```

Esto abriría *Gazebo* y cargaría el modelo del *UR3* modificado con la pinza ya incorporada.

Después habría que lanzar los siguientes*.launch*:

```
roslaunch ur3_grip_moveit_config ur3_grip_moveit_planning_execution.launch sim:=true

roslaunch ur3_grip_moveit_config moveit_rviz.launch config:=true
```

Sin embargo, para no tener que estar pendiente de los distintos argumentos y sólo tener que lanzar un fichero, se recomienda, en caso de que no exista, crear un nuevo *.launch* que abra los dos anteriores con sus respectivos argumentos ya definidos. Este archivo tendría la siguiente forma:

```
<launch>

<include file="$(find ur3_grip_moveit_config)/launch/ur3_grip_moveit_planning_execution.launch">

      <arg name="sim" default="true" />

  </include>

<include file="$(find ur3_grip_moveit_config)/launch/moveit_rviz.launch">

      <arg name="config" default="true"/>

  </include>

<include file="$(find entornos)/launch/entorno_def.launch">

  </include>

</launch>
```

Además este *simulacion.launch* permite, a la vez, cargar las geometrías que se diseñaron para definir el entorno.

Si a la hora de ejecutar *Rviz* la terminal en cuestión da algún tipo de error, es posible que sea debido a la configuración de la tarjeta gráfica. Se recomienda en ese caso escribir el siguiente comando al abrir la terminal en cada ocasión para evitar dichos errores:

```
export LIBGL_ALWAYS_SOFTWARE=1
```

Una vez está todo cargado en el ordenador tendrían que estar abiertas dos pantallas: una de *Gazebo* con el modelo del robot cargado y una de *Rviz* con el modelo del robot y el entorno cargado como puede verse en la figura 6.1 inferior.

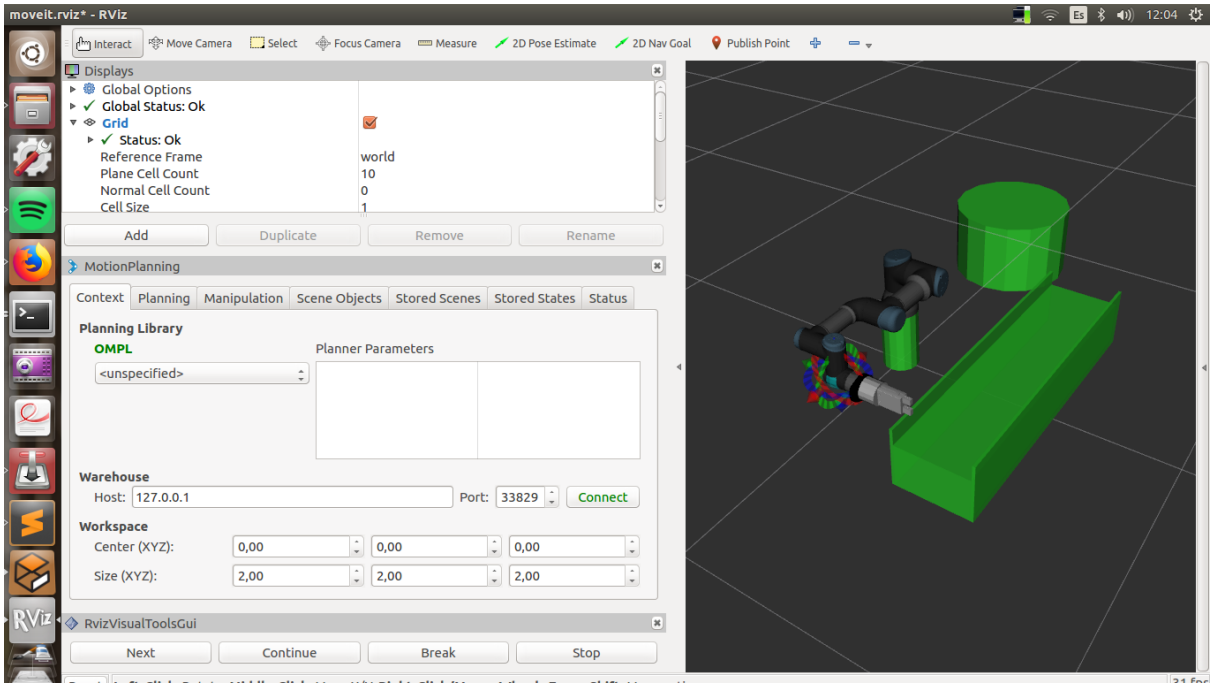

Figura 6.1: Modelo del robot en *Rviz* con su entorno.

En este momento se puede cargar, si se desea, el objeto extra que se mueve por el entorno de trabajo del robot. El programa se lanzaría:

```
rosrun entornos objeto_movimiento
```

Seleccionando *n* cuando pregunta si hay un sistema de visión, e indicando por pantalla el tiempo de espera entre movimientos del objeto (para determinar la velocidad del mismo), el objeto se crearía en medio de la "*cinta*" y se desplazaría sobre ella.

Por último, para hacer funcionar los programas de movimiento del robot, en primer lugar habría que ejecutar el que corresponde al generador de los puntos que va a seguir el robot. Para esto se lanzaría:

```
rosrun movimiento_ur3 generar_puntos
```

Se da la opción de añadir puntos a un fichero anterior ya existente o crear uno nuevo. Se elige la opción que más se ajuste a lo que se busca. Ahora moviendo el robot, o bien a través de la consola de *Moveit! (rosrun moveit_commander moveit_commander_cmd.py)* o bien de forma manual interfaz gráfica de *Rviz* (las flechas que pueden verse en la figura *x* sobre la pinza del robot), se buscan los puntos a los que el robot debe desplazarse, pulsando 1 cuando esté en una posición deseada. Se repite hasta que están todos los puntos, entonces se pulsa 2 para finalizar.

Para que el robot se mueva siguiendo esta trayectoria hay que ejecutar el siguiente programa:

```
rosrun movimiento_ur3 programa_movimientos_fichero
```

Este, tras indicarle el tiempo que debe esperar entre movimientos, y si ha de ejecutar la secuencia una vez o en bucle, lee las posiciones generadas en con el programa anterior y calcula la trayectoria que ha de seguir para moverse de un punto a otro.

Modo robot real:

Para utilizar *ROS* con el robot real se hace de forma muy similar a la simulación, de hecho, los últimos pasos desde que se ha lanzado *Rviz* son completamente iguales.

La primera diferencia respecto al proceso anterior es que no hay que lanzar *Gazebo*, sino que hay que ejecutar el siguiente programa si la configuración previa del robot ya se ha realizado previamente:

```
roslaunch ur_modern_driver ur3_bringup_mod.launch
```

(El fichero .launch modificado no es original del paquete descargado, surge a partir del existente utilizando el modelo que sí incluye la pinza).

Si no se ha realizado la configuración previa del robot, hay que seguir los siguientes pasos:

1. Dentro del menú de *Polyscope*, se entra en la parte de configuración del robot, y se selecciona “Configurar Red” y aparece el siguiente menú:

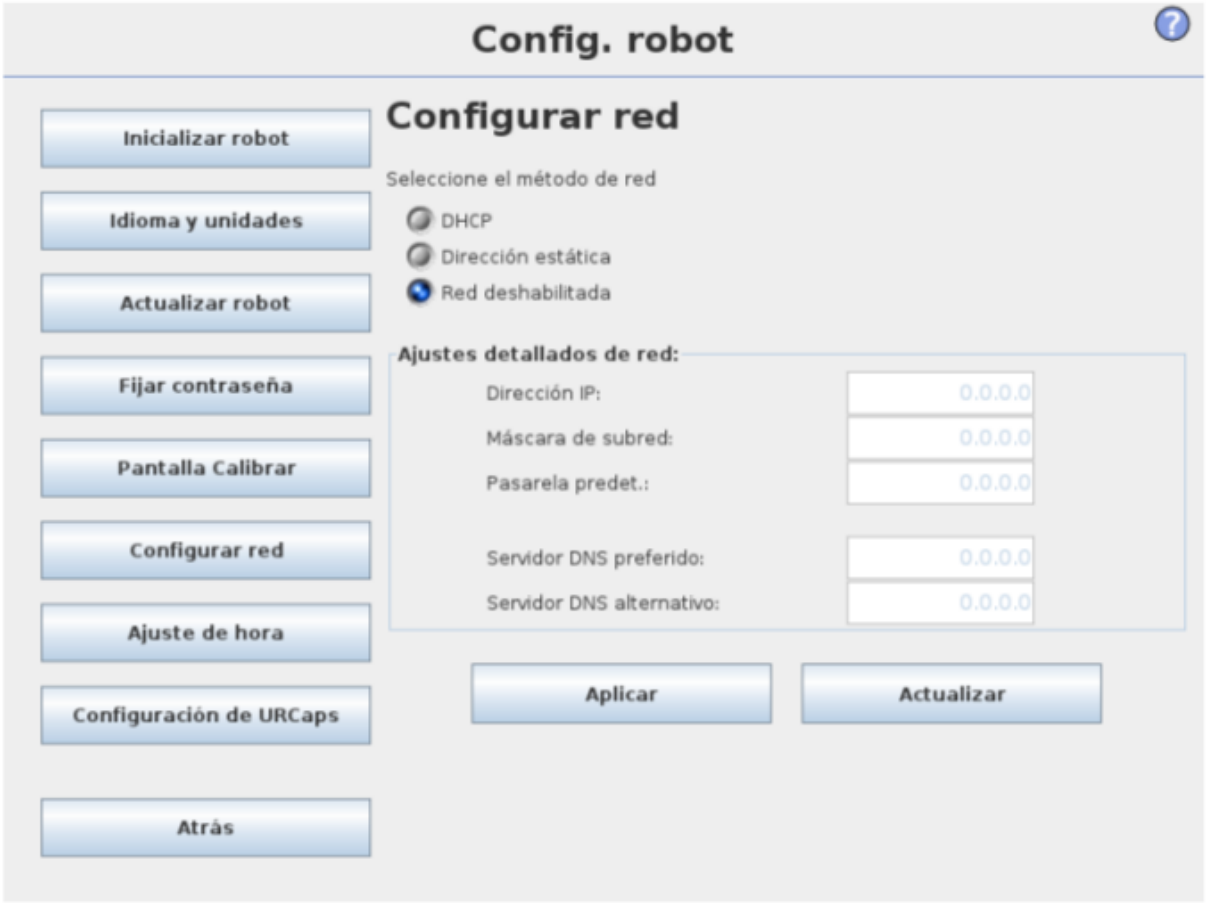

Figura 6.2: Menú de configuración de red de *Polyscope*

2. Una vez aquí, se selecciona la opción de "Dirección estática" y se indica una IP para el robot. En este caso es 158.42.206.10.

3. Se guarda la configuración, y si el robot ya se encuentra conectado al ordenador, se puede comprobar que la conexión existe haciendo: *ping 158.42.206.10*

Con esto, y el comando anterior, ya se puede controlar el robot real con *ROS.*

Para ejecutar *Moveit!* se hace de una forma muy similar a la anterior, se lanzan los siguientes comandos en la terminal:

```
roslaunch ur3_grip_moveit_config ur3_grip_moveit_planning_execution.launch
```

```
roslaunch ur3_grip_moveit_config moveit_rviz.launch config:=true
```

Y de nuevo se recomienda la creación, en caso de no existir de un archivo que sea capaz de llamar a esos dos ficheros y al del entorno para que se ejecuten al mismo tiempo. La sintaxis de dicho *.launch* sería la siguiente:

```
<launch>
```

```
<include                                                    file="$(find
ur3_grip_moveit_config)/launch/ur3_grip_moveit_planning_execution.laun
ch">

  </include>

<include                                                    file="$(find
ur3_grip_moveit_config)/launch/moveit_rviz.launch">

      <arg name="config" default="true"/>

  </include>

<include                                                    file="$(find
moveit_tutorials)/doc/move_group_interface/launch/entorno.launch">

  </include>

</launch>
```

A partir de este punto, se pueden seguir los pasos indicados en el modo de simulación.

DOCUMENTO DEL PRESUPUESTO

PRESUPUESTO

1. JUSTIFICACIÓN DEL PRESUPUESTO.

Antes de empezar el desglose del presupuesto como tal, es necesario hacer una breve introducción al mismo. A pesar de que el presente proyecto se trate de un trabajo de fin de máster con carácter puramente académico, es necesario conocer la dispensa económica que supondría la reproducción total o parcial de este proyecto. Es por tanto que en este presupuesto se expondrán los siguientes aspectos: el coste económico de la mano de obra necesaria para llevar a cabo este proyecto, el gasto necesario para adquirir los materiales que se han utilizado y por último, el coste del material de corta vida útil. De forma adicional se adjuntará un resumen con el que podrá verse el coste hipotético total de este proyecto de tratarse de un trabajo real.

Este presupuesto se ha realizado conforme a lo estipulado en el documento "RECOMENDACIONES EN LA ELABORACIÓN DE PRESUPUESTOS EN ACTIVIDADES DE I+D+I" del Centro de Apoyo a la Innovación, la Investigación y la Transferencia de Tecnología (CTT), utilizándose la versión referente a 2015 a la hora de tener en cuenta precios y desgloses. Se ha utilizado este formato de presupuesto debido al carácter del proyecto presente.

2. ESTUDIO ECONÓMICO.

2.1 Coste personal

Para el cálculo de los costes asociados al personal que ha intervenido en el proyecto se ha utilizado la siguiente fórmula:

$$Coste\ de\ personal\ (€) = Coste\ por\ hora\ \left(^{€}/_{h}\right) * número\ de\ horas\ dedicadas$$

A la hora de considerar el coste por hora, se ha tomado como referencia el precio de un titulado superior dentro de la categoría de personal eventual. En el documento del CTT se recomienda un salario comprendido entre 28.4 €/h y 41.2 €/h, por lo que para la elaboración de este presupuesto se ha tomado la media de ambos precios: 34.8 €/h. Este precio ya incluye tanto los costes indirectos como lo correspondiente a la Seguridad Social.

Costes de personal	Tiempo (h)	Coste (€/h)	Total (€)
Instalación Ubuntu	5	34,8	174
Instalación *ROS Kinetic*	20	34,8	696
Instalación de paquetes necesarios	10	34,8	348

Documentación y tutoriales	50	34,8	1740
Desarrollo de paquetes	100	34,8	3480
Pruebas con simulador	25	34,8	870
Pruebas con robot real	30	34,8	1044
Redacción de documentos	60	34,8	2088
Total	300		10440

Este coste total de personal, si se desglosa de acuerdo a los costes correspondientes a Seguridad Social (32.1%) y costes indirectos (16.5€/h), queda:

Concepto	Coste (€)
Honorarios	2138,76
Seguridad social (32,1%)	3351,24
Costes indirectos (16,5 €/h)	4950
Total	10440

2.2 Material inventariable

La expresión utilizada para el cálculo del coste del material inventariable utilizado también viene dada en el documento de recomendaciones y es la siguiente:

$$Coste\ material\ inventariable = \frac{Número\ meses\ de\ uso}{12\ {}^{meses}/_{año} * Periodo\ amortización} * Coste\ del\ equipo * porcentaje\ uso$$

Los periodos de amortización utilizados para el cálculo son los que vienen recomendados en el documento del CTT referenciado anteriormente, siendo 10 años para los equipos de docencia e investigación (que se han utilizado, sobre todo, el robot y los equipos adjuntos al mismo) y de 6 años para equipo informático y *software*.
El cálculo del porcentaje de uso se ha hecho teniendo en cuenta las tareas llevadas a cabo en la sección 2.1, viendo qué equipos se han utilizado para cada fase, sumando el número total de horas que ese equipo se ha utilizado y por último dividiéndola entre la duración total del proyecto.

Equipo	Tiempo de uso (meses)	Años amortización	Coste	Porcentaje uso	TOTAL
Ordenador Portátil Sony Vaio	4	6	769	100,00%	42,72
Ubuntu 16.04	4	6	0	100,00%	0,00
ROS Kinetic	4	6	0	78,33%	0,00
Simulador Rviz	4	6	0	68,33%	0,00
Simulador Gazebo	4	6	0	58,33%	0,00

Paquetes varios *ROS Kinetic*	4	6	0	71,67%	0,00
UR3	4	10	13500	26,67%	120,00
Punto acceso Wi-fi	4	6	25	100,00%	1,39
Robotiq Wrist Camera	4	10	4730	26,67%	42,04
Festo DHPS 20-a	4	10	650	26,67%	5,78
Cinta transportadora	4	10	3500	10,00%	11,67
Webcam	4	6	100	51,67%	2,87
TOTAL					206,16

2.3 Material fungible

En esta sección se tiene en cuenta el material de vida corta y que no puede reutilizarse para otros proyectos. En este caso es, sobre todo, referente a la impresión del documento de memoria y presupuesto.

Concepto	Precio (€)
Impresión	15
Encuadernación	5
Folios (500 uds)	4,64
Total	24,64

3. RESUMEN DEL PRESUPUESTO

Por último, se incluye el resumen del presupuesto que recopila el coste total de cada una de las partes expuestas anteriormente y realiza el cálculo total del mismo teniendo en cuenta la aplicación del IVA (21%).

Concepto	Total (€)
Mano de obra (300h)	10440
Material Inventariable	206,16
Material Fungible	24,64
Subtotal	10670,80
IVA (21%)	2240,87
TOTAL	12911,66