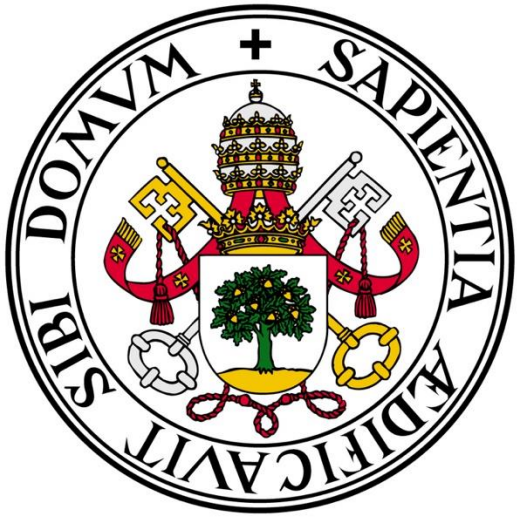
SIMULACIÓN ROBÓTICA COLABORATIVA CON ROS



UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería electrónica industrial y automática

Autor:

María Molpeceres Jorge

Tutor: Alberto Herreros López

Departamento: Ingeniería de Sistemas y Automática

Valladolid, junio de 2020.

Resumen:

La robótica colaborativa puede provocar un gran avance en el ámbito de la medicina, ocasionando un cambio en el modo de realizar las operaciones, como influyendo en su dificultad para hacerlas de manera mas sencilla.
Si a estos avances se le añade además la capacidad de un software de programación flexible y de código abierto como ROS, se pueden obtener muy buenos resultados.

En este proyecto se pretende conseguir que la operación de extraer tumores malignos del cerebro ubicados en la hipófisis, a través del orificio nasal se realice de manera mas sencilla para el profesional de la medicina, contando con el apoyo de dos robots colaborativos programados a través del sistema robótico ROS.

Palabras clave:

ROS, robots, simulación, operación, Python.

Tabla de contenido

1. Introducción y objetivos 6

2. Robots colaborativos 7

2.1. Normas y estándares para los robots colaborativos 8

2.2. UR3 10

2.2.1. Conceptos básicos del UR3 11

2.2.2. Programación UR3 13

3. Garra Robotiq 15

4. ROS (*Robot Operating System*) 17

4.1. Definición 17

4.2. Versiones de ROS 17

4.3. Instalación de ROS 20

4.4. Conceptos básicos de ROS 22

5. Desarrollo del proyecto 25

5.1. Creación del entorno del trabajo 25

5.2. Diseño del entorno de simulación 27

5.2.1. Rviz 27

5.2.2. Gazebo 28

5.2.3. URDF (*Unified Robot Description Format*) 29

5.2.4. XACRO 36

5.3. Programación del entorno 40

5.3.1. Moveit! 40

5.3.2. Programa del entorno 48

6. Conclusiones 57

7. Bibliografía 59

8. Anexos 60

8.1. Anexo 1: Programa ur3.urdf 60

8.2. Anexo 2: Programa ur3.xacro 65

8.3. Anexo 3: Programa ur3moveit.xacro 70

8.4. Anexo 4: Programa move.py 98

8.5. Presupuesto del proyecto 115

Tabla de figuras

Figura 1: Esquema craneal, posición de la hipófisis ---- 6
Figura 2: Robot colaborativo FANUC ---- 7
Figura 3: Pictogramas de la ISO 10218-2 que indican el tipo de operación colaborativa ---- 10
Figura 4: Robot colaborativo UR3 ---- 11
Figura 5: Pantalla inicial Polyscope ---- 11
Figura 6:Pantalla de iniciación con el robot iniciado ---- 12
Figura 7: Pantalla de movimiento manual ---- 13
Figura 8: Pantalla de programación Polyscope ---- 14
Figura 9: Garra 2F-85 Robotiq ---- 15
Figura 10: Pantalla de control manual de la garra ---- 16
Figura 11: Versiones de ROS actuales ---- 19
Figura 12:Ejecución del comando rosdep update ---- 21
Figura 13: Ejemplo rqt_graph ---- 24
Figura 14: Creación de Paquete ---- 25
Figura 15: Carpeta Smros_wr ---- 25
Figura 16: Primer ejemplo de selección de paquete ---- 26
Figura 17: Confirmación del paquete que se esta utilizando ---- 26
Figura 18: Logo RVIZ ---- 27
Figura 19: Logo Gazebo ---- 28
Figura 20: Ejemplo de UR3 ---- 32
Figura 21: Programa ur3.urdf ---- 32
Figura 22: Resultado en Rviz del ur3.urdf ---- 33
Figura 23: Rviz "ur3.urdf" ejes en 0º ---- 34
Figura 24: Rviz "ur3.urdf" ejes desplazados ---- 34
Figura 25: Esquema programa ur3.urdf ---- 35
Figura 26: Programa ur3.xacro ---- 36
Figura 27: Rviz "ur3.xacro" ---- 37
Figura 28: Rviz "ur3.xacro" ejes desplazados ---- 37
Figura 29: Ejecución del programa ur3.xacro con TF visible ---- 38
Figura 30: Ejecución del programa ur3.xacro ---- 38
Figura 31: Visualización de las colisiones ---- 39
Figura 32: Programa ur3.xacro ---- 39
Figura 33: pantalla principal de "MoveIt" setup assistant ---- 40
Figura 34: Pantalla de generar matriz de colisión ---- 41
Figura 35: Ejemplo de pares de enlace que nunca pueden colisionar ---- 42
Figura 36: Agregar articulación virtual. ---- 43
Figura 37: Pantalla de grupos de planificación ---- 44
Figura 38:Pantalla para definir cadena cinemática ---- 44
Figura 39: Robot 1 posición en colisión ---- 45

Figura 40: Robot 2 en posición Home ---- 45
Figura 41: Pantalla "End Effectors" Robot 2 ---- 46
Figura 42: Pantalla generadora de archivos ---- 46
Figura 43: Pantalla generar URDF ---- 47
Figura 44: Ejecución en Rviz del archivo ur3moveit.xacro ---- 47
Figura 45: Comandos para ejecutar demo.launch ---- 48
Figura 46: Resultado en Rviz de ejecutar demo.launch ---- 49
Figura 47: rqt_graph del archivo demo.launch ---- 50
Figura 48: rqt_grahp programa move.py ene ejecución ---- 51
Figura 49: Función go_to_Home_Robot1 ---- 52
Figura 50: Función add_camara ---- 53
Figura 51: Ejecutando la función add_camara ---- 54
Figura 52: Función attach_camara ---- 54
Figura 53: Ejecutando función attach_camara ---- 55
Figura 54: Funciones detach_intru2 y remove_intru2 ---- 55
Figura 55: Programa en ejecución ---- 56

1. Introducción y objetivos

La finalidad de este proyecto es el diseño de un sistema robótico para dar un apoyo y ayuda, al personal médico en la operación de extracción de tumores cerebrales, a través del orifico nasal.

La operación consiste en la extracción de un tumor cerebral, que esta localizado en la hipófisis, una glándula situada "justo en el centro del cráneo".

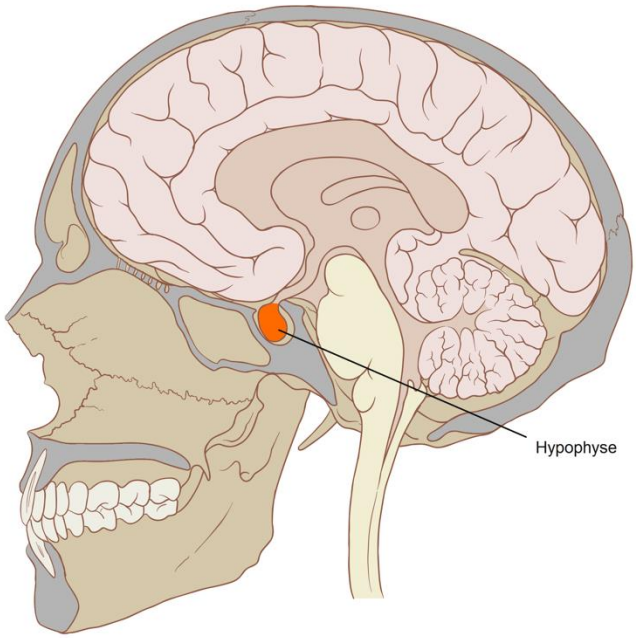


Figura 1: Esquema craneal, posición de la hipófisis

Hace unos años el método de extracción de estos tumores se hacia levantando el labio inferior en un proceso bastante aparatoso, ahora los médicos utilizan un neuronavegador que ayuda a los cirujanos a conocer con exactitud en qué punto de la anatomía del paciente se encuentran en cada momento. Con este tipo de cirugía, que se lleva a cabo con anestesia general, el enfermo sólo pasa entre tres y cinco días en el hospital, básicamente hasta que se le retira con éxito el taponamiento de las vías nasales establecido después de la operación.[1]

Con ayuda de los robots lo que se pretende es liberar al cirujano de la sujeción del neuronavegador y que ese trabajo le ejerza el robot, además de proporcionar otro robot que sea capaz de aproximar al cirujano las herramientas que se vayan necesitando para realizar la operación de forma correcta.

[1] https://www.vozpopuli.com/altavoz/next/sacar-tumor-cerebro-nariz_0_1065494746.html

2. 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, EE. UU. La empresa KUKA en una colaboración con el Centro Aeroespacial Alemán desarrolló el primer robot colaborativo de uso industrial, el LBR 3 en el año 2004.[2]

Figura 2: Robot colaborativo FANUC

Sin embargo, no fue hasta la década de 2010 que su diseño, producción y su uso empezó a extenderse. 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.

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. Esto sumado al hecho de que sus *footprints* (la superficie de su base de sujeción) suelen ser muy reducidas, 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 de seguridad es debido a que, por su diseño, o bien se mueven a velocidades reducidas, o bien estas

[2] https://www.aer-automation.com/mercados-emergentes/robotica-colaborativa/

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.

2.1. Normas y estándares para 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. Las dos primeras se tratan de normas que hablan sobre la robótica en general y la ultima se publicó de forma complementaria a estas dos, debido a que no cubrían las reglas de seguridad para robots colaborativos.[3]

Según la ISO 10218-1:2011 hay que tener en cuenta cuatro características para los robots colaborativos:

- Parada de seguridad monitorizada: Esta 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.

[3] ISO 10218-1:2011, ISO 10218-2:2011 e ISO/TS 15066:2016

- 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.

- 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 varié 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.

- 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 (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.

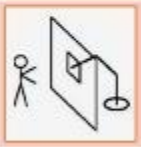
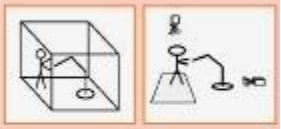
Type of collaborative operation	Pictogram (ISO 10218-2)
Safety-rated monitored stop	
Hand guiding	
Speed and separation monitoring	
Power and force limiting by inherent design or control	

Figura 3: Pictogramas de la ISO 10218-2 que indican el tipo de operación colaborativa

Cabe destacar, 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 este proyecto, 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.[4]

2.2. UR3

El UR3 de Universal Robots es un robot industrial colaborativo ultraligero y compacto más pequeño de la gama, ideal para tareas de montaje ligeras y bancos de trabajo automatizados. Este compacto robot de sobremesa solo pesa 11 kg, pero cuenta con una capacidad de carga de 3 kg, una rotación de 360° en todos sus ejes y una rotación infinita en el último eje.[5]

[4] Robots y dispositivos robóticos. Requisitos de seguridad para robots industriales. Parte 2: Sistemas robot e integración. (ISO 10218-2:2011).
[5] https://www.universal-robots.com/es/productos/robot-ur3/

Figura 4: Robot colaborativo UR3

2.2.1. Conceptos básicos del UR3

La forma de programación mas sencilla del UR3 es usando *Polyscope* la cual se trata de una interfaz grafica desarrollada por la propia empresa de fabricación del robot, Universal Robots.

Polyscope 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. *Polyscope* se ejecuta en la pantalla táctil que viene junto con el robot, está basado en el sistema operativo Debian Linux.

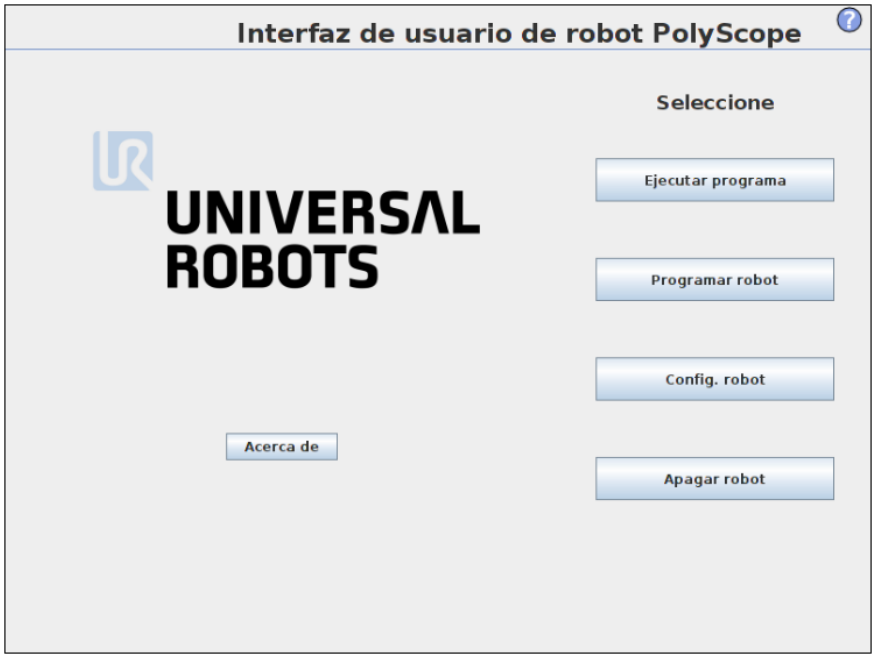


Figura 5: Pantalla inicial Polyscope

Lo primero y esencial es inicializar el robot y realizar la configuración de la red, si el robot no se inicializa o se deja la red sin configurar no es posible realizar ninguna conexión con el robot ni ningún movimiento.

La inicialización se debe realizar cada que se encienda el robot, con la pantalla de *Polyscope* en la mano dando al botón de "iniciar", esto se hace por seguridad, para que se pueda verificar la posición del robot real con la que se muestra en la pantalla de inicialización, realizando así una comprobación de que todo este correcto de manera visual.

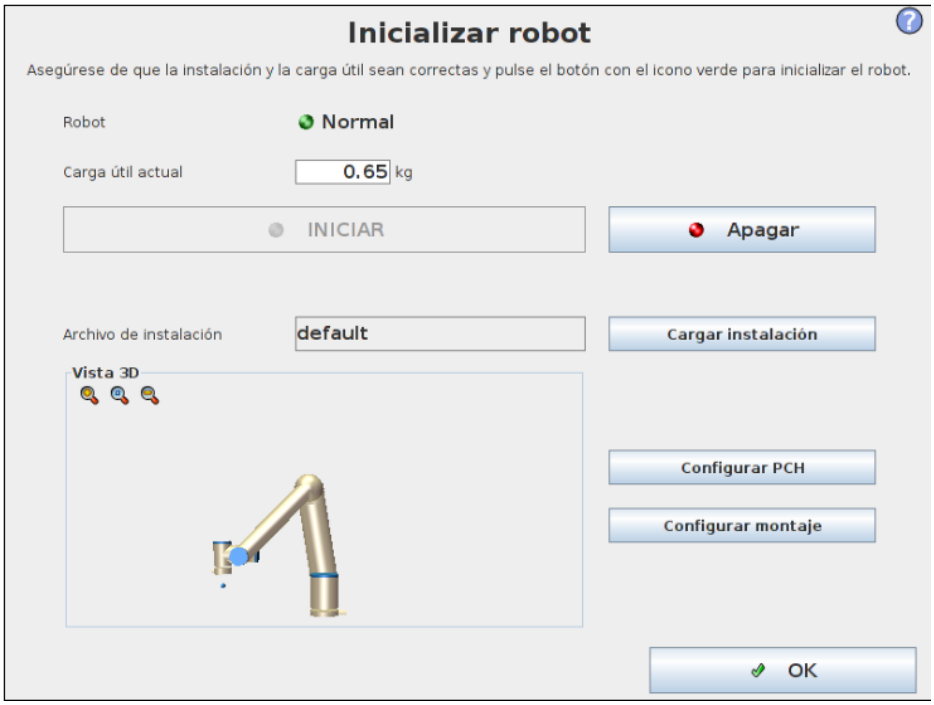


Figura 6:Pantalla de iniciación con el robot iniciado

Para el realizar movimientos de manera manual se puede realizar a través de dos métodos.

El mas sencillo es haciendo uso del movimiento libre del robot, manteniendo pulsado el botón que se encuentra detrás de la consola del robot, y que permite mover el robot agarrándolo físicamente y dirigiéndolo, realizando un suave movimiento de empuje de cada eje del robot.

O también 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.

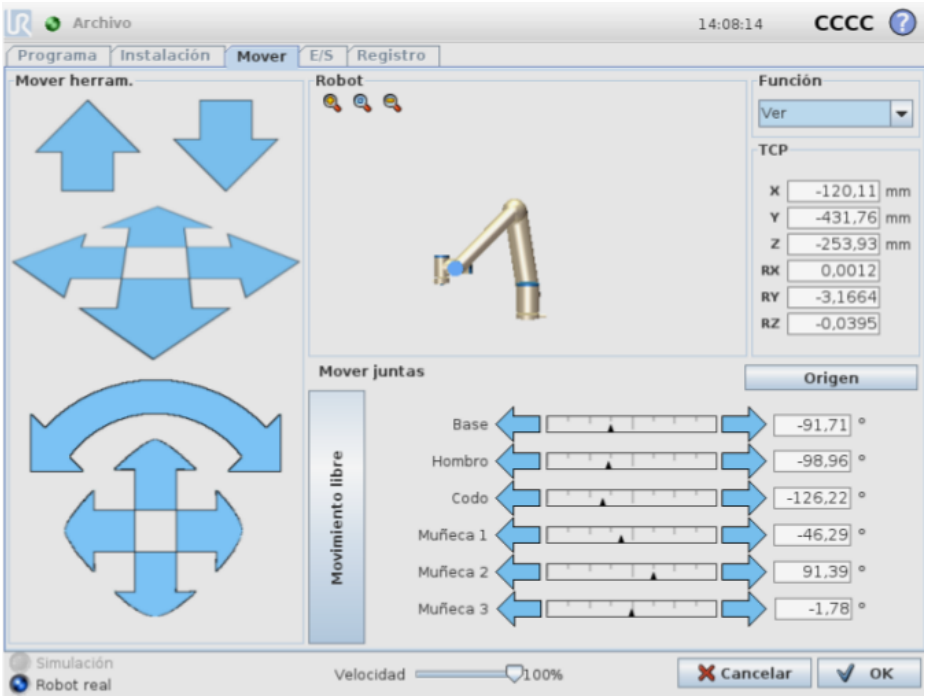


Figura 7: Pantalla de movimiento manual

Una vez en esta pantalla se puede mover el robot de forma manual usando como punto de origen de coordenadas el TCP de la herramienta, que se encuentra el centro del último eje, estos movimientos se realizan con las flechas del lado izquierdo de la pantalla.

También existe la opción de mover cada eje de manera independiente variando los grados con la parte del panel denominada “mover juntas”, situada en la parte inferior derecha.

De todos estos movimientos se tiene la capacidad de regular su velocidad a la que se ejecutan con la barra de velocidad situada en la parte inferior de la pantalla hay que tener en cuenta que un movimiento ejecutado de forma manual nunca alcanzara la velocidad máxima que puede alcanzar el robot, siempre se encuentra limitada.

Todos lo movimientos que realicemos se ven representados con el dibujo del robot que se encuentra en el centro de la pantalla y la posición con respecto al TCP se observa su cambio en la parte superior derecha de la pantalla.

2.2.2. Programación UR3

La programación del UR3 se pude hacer de diversas maneras la mas sencilla es realizando el programa en *Polyscope*, para lo cual se crea un programa nuevo y se accede a la ventana de estructura obteniendo una serie de opciones para realizar el programa deseado.

Se muestran varias pestañas “Básico”, “Avanzado” y “Asistente”. Con ayuda de todos estos comandos podemos realizar un programa bastante completo incluyendo buques de programación, diferentes tipos de movimientos e incluso la espera de señales por parte de otro sistema de programación que puede ser, por ejemplo, un autómata o señales conexionadas directamente a las cartas de entradas y salidas del robot, estas también pueden ser señales seguras, siempre que se cablee en doble canal y a la carta especial de seguridad del robot.

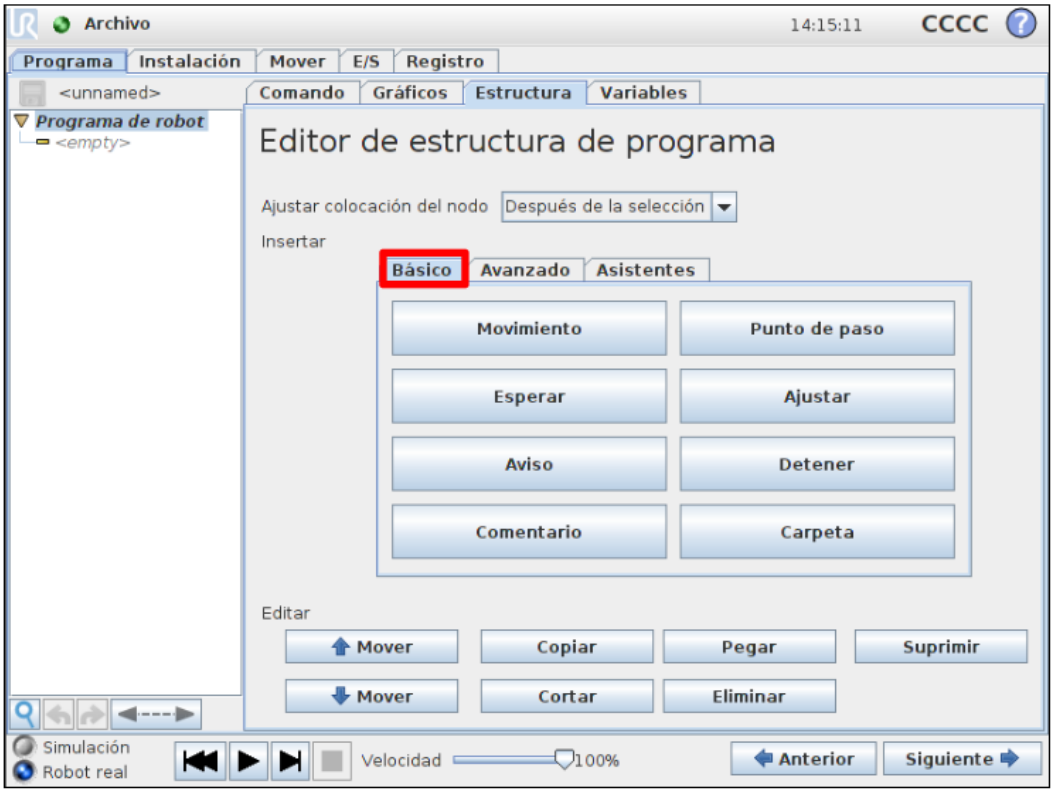


Figura 8: Pantalla de programación Polyscope

Otra manera de realizar la programación del UR3 son los *Scrips*, los cuales 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. Solo que utilizando sus propias formas de denominar a las unidades de definición.

Lo más interesante de esta programación es el uso de funciones que podemos crear y llamar previamente siendo posible utilizar estas funciones varias veces o incluso

en otros robots, lo que nos permite darles uso en nuestro sistema a funciones creadas por otros usuarios.
Lo que provoca que esta programación sea mucho mas flexible que la anterior, añadiéndole también una mayor dificultad.

Los métodos de programación hasta ahora expuestos son los creados por la propia empresa de *Universal Robots* para el uso de sus robots. Pero lo que utilizaremos para este proyecto será *ROS* un sistema de programación compatible con mas robots, lo que nos proporciona aun más flexibilidad a la hora de realizar nuestro sistema, pero añade mayor grado de dificultad.

3. Garra Robotiq

La manipulación de los diferentes objetos del sistema se realizará mediante el uso de la garra Robotiq 2F-85, se trata de una garra colaborativa formada por dos dedos, capaz de controlar la presión que ejerce para evitar daños a personas o a objetos de su entorno, a través del control que ejerce sobre los aumentos de corriente que aumentan de forma proporcional a la presión que se ejerce.[6]

Figura 9: Garra 2F-85 Robotiq

Esta función no solo permite que la garra sea segura para trabajar cerca de personas sin ninguna barrera de seguridad, si no que proporciona mucha flexibilidad a la pinza

[6] https://robotiq.com/es/productos/pinza-adaptable-2f85-140

ya que es capaz de agarra objetos de diferente tamaño sin dañarlos, siempre dentro de su rango de separación de 85mm y teniendo en cuenta que la posición final de separación de los dedos se la damos mediante los valores del programa.

También cabe la posibilidad de regular la velocidad de cierre y apertura de la garra, dándole a cada uno un valor diferente o igual, siempre a de tenerse en cuenta que a mayor velocidad la garra llevara mayores inercias provocando mas riesgo de ejercer mayor fuerza sobre un objeto inesperado a la hora de realizar el agarre.

La garra puede ejercer una fuerza desde los 20N hasta los 235N, esta presión también es programable lo que nos permite que sea variable y hacerla dependiente de objetos o elementos externos al robot.

Figura 10: Pantalla de control manual de la garra

La instalación de la garra se hace directamente al robot mediante un USB donde se encuentra el controlador de la garra, lo que hace que sea un complemento muy sencillo de dirigir mediante el propio sistema de programación del UR3, denominado URCaps. *URCaps* es un sistema de *software* propio de Universal Robot para la programación y el uso de periféricos.

También dispone de una conexión USB que permite controlar la garra desde otro sistema operativo como es en el caso de este proyecto. Al realizar la conexión de la garra a otro sistema provoca que el movimiento y el control por el sistema del UR3 quede inhabilitado, cuando se realice la desconexión del otro sistema es importante devolver el poder del control al sistema *URCaps.*

4. ROS (*Robot Operating System*)

4.1. Definición

ROS es un sistema operativo robótico basado en un marco de trabajo de código abierto y gratuito, utilizado para la programación de robot, basado en drivers y librerías que permiten la comunicación con una amplia variedad de sistemas robóticos.

Sin embargo, no se considera un sistema operativo como tal, ya que para su uso es necesario del apoyo de un sistema operativo completo que gestione los recursos del *hardware,* organice y prioricé los servicios a las aplicaciones del *software*.

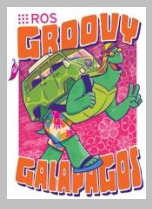
Aunque si tiene la capacidad de control de dispositivos de bajo nivel, comunicación entre procesos, implementación de funciones de uso común, administración de paquetes, llamadas de funciones y sistema de configuraciones para el control y programación de robot.[7]

4.2. Versiones de ROS

Actualmente existen numerosas versiones de ROS, ya que al ser código abierto sufre modificaciones y mejoras constantes, como podemos observar en la siguiente tabla, algunas se encuentran en proceso de lanzamiento (“amarillas”), disponibles para su uso (“verdes”) y otras ya descatalogadas según las reglas de lanzamiento de la comunidad de ROS (“grises”).

Versión	Fecha de lanzamiento	Poster	*Tortuga*, el en tutorial	Fecha de eliminación
ROS Noetic Ninjemys	Mayo, 2020 (previsto)	TBA	TBA	Mayo, 2025 (previsto)
ROS Melodic Morenia	23 de mayo, 2018			Mayo, 2023

[7] http://wiki.ros.org/

ROS Lunar Loggerhead	23 de mayo, 2017			Mayo, 2019
ROS Kinetic Kame	23 de mayo, 2016			Abril, 2021
ROS Jade Turtle	23 de mayo, 2015			Mayo, 2017
ROS Indigo Igloo	22 de julio, 2014			Abril, 2019
ROS Hydro Medusa	4 de septiembre, 2013			Mayo, 2015
ROS Groovy Galapagos	31 de diciembre, 2012			Julio, 2014
ROS Fuerte Turtle	23 de abril, 2012			--
ROS Electric Emys	30 de agosto, 2011			--
ROS Diamondback	2 de marzo, 2011			--

ROS C Turtle	2 de agosto, 2010			--
ROS Box Turtle	2 de marzo 2010			--

Figura 11: Versiones de ROS actuales

La creación de versiones de ROS tiene el propósito de que los desarrolladores tengan una base relativamente estable, y con cada versión ir corrigiendo errores y añadiendo mejoras, por esta razón se deja los paquetes de nivel superior con reglas menos estrictas para su posible mantenimiento y mejora del sistema.

A mayores, la comunidad de ROS tiene unas reglas creadas para los lanzamientos, el tiempo de soporte y las actualizaciones de las nuevas versiones:

- El tiempo de lanzamiento de las versiones de ROS se basa en la necesidad y en los recursos disponibles, se intenta lanzar una nueva versión cada año.
- Los lanzamientos que sean numero par serán una versión LTS (*Long Term Support*), lo que implica que tendrá soporte y será actualizada durante mas tiempo, respaldada por cinco años.
- Las versiones de números impares son versiones normales de ROS, compatibles, durante dos años.
- Cada versión de ROS es compatible solo con una versión de Ubuntu LTS

La versión del sistema operativo a utilizar depende de la versión de ROS con la que se quiera trabajar.

Los sistemas operativos que soportan la instalación de ROS de forma mas fiable son: Ubuntu, Debian y Windows 10, esta última solo para las versiones más actuales de ROS.
A mayores existen versiones de sistemas operativos que están en experimentación o en proceso de construcción como Arch Linux, Gentoo, OS X o OpenEmbedded.

Para este proyecto se usará la versión ROS *Melodic Morenia* y el sistema operativo de Ubuntu en la versión 18.04, que es la versión compatible con dicha versión de ROS.

4.3. Instalación de ROS

Para la instalación de ROS hay que seguir los siguientes pasos:

1. Ejecutar la aplicación del terminar y configurar el sistema operativo para la aceptación del *software* de los paquetes propios de la instalación

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

2. Configuramos el acceso con la llave

```
sudo apt-key adv --keyserver 'hkp: //keyserver.ubuntu.com: 80' --recv-key C1CF6E31E 6BADE8868B172B4F42ED6FBAB17C654
```

3. Se actualiza el sistema de paquetes disponible mas sus versiones e instalamos la versión completa

```
sudo apt update

sudo apt install ros-melodic-desktop-full
```

4. Se comprueba que la instalación de ROS *melodic* se ha realizado correctamente realizando una búsqueda de los paquetes disponibles y configurando el entorno agregando las variables ROS cada vez que se inicie un terminal.

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

5. Por ultimo, se instalan las dependencias para poder construir espacios propios de trabajo ejecutando

```
sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python
-wstool build-essential
```

Al inicializar el comando rosdep, se debería obtener un resultado similar al de la imagen inferior:

```
maria@maria-VirtualBox:~$ rosdep update
reading in sources list data from /etc/ros/rosdep/sources.list.d
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/osx-homebrew.y
aml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/base.yaml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/python.yaml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/rosdep/ruby.yaml
Hit https://raw.githubusercontent.com/ros/rosdistro/master/releases/fuerte.yaml
Query rosdistro index https://raw.githubusercontent.com/ros/rosdistro/master/ind
ex-v4.yaml
Add distro "ardent"
Add distro "bouncy"
Add distro "crystal"
Add distro "dashing"
Add distro "eloquent"
Add distro "foxy"
Add distro "groovy"
Add distro "hydro"
Add distro "indigo"
Add distro "jade"
Add distro "kinetic"
Add distro "lunar"
Add distro "melodic"
Add distro "noetic"
updated cache in /home/maria/.ros/rosdep/sources.cache
```

Figura 12:Ejecución del comando rosdep update

Cuando se instalan dependencias nuevas puede causar algún error dentro del sistema para ser ejecutarlo, suele ser por la falta de algún paquete que utilice la dependencia y nuestro sistema carezca de ello.

Para solucionar esto se puede usar el siguiente comando sustituyendo los puntos suspensivos por la dependencia que nos indica el error.

```
sudo apt install ……..
```

A mayores cada vez que se instale algún paquete o dependencia es recomendable actualizar el sistema de paquetes disponible y sus versiones, como se realizó en el punto 3 de este apartado.[8]

4.4. Conceptos básicos de ROS

- Paquetes:

El *software* de *ROS* se organiza en paquetes, donde cada paquete contiene una combinación de código, datos y documentación.

Estos paquetes se encuentran dentro de los espacios de trabajo, en el directorio *src.* Cada directorio de un paquete debe incluir un *CMakeList.txt* que es el fichero de compilación y el fichero *package.xml* que describe el contenido del paquete y como el sistema interactúa con él, estos dos archivos son los que diferencian una carpeta normal de un paquete de ROS. Los comandos para crear un paquete dentro del entorno de trabajo *catkin_ws:*

```
cd ~/catkin_ws/src

catkin_create_pkg <Nombre del paquete> rospy
```

Estos comandos crean un directorio con el nombre que le otorguemos al paquete e incluyen todos los documentos que hemos mencionado anteriormente de forma automática.

No es necesario que el paquete sea creado dentro de la carpeta con el nombre *catkin_ws,* solo que es el mas usado dentro de la comunidad de ROS ya que es el ejemplo que se utiliza en los tutoriales, pero el nombre puede ser cualquiera, y ubicar el paquete donde se quiera dentro de nuestro sistema operativo.

- Nodos:

Dentro de los paquetes de ROS se crean archivos ejecutables llamados nodos.

Los nodos son unidades de *software* de procesamiento de datos e información donde se implementan todos los requisitos funcionales para una aplicación de *ROS*. Estas unidades de *software* utilizan las bibliotecas de ROS para comunicarse con otros

[8] http://docs.ros.org/melodic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html

nodos se encargan de publicar o suscribirse a un *topic*, además de, proporcionar o usar algún servicio.
La funcionalidad de la creación de nodos es garantizar la modularidad y flexibilidad en los proyectos que se creen con *ROS*.

Existen comandos de *ROS* que permiten ejecutar directamente un nodo sabiendo dentro de que paquete se encuentra (*rosrun*) o ver que nodos están ejecutándose activamente en un terminal en el momento de la ejecución del comando (*rosnode list*) o incluso proporcionar información del propio nodo (rosnode info <nombre_del_nodo>)

- *Topics*:

Para comunicarse un nodo entre otro se usan las entidades denominadas *topics*, el funcionamiento es sencillo un nodo es el encargado de publicar un *topic*, mientras que otro nodo se suscribe a dicho *topic* de esta manera esos dos nodos están comunicados y pueden interaccionar entre ellos mandándose mensajes o servicios a través de esta línea de comunicación, es importante que los mensajes o servicios que envié un nodo y que espere recibir el otro nodo sean del mismo tipo y que presenten la misma estructura, si no es así ocasionaríamos un error de comunicación entre los nodos.

Hay que saber distinguir entre un mensaje, un servicio y una acción, aunque la vía de comunicación de todos se a la misma.
Los mensajes se programan para que se envíen de forma constante, con un tiempo de separación entre uno y otro. Además, la perdida de alguno de estos mensajes debe de tener poca relevancia, ya que la comunicación es repetida en un corto espacio de tiempo.
En cambio los servicios tienen mayor relevancia y necesitan de una respuesta de confirmación de que la comunicación se ha realizado con existo, ya que se envían solo cuando se produce algún evento determinado, esto sirve para no sobrecargar el sistema y evitar enviar datos constantes cuando no son realmente necesarios, es importante tener en cuenta que los servicios ocasionan que el programa sea mas secuencial, porque al solicitar el uso de un servicio es necesario recibir la respuesta de confirmación para que el programa continué.

Para evitar que un proceso se bloquee al no recibir la respuesta de confirmación de un servicio o simplemente poder ir realizando más objetivos mientras se espera a que otro se cumpla, existen las acciones que están formadas por una solicitud, una respuesta y unas variables de realimentación que nos permiten ejecutar mas funciones del programa sin bloquearse o sin esperar a la respuesta, pero asegurándonos de que no perdemos información, esto permite dar mayor dinamismo a los programas y a los procesos.

Uno de los comandos que muestra un gráfico para observar de forma fácil e intuitiva los nodos y los *topics*, además de los servicios, mensajes y acciones que se están ejecutando en ese momento y la forma en la que interactúan unos con otros, es *rqt_graph,* es muy útil para observar si tenemos algún problema de comunicación entre los nodos.

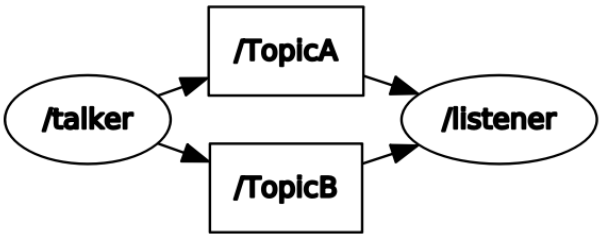


Figura 13: Ejemplo rqt_graph

En la figura superior se puede ver un ejemplo de la aplicación del comando *rqt_graph*, donde los nodos se representan mediante elipses y los *topics* dentro de rectángulos, estas forman graficas pueden ser variables dependiendo de la versión de ROS con la que se trabaje, ya que los *topics*, pueden venir representados sobre la flecha que une los diferentes nodos precedido de /.
A veces, pueden existir nodos que no se comuniquen con ningún otro nodo, pero esto no suele ser muy común, ni muy buena señal en la realización de un proceso.

Otro comando de ROS que nos puede facilitar el trabajo con los nodos y los *topics* es *rosnode <nombre del nodo>,* este comando imprime información específica del

nodo, mostrando los *topics* y los servicios que tiene definidos y especificando cuales están en uso.

5. Desarrollo del proyecto

5.1. Creación del entorno del trabajo

Para todas las operaciones que se realicen en nuestro sistema ROS es necesario la configuración del entorno, Para ello se utiliza el siguiente comando, el cual, es necesario e imprescindible cada vez que se quiera trabajar con ROS y es importante recordar ejecutarlo siempre que se abra un terminal nuevo.

```
$ source /opt/ros/melodic/setup.bash
```

Para iniciar el proyecto lo primero es generar un entorno de trabajo propio en ROS. Para ello, nos se creará un entorno de trabajo propio de ROS, con el nombre que se quierá, por ejemplo: "*Smros_wr*"
Con los comandos que se han mencionado antes y sustituyendo por nuestro propio nombre de entrono, ser realiza la creación y complicación del nuevo entorno creado.

```
maria@maria-VirtualBox:~$ mkdir -p ~/Smros_wr/src
maria@maria-VirtualBox:~$ cd Smros_wr/
maria@maria-VirtualBox:~/Smros_wr$ catkin_make
```

Figura 14: Creación de Paquete

Como resultado de la compilación del entorno, se debería de obtener dentro de la ubicación que se ha especificado anteriormente una carpeta con el nombre "*Smros_wr*", y en su directorio tres carpetas nuevas llamadas "*build*", "*devel*" y "*src*". Dentro de la carpeta "*devel*" debería de existir varios archivos con la terminación *.*Sh* confirmando que la compilación se ha realizado correctamente.

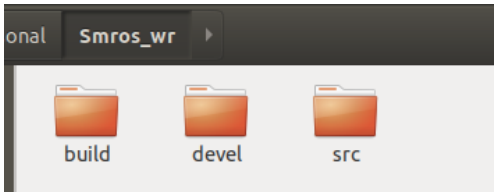


Figura 15: Carpeta Smros_wr

Hay que tener en cuenta que para la utilización de este entorno y con el sistema de ROS hay que asegurarse que el entorno de trabajo que se quiera usar esta superpuesto a los demás para ello cada vez que se inicie un terminal hay que realizar una de estas dos opciones:

1. Abrir un terminal, ubicarse dentro del entorno que se vaya a utilizar y ejecutar el comando:

```
source devel/setup.bash
```

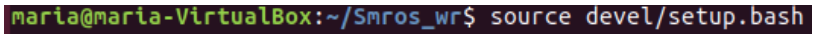


Figura 16: Primer ejemplo de selección de paquete

2. O directamente cada vez que se ejecute la apertura de un terminal se ejecutara el siguiente comando:

```
. ~/Smros_wr/devel/setup.bash $ROS_PACKAGE_PATH
```

Si se quiere asegurar que el espacio de trabajo con el que se quiere trabajar es el correcto se puede comprobar ejecutando el siguiente comando y la confirmación será el retorno del terminal indicándonos la ubicación, de la carpeta “src”, del entorno que se pretende trabajar.

```
echo $ROS_PACKAGE_PATH
```

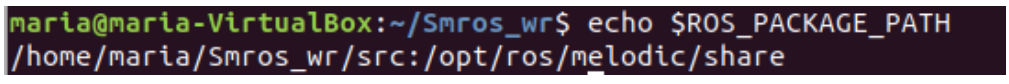


Figura 17: Confirmación del paquete que se esta utilizando

Los archivos generados por el comando de compilación hay que tener en cuenta que tiene un autor y sus datos. Por eso es recomendable, personalizar estos archivos.

5.2. Diseño del entorno de simulación

Para diseñar el entorno de simulación donde van a trabajar los robots y realizar la simulación de la programación se necesita diseñarlo mediante *URDF*, además, para suplementar a *URDF* y eliminar sus limitaciones utilizaremos los macros *Xacro*.
Para la visualización de nuestro entorno de simulación y las pruebas de programa, se usará las aplicaciones *Rviz* y *Gazebo*

5.2.1. Rviz

Rviz es una herramienta de visualización 2D y 3D para aplicaciones de ROS, proporciona una vista del modelo de robot, captura la información de los sensores en el sistema interno del robot y reproduce los datos capturados. Puede mostrar datos de cámara, láseres y dispositivos 3D y 2D, como imágenes y nubes de puntos.[9]

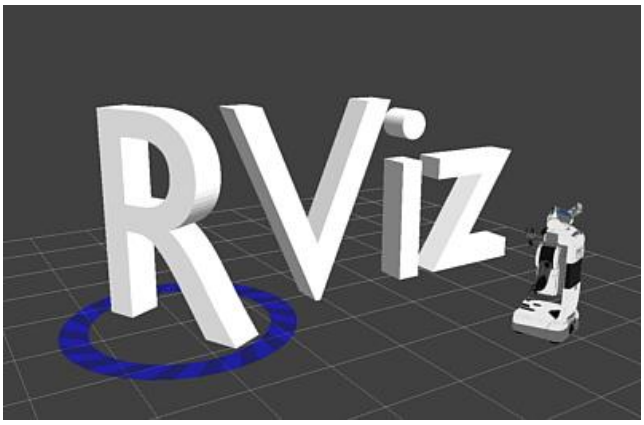


Figura 18: Logo RVIZ

Rviz se instala al realizar la instalación del paquete completo de ROS, para ejecutarlo es necesario crear un archivo propio de la aplicación de Rviz, especificando las condiciones del entorno que se quiere recrear, un sistema de referencia principal y algunas características propias del programa como puede ser la trasparencia de los objetos a representar.
La ventaja de este programa es que, una vez ejecutado el entorno, con las condiciones del archivo inicial que se ha creado, se pueden hacer modificaciones sobre muchas de estas características y realizar un nuevo archivo de Rviz para una

[9] https://docs.aws.amazon.com/es_es/robomaker/latest/dg/simulation-tools-rviz.html

ejecución posterior, evitando así que cada vez que se lance el archivo se tengan que modificar de nuevo las características.

5.2.2. Gazebo

Gazebo es un simulador de entornos 3D que posibilita evaluar el comportamiento de varios robots en un mundo virtual. Permite, entre muchas otras opciones, diseñar robots de forma personalizada, crear mundos virtuales usando sencillas herramientas CAD e importar modelos ya creados.[10]

Además, es posible sincronizarlo con ROS de forma que los robots emulados publiquen la información de sus sensores en nodos, así como implementar una lógica y un control que dé ordenes al robot.

Para ejecutar un entorno determinado es recomendable crear un archivo *.world donde se especifique las características del sistema de simulación, ya que Gazebo permite crear un entorno con parámetros modificables como la altitud, la gravedad, incluso la temperatura y un montón de condiciones físicas que en un entorno real pueden ser variables.

Otra ventaja es poder añadir condiciones de luz que pueden ir cambiando a lo largo del día según pasan las horas o añadir focos de luz que en el mundo real serian proporcionados por luces artificiales.

Figura 19: Logo Gazebo

[10] http://gazebosim.org/tutorials?tut=ros_overview

Esta es una de las principales diferencias con Rviz, mientras Gazebo se centra en una reproducción mas real del entorno añadiendo condiciones físicas, Rviz permite visualizar y modificar las condiciones que se añadan a los sensores, que pueden ser externos al robot o integrados en él. Lo mas favorables es que los dos entornos de simulación se pueden ejecutar a la vez, es decir, no estas obligado a ejecutar tu entorno en un único simulador, si no que se puede ejecutar el programa en los dos simuladores a la vez siendo uno complementario del otro. Lo que permite tener unas condiciones físicas determinadas y ver la eficacia de los diversos sensores.

5.2.3. URDF (*Unified Robot Description Format*)

URDF es utilizado en ROS para el modelaje de robot usando el lenguaje de programación XML (*Extensible Markup Language*), que de forma resumida es un lenguaje que permite definir etiquetas personalizadas para la descripción y organización de datos, en este caso se utiliza para definir los elementos que van a componer el entorno de simulación, además de su posición y geometría. Y la relación que tienen unos elementos con otros.

Hay que tener en cuenta alguno de los elementos esenciales y la estructura que tiene un documento XML, ya que compartirá la misma filosofía de programación que el programa en URDF, que se utiliza en el proyecto. Estos son:

- Prólogo

Se utiliza para dar una introducción sobre el programa que se va a realizar, e incluye información relevante como la versión que se va a utilizar, el tipo de codificación que se va a usar. Esta parte del código es opcional, pero puede ser muy relevante si queremos que nuestro programa sea utilizado por la comunidad de ROS.

- Cuerpo

Es la parte donde se escribe lo mas relevante de archivo de compilación, tiene una estructura de árbol, en la que debe de existir un elemento principal, dentro del cual se encuentran todos los demás elementos, estos elementos a su vez pueden estar formados por subelementos, a la hora de programar es importante tener en cuenta

que cada elemento o subelemento, tiene que ejecutar la apertura y el cierre no se puede dejar ningún elemento abierto.

<Elemento> (…) </ Elemento>

- Atributos

Es la manera de incorporarles características o propiedades a los elementos, deben ir entre comillas.

- Entidades predefinidas

Sirve para especificar caracteres especiales que no se pueden interpretar con el procesador XML. Se escriben entre el carácter &.

- Comentarios

Son aclaraciones escritas para la interpretación del programa, puede darles uso desde el propio autor o un lector que quiera dar uso al programa. Los comentarios son ignorados a la hora de realizar la compilación del programa.

<!--- Comentario del autor --->

Usando esta estructura y URDF lo que se pretende es describir la dinámica y la cinemática del robot, realizar una representación visual y establecer unas reglas de colisiones con el robot y su entorno. Se trata de una estructura en forma de árbol donde existen unos elementos principales en los cuales se va desarrollando los demás elementos y atributos.

La descripción de un robot mediante el uso de URDF consiste en crear un conjunto de cuerpos rígidos denominados *links,* cada *link* representa una estructura con unas características determinadas, que se le asignan en su descripción a la hora de realizar el programa.
Las características que se le establecen pueden ser básicas y necesarias como la geometría que tiene el *link*, hasta la inercia o el material con que este hecho.

La otra parte fundamental en la descripción del robot es las uniones que tienen los diferentes *links* entre ellos y el entorno, este elemento de llama *joint.* Los *joints* son las articulaciones del robot, en ellos se describen las propiedades dinámicas y cinemáticas, además de los limites de seguridad de la articulación, sus limitaciones y las características del movimiento de los *links*.

A la hora de definir la articulación hay que definir dos elementos obligatorios uno que se denomina como "padre" y el otro el "hijo" de la articulación ya que como se ha mencionado anteriormente la programación en URDF se tiene que hacer en estructura de árbol. Por lo que tenemos que definir nuestro elemento "padre" como principal e "hijo" como secundario, esto implica que el elemento "hijo" se mueve en relación con el "padre".
Además, hay que asignarles dos atributos obligatoriamente a la articulación, uno es el nombre y el otro el tipo de articulación, es decir, como se mueve el elemento con relación al otro, esta unión puede ser fija, es decir que el elemento hijo se quede de forma solida al elemento padre.

Para comenzar se desarrolla un pequeño programa para diseñar una imitación del UR3 que se va a utilizar y a lo largo del proyecto se irá mejorando.
Es muy útil, tener en cuenta a la hora de hacer programas en URDF que disponemos de un comando que nos permite chequear nuestro programa por si tenemos algún error.

```
check_urdf <nombre_del_programa.urdf>
```

Siempre teniendo en cuenta que lo que salta como error con este comando, son fallo en la estructura del propio programa no fallos ni de inercias, ni de materiales, ni peso de los elementos que se dispongan para nuestro robot.
A mayores se puede usar el siguiente comando que muestra una representación del programa, disponiendo visualmente en forma de gracfet los enlaces programados.

```
urdf_to_graphiz <nombre_del_programa.urdf>
```

Usaremos una imagen del UR3 de referencia.

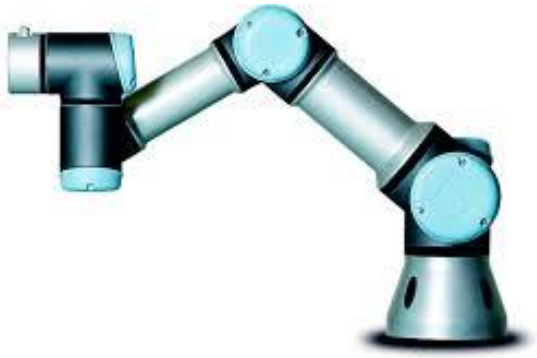


Figura 20: Ejemplo de UR3

Lo ideal es empezar por partes, desde abajo arriba, se empieza por la base.

Como se ha explicado antes, ponemos un prólogo donde especificamos la versión y se le da un nombre al robot y al nombre del proyecto en este caso “ur3”.

Y se crean dos elementos uno de ellos es el elemento mundo donde va a ir sujeto el robot y que no tienen ningún atributo mas.

Y el otro es la base del robot, que se representa mediante un cilindro, a mayores se establecerá que el elemento está compuesto por un metal y le daremos un color en formato RGBA, teniendo en cuenta que 255 es igual al valor 1 y donde el ultimo número representa la opacidad del material siendo uno el valor máximo.

Se situará el *link* en un punto del espacio en este caso en el origen y elevado 0.025 de la superficie. Este valor de origen es en referencia al sistema de coordenadas de todo el entorno, a mayores se puede añadir un origen en la definición de la articulación donde el sistema de referencia se encuentre en la articulación principal.

ur3.urdf

```
<?xml version="1.0"?>
<robot name="ur3">
  <link name="world"/>
  <link name="base">
    <visual>
      <geometry>
        <cylinder length="0.20" radius="0.15"/>
      </geometry>
      <material name="metal">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </visual>
  </link>
  <joint name="union_base" type="fixed">
    <parent link="world"/>
    <child link="base"/>
  </joint>
</robot>
```

Figura 21: Programa ur3.urdf

Y por último se definirá la unión entre estos dos elementos, en este caso se trata de una unión fija donde el elemento principal es el mundo que ejerce de "padre" y un "hijo" que es el cilindro que se ha creado.
Esta pequeña parte del programa se puede ejecutar y ver el resultado Rviz.

El resultado de este programa es un pequeño cilindro en 3D, con la unión que le hemos establecido. Es decir, el valor del *Fixed Frame* tiene que ser el elemento mundo, si el sistema no será valido.

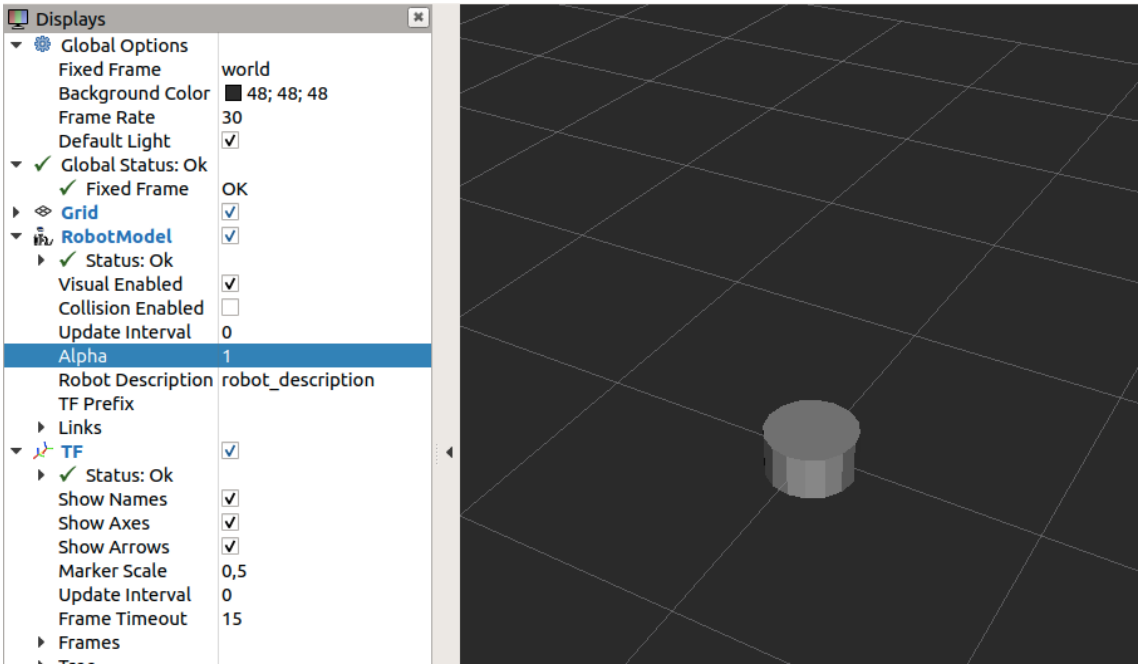


Figura 22: Resultado en Rviz del ur3.urdf

Con esta filosofía se van generando todas uniones para realizar la misma geometría que muestra el robot UR3 real, la única diferencia es que ahora las uniones en vez de fijas, como la unión anterior del mundo con la base del robot, serán móviles, y se moverán en relación con el elemento que establezcamos, en este caso al eje que este contiguo al elemento y sea el más próximo a la base.

Se realiza el programa "*ur3.urdf*" completo (anexo 1) y se ejecuta obteniendo un diseño aproximado del robot que se va a utilizar. En el entorno de Rviz se puede ejecutar los movimientos de los seis ejes que se han programado, esto permite observar puntos de colisión entre las diferentes partes del robot y la realizacion de movimiento de forma manual en un entrono muy básico.

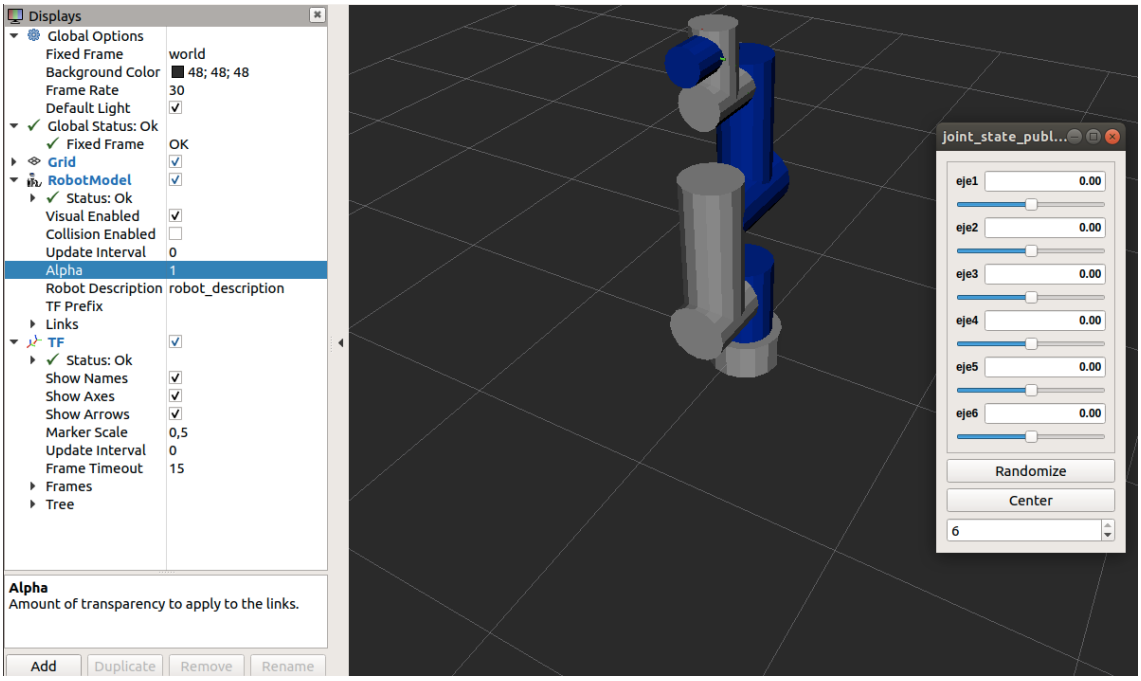


Figura 23: Rviz "ur3.urdf" ejes en 0º

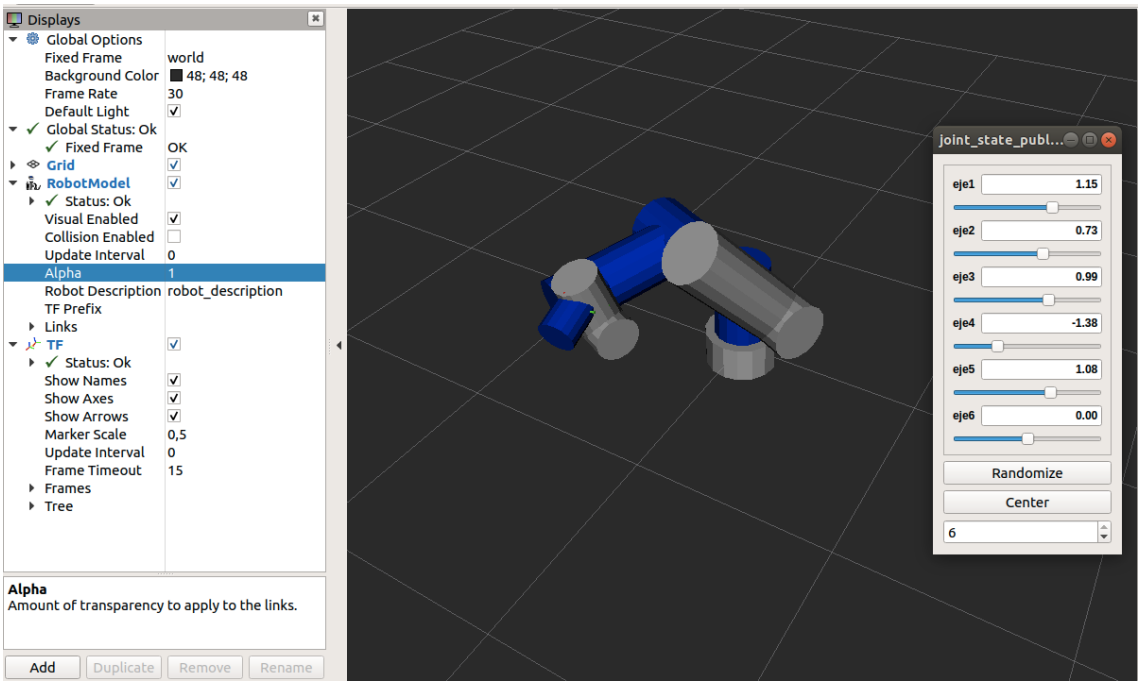


Figura 24: Rviz "ur3.urdf" ejes desplazados

Además, se puede visualizar en un esquema en forma de árbol con todas las uniones y elementos que origina el programa “ur3.urdf” que se ha creado. Como se puede observa se ve la relación que tienen los seis ejes entre si, la posición donde se encuentra con respecto al eje anterior y las uniones de los brazos que se hacen fijas con respecto al eje que las sujeta.

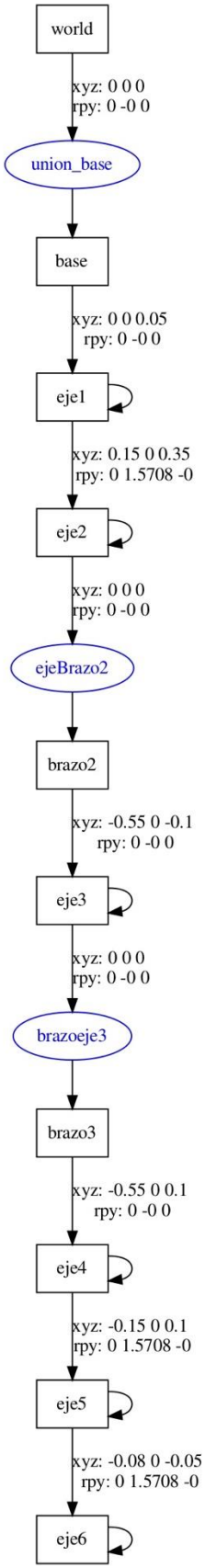


Figura 25: Esquema programa ur3.urdf

5.2.4. XACRO

Como se puede observar si se realiza el programa solo usando URDF se obtienen varias limitaciones, como no poder simular robot de una forma cerrada, la falta de etiquetas de definición de sensores o la ampliación o modificación de un robot cuando ya se han definido, aunque esto ultimo tiene bastante sentido ya que si alargamos un brazo de un robot o añadimos otro elemento puede resultar peligroso ya que los movimientos y cálculos realizados cambiarían y los movimientos que se habían programado anteriormente pueden provocar colisiones. A mayores queda una simulación con elementos poco reales.

Todas estas limitaciones se solucionan utilizando un sistema propio de ROS formado por marcos denominado XACRO que ejecuta todas las funciones consecuentes en el programa para simplificar y mejorar el código en URDF.
La parte negativa del uso del las macros XACRO es que para usar los elementos de ayuda que utilizamos en URDF como el comando “ckeck_urdf” no se puede aplicar directamente y si es necesario usarlos tenemos que pasar nuestro programa “*.xacro” a “*.urdf” usando un elemento conversor.

Para crear el programa con XACRO se usará la misma filosofía de programación que en el programa anterior.

ur3.xacro

```
<?xml version="1.0" ?>
<robot name="ur3" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- world -->
  <link name="world" />

  <!-- Robot1 -->
  <xacro:include filename="$(find ur3_description)/urdf/ur3.urdf.xacro"/>
  <xacro:ur3_robot prefix="robot1_" joint_limited="true"/>

  <!-- Joints -->
  <joint name="world_interface_to_world" type="fixed">
    <parent link="world" />
    <child link="robot1_base_link" />
  </joint>
</robot>
```

Figura 26: Programa ur3.xacro

En esta primera parte del programa se realizará la simulación con lo mismo que se ha creado cuando se realizó en el programa “ur3.urdf”, solo que ahora se utilizara las herramientas que proporciona los macros de XACRO. Pero se mantiene la misma

estructura de programa. Un elemento “padre” que este caso es el mundo que ejercerá de marco fijo y un elemento “hijo” que es el robot UR3, unidos por una articulación fija en el punto origen del marco fijo. Se pueden ejecutar movimientos manuales en el entorno de simulación desplazando cada uno de los ejes a una posición concreta.

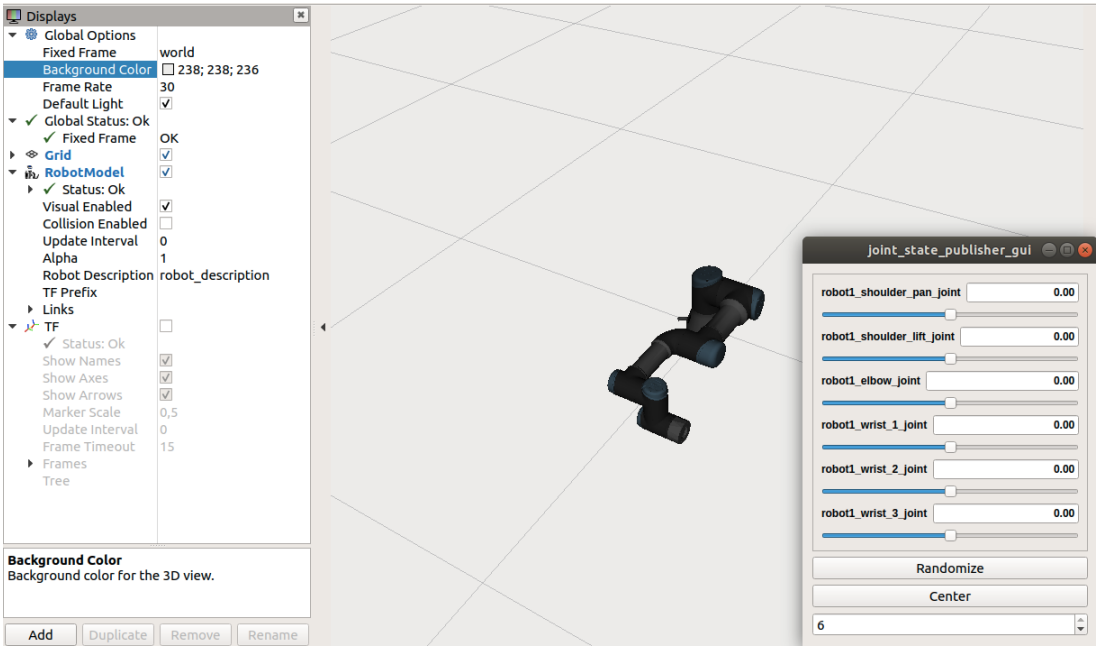


Figura 27: Rviz "ur3.xacro"

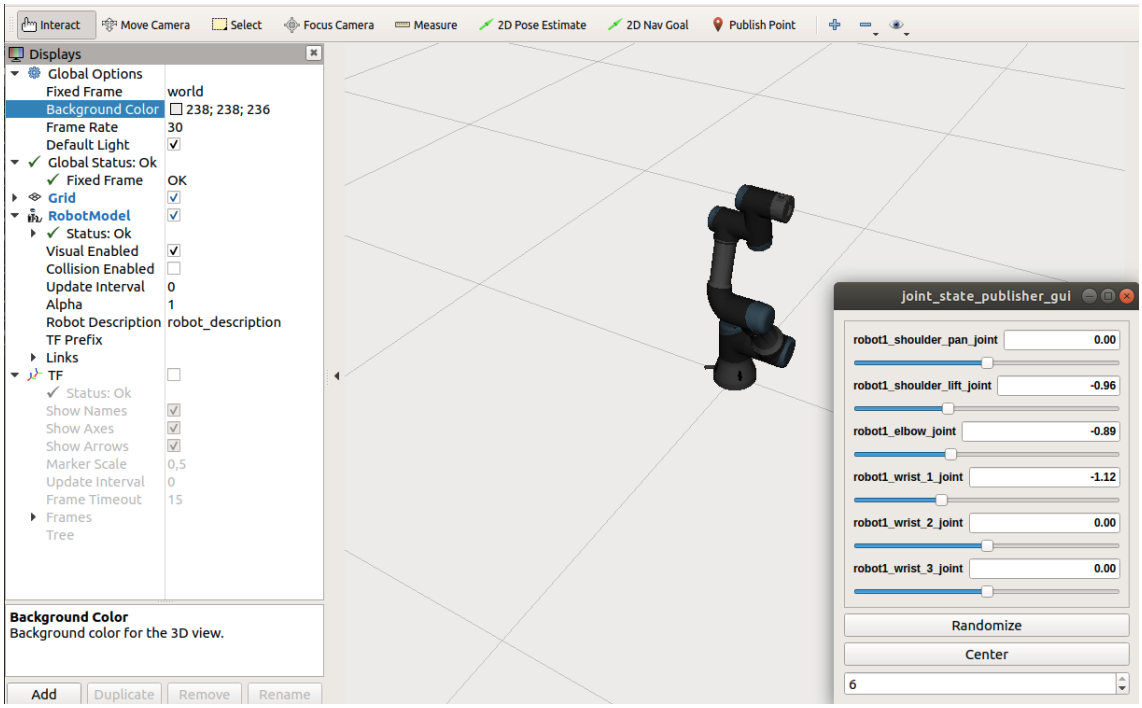


Figura 28: Rviz "ur3.xacro" ejes desplazados

A partir de esta estructura de programa se añadirán los demás elementos del robot y se construirá el entorno donde trabajará el robot.

Todos los elementos que se mantengan fijos al entorno los se añadirán como un único sistema, y la garra de manipulación y los pedestales donde colocaremos los robots serán sistemas independientes. Teniendo en cuenta que los orígenes que se establecen van en relación con el elemento que nombremos como principal en la articulación, es decir a la hora de programar la posición de un elemento hay que tener en cuenta que su posición si se la otorgamos en la parte de la articulación siempre será en referencia al eje de coordenadas del elemento que ejerza de principal. Obteniendo el siguiente resultado:

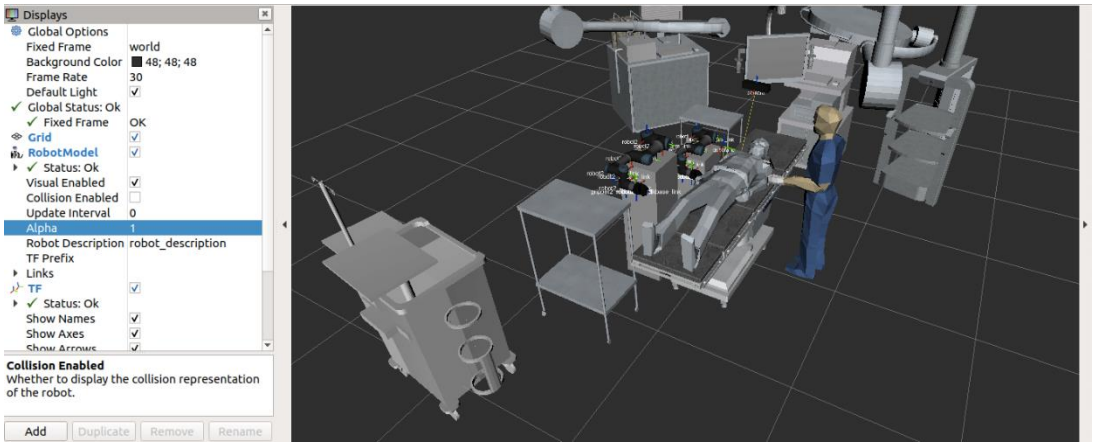


Figura 29: Ejecución del programa ur3.xacro con TF visible

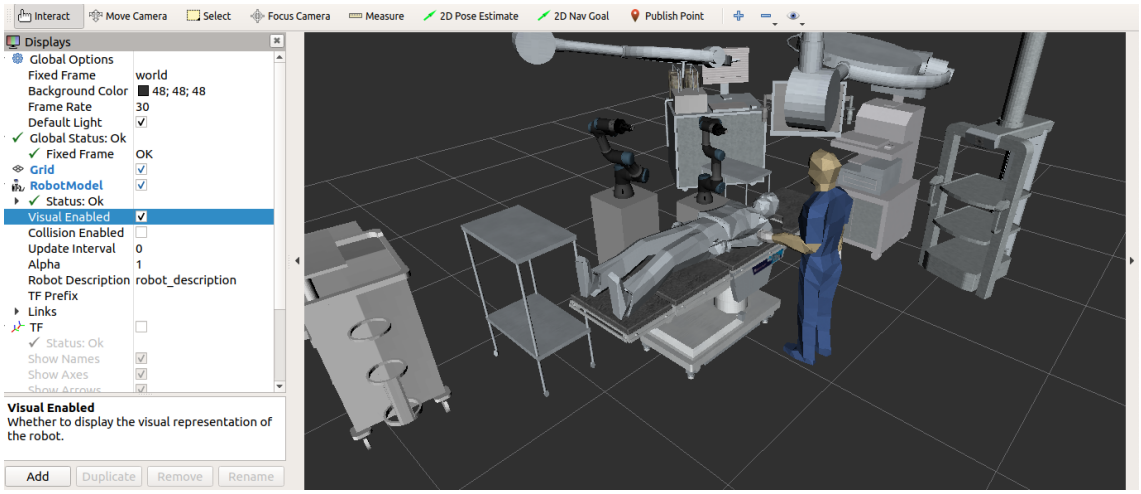


Figura 30: Ejecución del programa ur3.xacro

Es necesario para después realizar los movimientos y calcular la matriz de interacciones añadir a cada sistema su geometría de colisión, es decir, programar

una geometría a cada elemento que se encuentre con posibilidad de chocar con el robot, y que sea un objeto que se mantenga fijo en el entorno.

Por ejemplo, las personas son elementos variables por lo que no se les puede realizar una geometría de colisión.

Los robots también tienen su geometría de colisión, para tener en cuenta el espacio que ocupan en el sistema y que la colisión se mueva en relación con cada movimiento del robot, así evita colisionar con elementos del quirófano, con el otro robot e incluso con el mismo. Estas colisiones se pueden visualizar en Rviz.

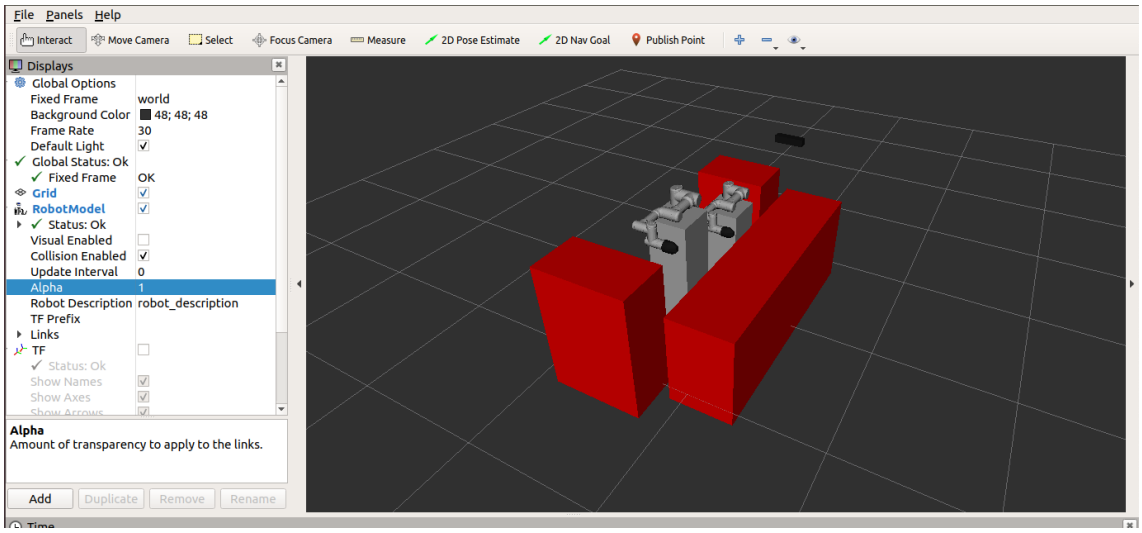


Figura 31: Visualización de las colisiones

El programa completo con las colisiones se encuentra en el anexo 2.

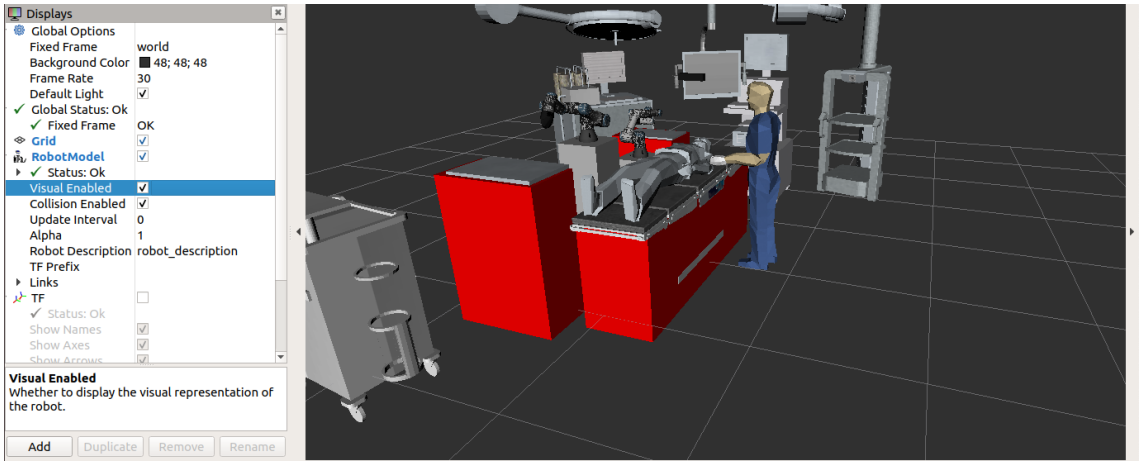


Figura 32: Programa ur3.xacro

5.3. Programación del entorno

Para la ejecución de los movimientos de los robots y realizar el programa de las trayectorias, se utilizará "*Moveit!",* esta aplicación se usa para obtener los nodos y realizar las conexiones de los enlaces de los robots y calcular su forma de trasmitirse dichos enlaces, para ejecutar los movimientos.

5.3.1. Moveit!

"MoveIt!" es un software de código abierto para ROS, que esta diseñado para la programación de la manipulación y movimientos de los robots. Su capacidad de paquetes y capacidad de diseño de control de movimiento es inmensa.[11]

Por lo que se utilizará un asistente propio de *"MoveIt!"* en el cual cargaremos nuestro programa ur3.xacro, ubicado en el paquete propio del proyecto de simulaciones.

Lo primero será ejecutar el comando en el terminal para abrir el asistente de la aplicación y seleccionar el programa a través del cual se realizaran los cálculos y la generación de los archivos necesarios.

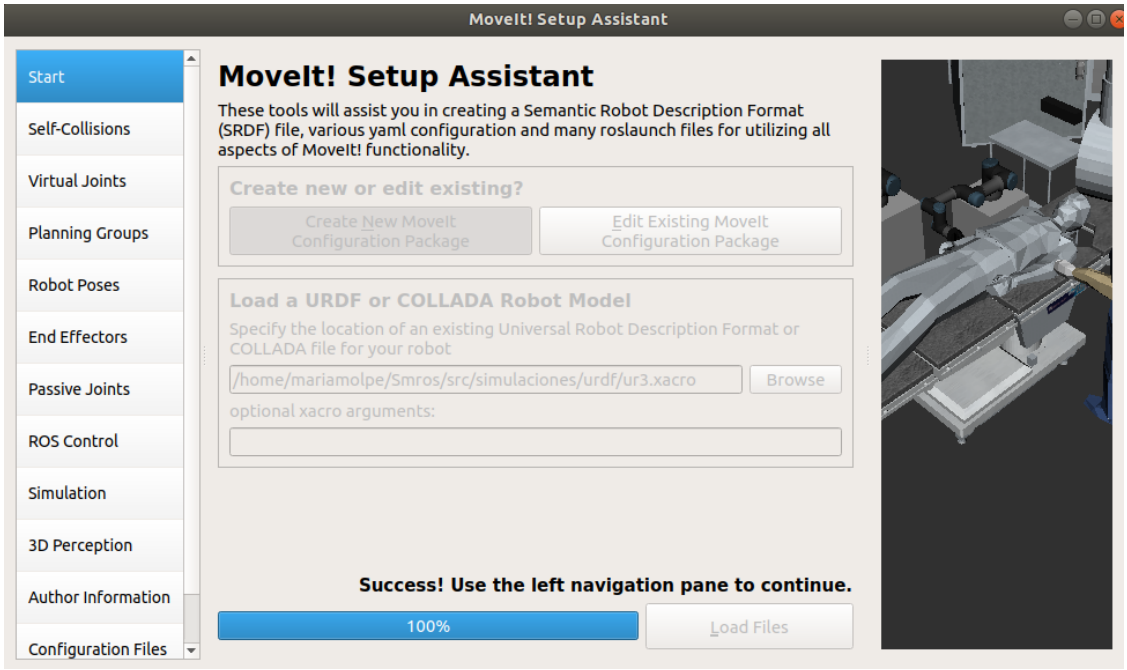


Figura 33: pantalla principal de "MoveIt" setup assistant

[11] https://github.com/ros-planning/moveit_tutorials/blob/master/doc/move_group_python_interface/scripts/move_group_python_interface_tutorial.py

Una vez compilado y cargado en el asistente lo primero será generar la matriz de auto-colisión.

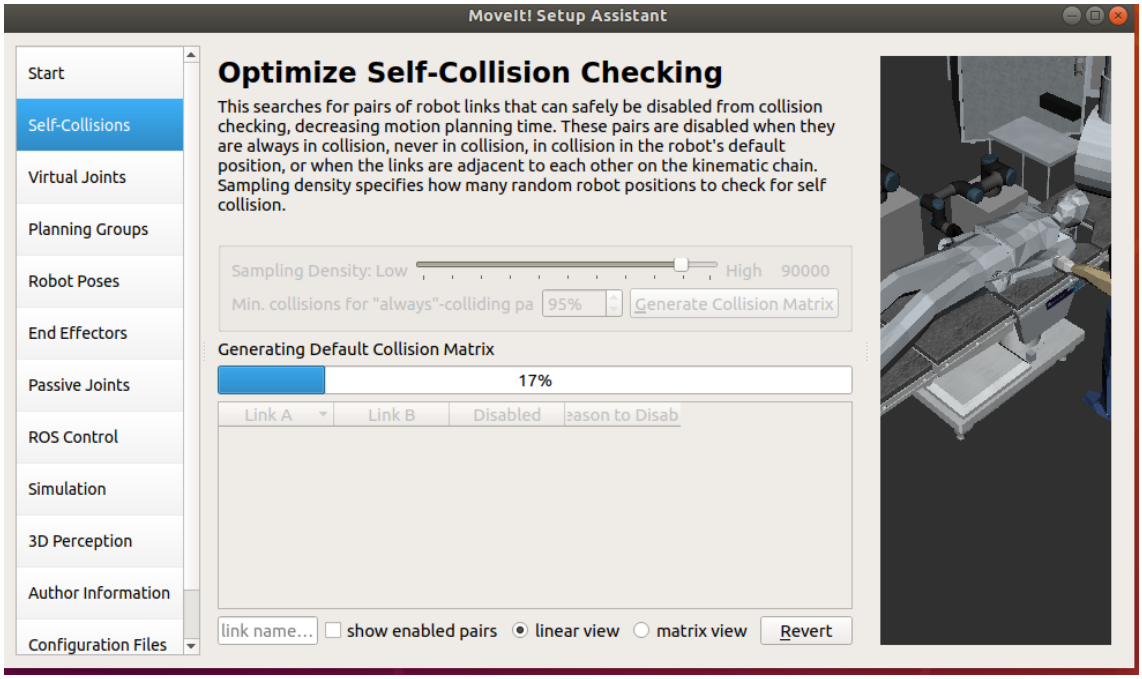


Figura 34: Pantalla de generar matriz de colisión

Lo que realiza este sistema es la búsqueda de pares de enlaces que se pueden deshabilitar de forma segura mediante la verificación de colisión, es decir, lo que proporciona la matriz es la capacidad de eliminar tiempo a la hora de la planificación de movimientos, ya que con esta matriz se descartan las colisiones que no son posibles, ya sea por distancia, por las limitaciones de los ejes del robot o por que son enlaces adyacentes.

Por ejemplo, el eje numero 1 del robot dos, nunca estaría en colisión con el sistema del quirófano, debido a que no se puede despejar de la base del pedestal.

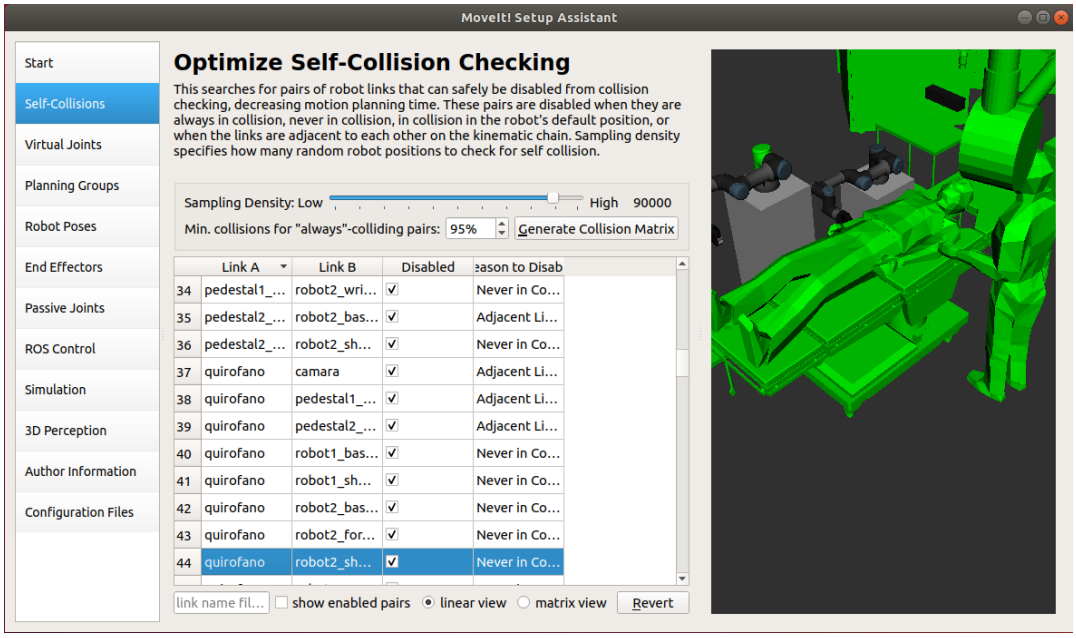


Figura 35: Ejemplo de pares de enlace que nunca pueden colisionar

Los parámetros que se ajustan a la hora de generar la matriz de colisión es el valor del numero de comprobaciones de colisión que se hacer de manera aleatoria, cuanto mas elevado mas comprobaciones hará, pero tardará mas en realizar el tiempo de procesamiento, para calcular dicha matriz.

Lo siguiente será realizar la agregación de articulaciones virtuales, lo se usará para realizar la fijación de la base de nuestro robot al sistema mundo. Lo que representa esta junta virtual es el movimiento del robot con relación a nuestro sistema base.

Esta articulación virtual se añade por igual a los dos robots.

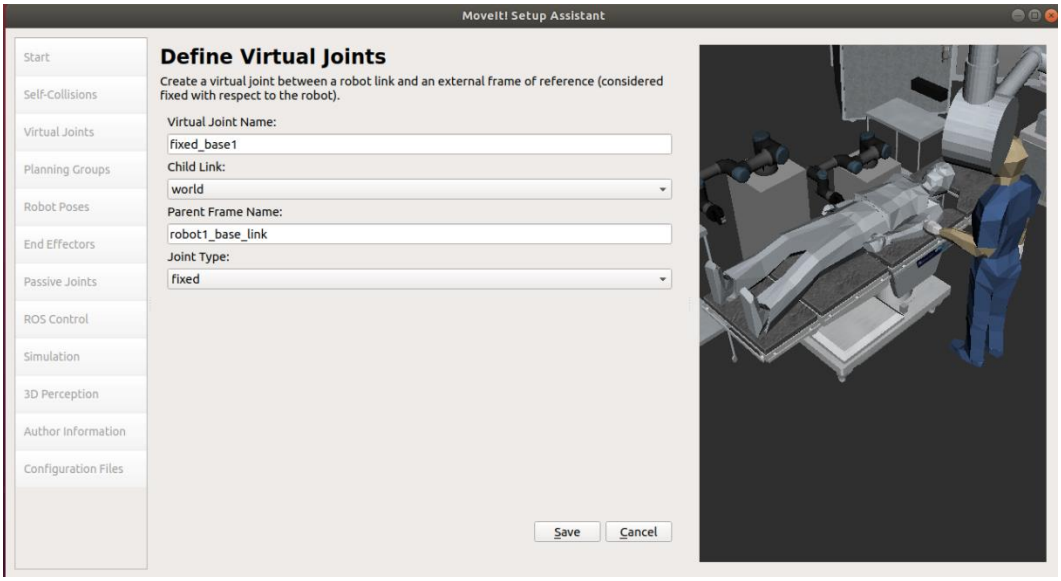


Figura 36: Agregar articulación virtual.

Es necesario agregar los grupos de planificación para describir semánticamente las diferentes partes del robot, y añadir la resolución cinemática de los robots, para ello se específica el nombre del grupo que se va a realizar y la solución cinemática que se quiera aplicar al sistema en este caso será *"trac_ik_kinematics_plugin/TRAC_IKKinematicsPlugin"* porque es un solucionador de cinemática inversa desarrollado por *TRACLabs* que combina dos implementaciones de IK mediante subprocesos para lograr soluciones más confiables que los solucionadores de IK de código abierto disponibles comunes, lo que proporciona un solucionador alternativo de cinemática inversa, ya que otros métodos no funcionan bien en presencia de limites de unión.

Los tiempos se dejará los valores predeterminados, pero para el algoritmo de planificación de movimientos usaremos la librería "RRTConnect", por su sencillez y su rápida conexión. Se establecen los mismos parámetros para los dos robots.

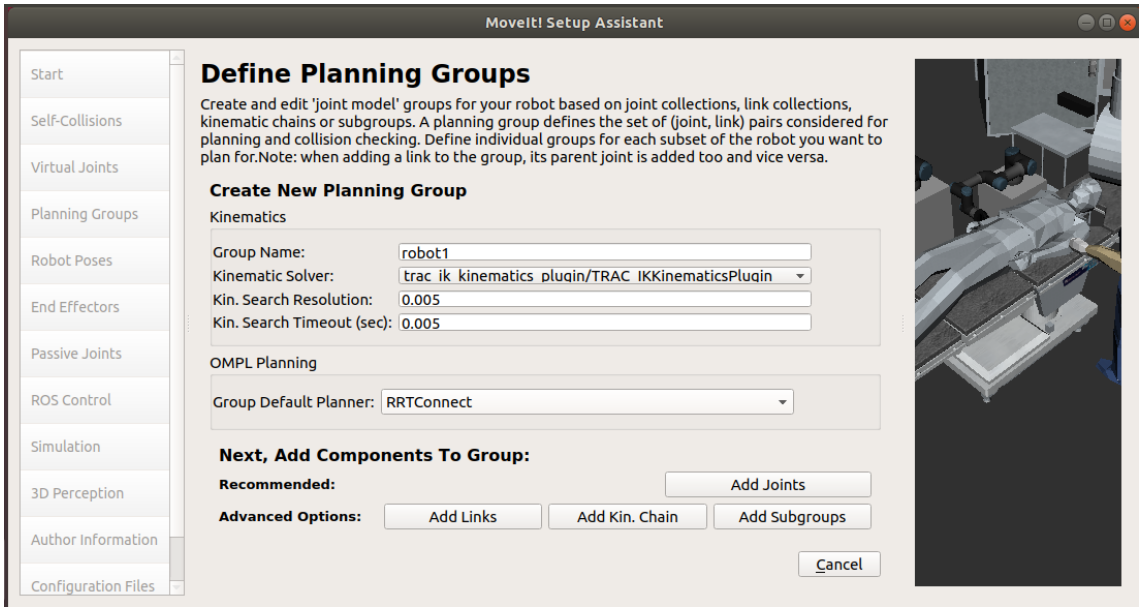


Figura 37: Pantalla de grupos de planificación

Lo siguiente es especificar la cadena cinemática de cada uno de los robots, en este caso el elemento "*Base_Link*" es la base de los robots y el elemento final "*Tip Link*" de la cadena es base de la garra robotiq.

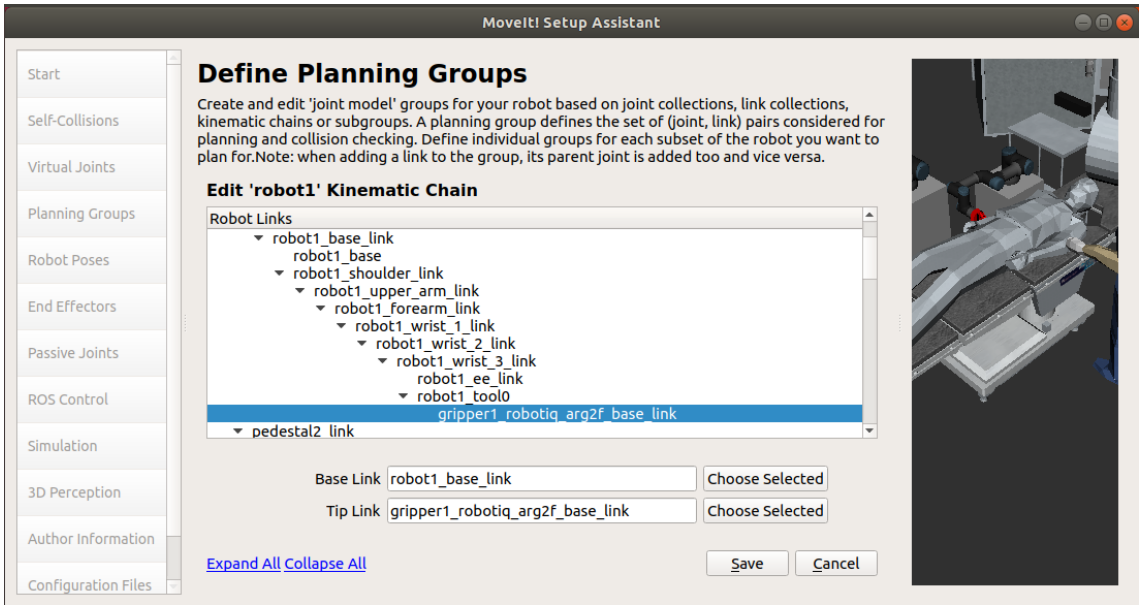


Figura 38:Pantalla para definir cadena cinemática

Se define para cada robot una posición de inicio, dándoles un nombre para la posición que definamos, en esta pantalla se pueden verificar las colisiones del robot con el mismo. Para comprobar que la definición del robot esta realizada correctamente.

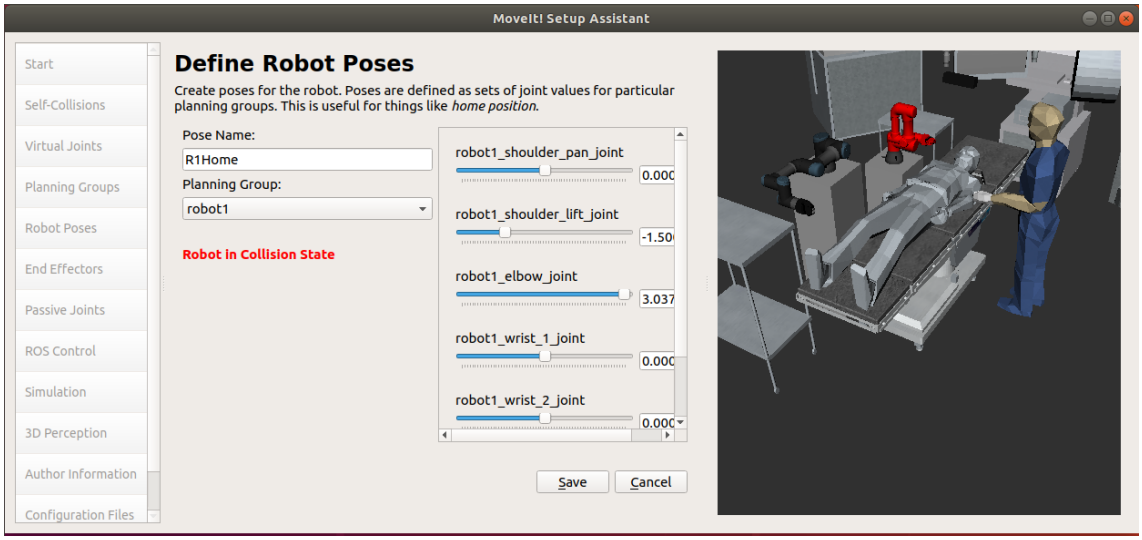


Figura 39: Robot 1 posición en colisión

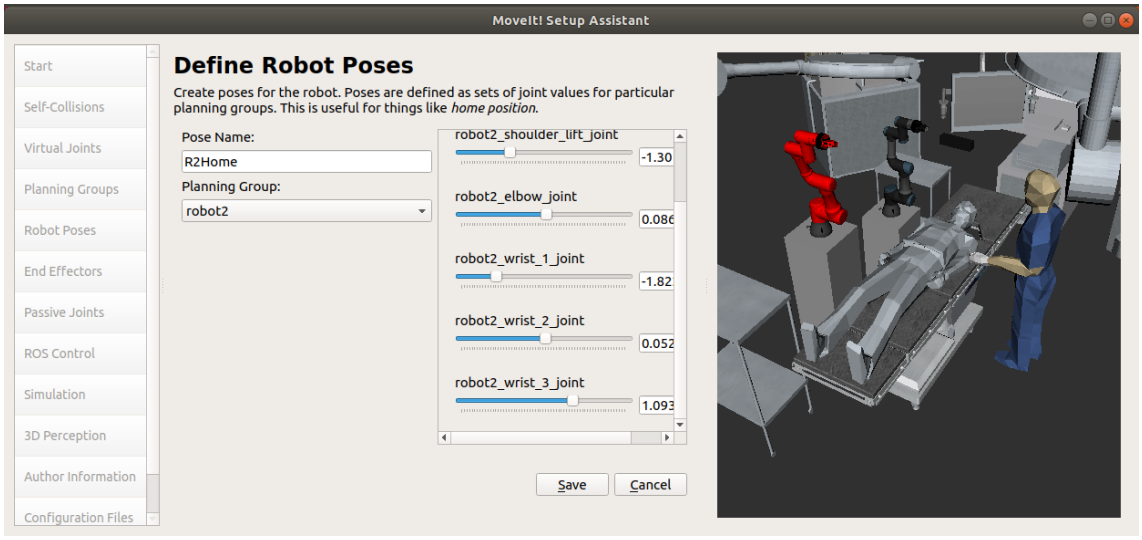


Figura 40: Robot 2 en posición Home

Lo último a realizar será definir cual será el enlace para el cual el robot realice el movimiento, en este caso son las dos garras de manipulación de cada robot, ya que son los elementos que se van a encargar de la manipulación de los objetos y en definitiva es el elemento que en una posición final de agarre importa donde se encuentre colocado. Esto se definen en el apartado de “End Effectors” y se crea uno para cada robot.

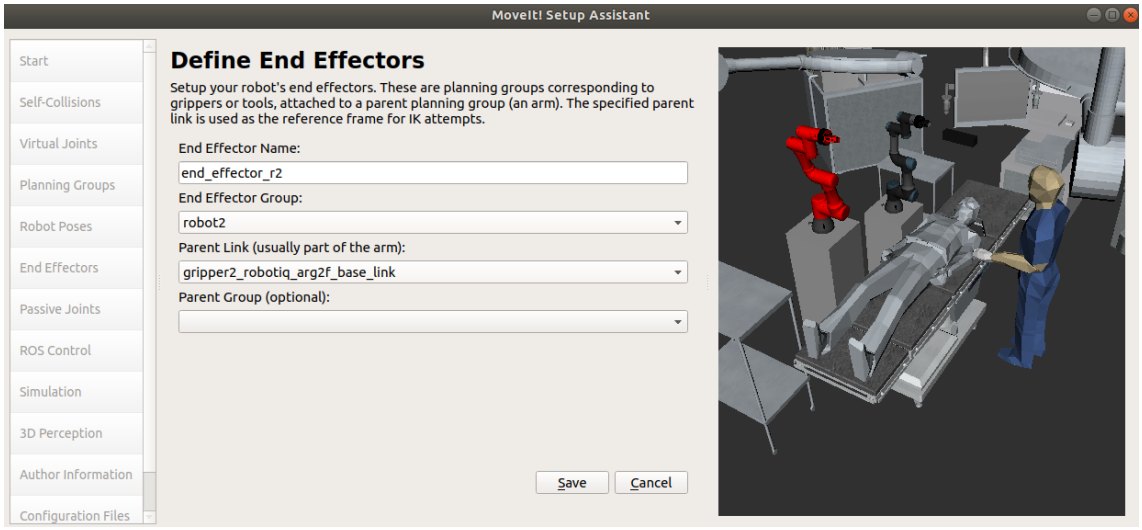


Figura 41: Pantalla "End Effectors" Robot 2

Con todos estos datos que se han añadido, más el autor y el email, que son estrictamente necesarios para añadirlos a cada documento que genere el asistente, se puede obtener todos los documentos necesarios para realizar los movimientos y la programación de trayectorias. Estos documentos que genera el asistente es necesario guardarlos en un paquete nuevo, en este caso se llamará “moveit” y se ubicará dentro de entorno de trabajo creado anteriormente, y es el paquete al que se accederá para llamar a los documentos que sean necesarios en el momento de realizar las trayectorias.

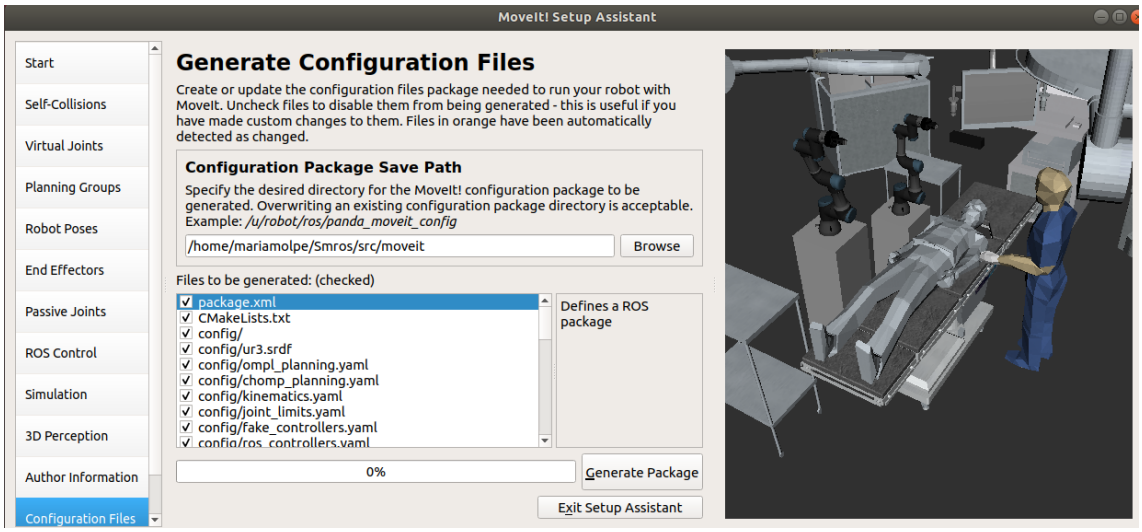


Figura 42: Pantalla generadora de archivos

Además de todos estos archivos, el asistente genera un documento en programación XML para URDF, donde se especifica cada colisión y cada transmisión de los enlaces que tienen las articulaciones móviles.

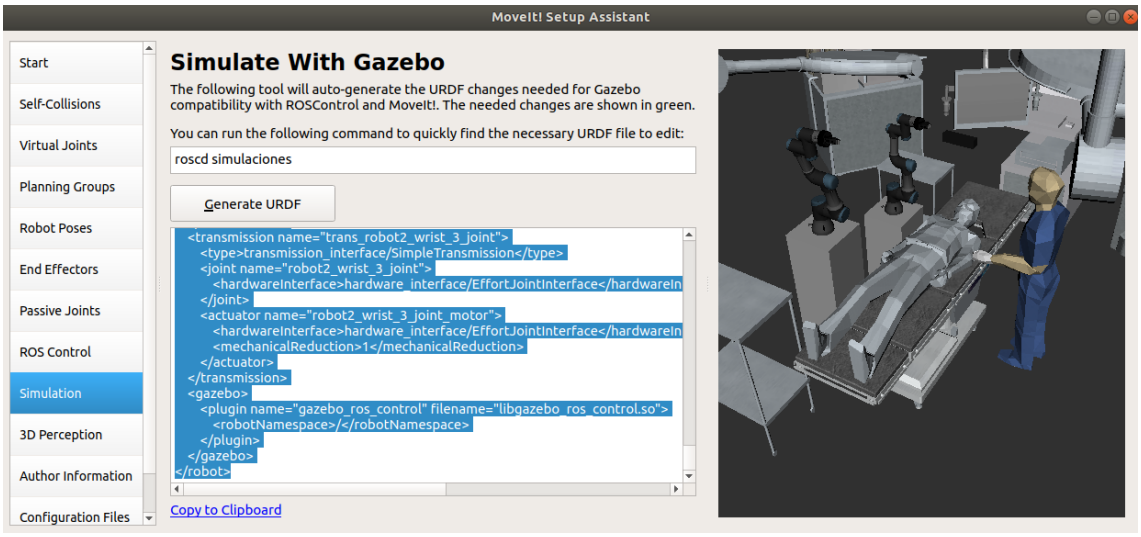


Figura 43: Pantalla generar URDF

Este archivo se puede simular en Rviz, visualmente no se diferencia en nada con el archivo que se ha creado anteriormente, pero a la hora de realizar el movimiento de los robots tiene generado un apartado para "*MotionPlanning*" que es el apartado donde se recogen toda la definición de trasmisiones e inercias que se han establecido en el nuevo archivo (Anexo 3).

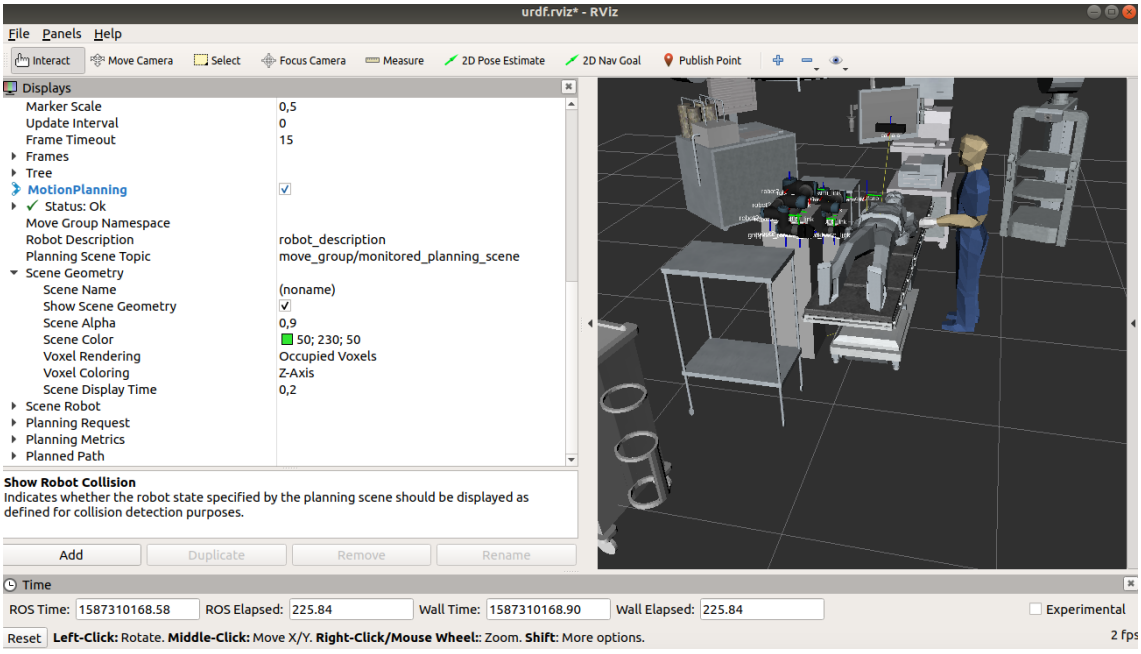


Figura 44: Ejecución en Rviz del archivo ur3moveit.xacro

5.3.2. Programa del entorno

Para realizar el programa se usará el lenguaje de programación Python, lo primero a ejecutar será el entorno para la simulación con todas las propiedades que se han explicado anteriormente.[12]

Para ello se ejecutará el documento *"demo.launch"* presente en el paquete moveit de la siguiente manera, primero se habilitará el entorno para trabajar con ROS se accederá al entorno Smros/src donde se encuentran los paquetes y se habilitará.

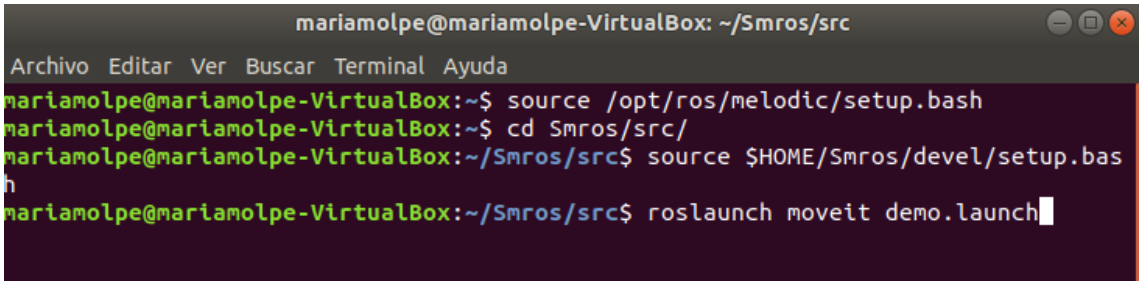


Figura 45: Comandos para ejecutar demo.launch

Al realizar la secuencia anterior de comando obtenemos el entorno diseñado simulado en *Rviz*, a diferencia de las simulaciones anteriores esta tiene activado la propiedad de *MotionPlanning* lo que permite realizar movimientos con los robots implantados de forma manual, es decir directamente en el entorno de simulación con el ratón o enviando al robot un movimiento determinado de sus ejes o incluso una operación final, todo esto operando directamente sobre el sistema de simulación *Rviz.*

Como se puede observar en la siguiente figura nuestro sistema de comunicación OMPL, es *RRTConect* el sistema que se seleccionó a la hora de realizar los documentos del sistema *Moveit!,* es el sistema que se encargará de calculara el movimiento de cada uno de los ejes del robot para llegar a la posición final que se establezca a la hora de realizar el programa, en el caso de ser imposible alcanzar la posición es el encargado de notificarlo con un error en los terminales de ejecución o si sufre el sistema algún punto de colisión.

Este sistema de comunicación OMPL puede ser cambiado en el entorno de simulación Rviz.

---

[12] http://docs.ros.org/melodic/api/moveit_tutorials/html/doc/move_group_python_interface/move_group_python_interface_tutorial.html#the-entire-code

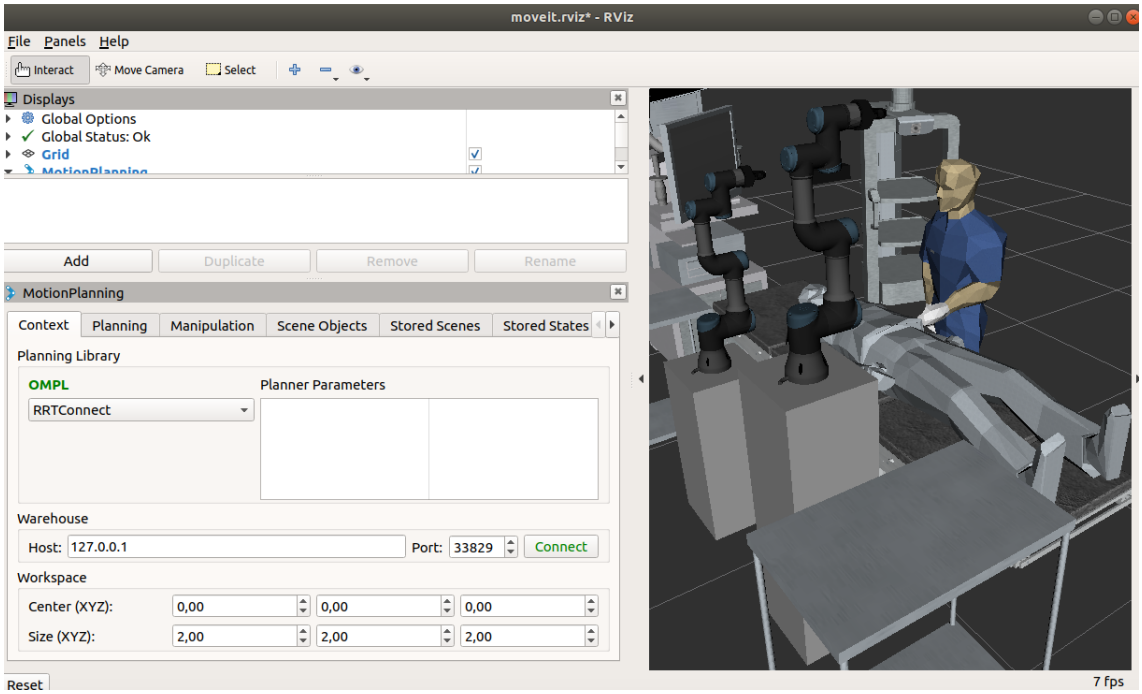


Figura 46: Resultado en Rviz de ejecutar demo.launch

Es importante tener en cuenta los nodos y topics que son habilitados a la hora de poner en marcha la simulación ejecutando los comandos, además de los documentos que se ejecutan.

El comando roslaunch lo que realiza es un llamada al documento que especificamos en la línea de comando, en este caso *demo.launch,* este a su vez hace una llamada a los diversos documentos de los que ha sido compuesto los cuales realizan las diferentes funciones que permiten la correcta ejecución del entorno.

Analizando parte por parte el documento, lo primero que se realiza en una llamada al archivo donde se determina el tipo y las características de la conexión en este caso se realiza el enlace al puerto local con un número de defecto y un host propio que permite realizar la simulación.

A continuación, se realiza la llamada al archivo que dispone de las unidades de descripción del entorno, hay que tener en cuenta que realiza una unión con nuestro archivo inicial, es decir sigue obteniendo los datos del programa del *ur3.xacro* (Anexo 2), esto permite realizar pequeños cambios en nuestro entorno, aunque ya se encuentren realizados los cálculos y los documentos relevantes. Lo que no esta permitido modificar son las colisiones y el sistema de planificación que se haya

definido, en este caso el sistema de los robots, las garras *robotiq* y los elementos geométricos de colisión que se generaron en la programación del entorno.
Y para finalizar la ultima llamada que se realiza en el programa es la llamada al sistema de simulación *Rviz,* para ello se ha generado un archivo donde se especifican las condiciones iniciales con las que se abrirá la simulación. Algunas como el sistema de referencia inicial o el sistema de unión de los elementos son elementos que si no son correctos pueden provocar que no se pueda abrir el programa, pero la mayoría son opcionales y modificables una vez lanzado el ejecutable.

A mayores se tiene la opción dentro del archivo **.launch* de deshabilitar la simulación en el programa *Rviz* y usar otro simulador como *Gazebo* o incluso los dos simuladores a la vez, lo que es imprescindible para poder ejecutar los simuladores es la creación de su propio archivo de ejecución donde se definen los parámetros iniciales del entorno.

Tras la ejecución obtenemos el siguiente esquema de nodos y topics, y la forma con la que se relacionan:

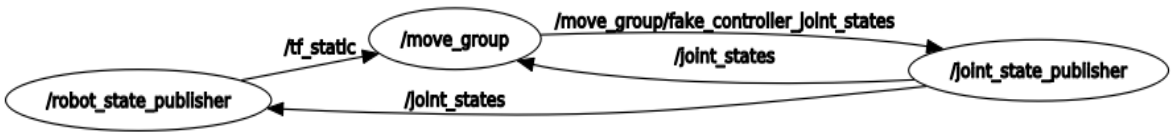


Figura 47: rqt_graph del archivo demo.launch

Como se observa existen tres nodos en ejecución con cuadro vías de comunicación. El nodo /robot_state_publisher publica información al nodo /move_group atraves del topic /tf_static este topic es del tipo tf2_msgs/TFMessage, lo que significa que se envia una cabecera en formato strings, un id de identificación del sistema de coordenadas, y el propio sistema de coordenadas, este sistema de coordenadas será la base de nuestro sistema de movimiento.

A su vez el nodo /robot_state está suscrito al topics /joint_states que se encarga de publicar el estado de los conjuntos móviles como es la posición la velocidad y el esfuerzo de cada uno de los ejes del robot, siempre precedido de un encabezado que lo identifique.
Y por último el nodo /move_group publica un topics que comparte tipo con el topics /joint_states, es decir, mandan la misma estructura de caracteres solo que cada nodo luego lo utiliza para funciones diferentes dentro de su programación.

Con esta base, se crea el programa donde se especifican los movimientos y objetivos que deben realizar los robots en la simulación. La estructura del programa se realiza mediante funciones, para un mayor orden y simpleza.

Para ello, se crea un nodo que se llama moveit_comander, para comunicarse con los nodos anteriores y se extraen los datos que sean necesarios y útiles para realizar la programación, a través de funciones, del propio nodo.
Para un uso mas sencillo del programa se realizará la creación de una clase para facilitar la declaración y el uso de todas las variables. Se manejará el uso de una sola de escena de simulación, pero de dos grupos de movimiento que son los dos robots, cada uno tendrá su grupo final de efecto que será la garra de cada robot y su propio planificador de movimiento. Para todo esto se crea una función, que se realizará su ejecución nada mas iniciar el programa.

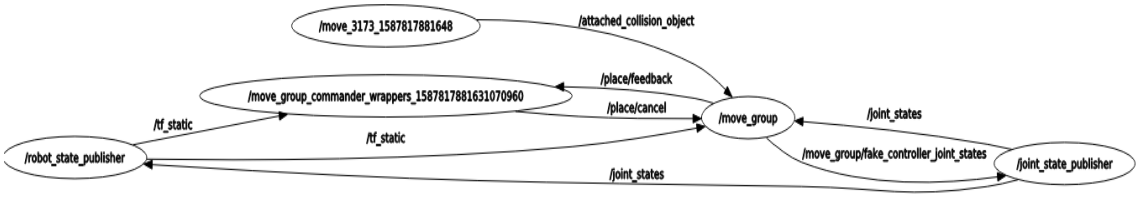


Figura 48: rqt_grahp programa move.py ene ejecución

Con estos datos ya se puede llamar a las diferentes funciones de movimiento de robot, para ello en la función se nombrará como argumento la clase creada en el inicio del programa.

Y lo primero, será especificar el robot que vaya a ejecutar el movimiento, y darle la posición de cada uno de los ejes que componen el robot, en el caso del UR3 está compuesto por seis ejes que todos realizan un movimiento rotatorio, por lo que se dará los valores de posición en radianes.

```
def go_to_Home_Robot1(self):

  robot = self.move_group1
  joint_goal = robot.get_current_joint_values()
  joint_goal[0] = 0
  joint_goal[1] = -1.59
  joint_goal[2] = 0
  joint_goal[3] = -1.59
  joint_goal[4] = 0
  joint_goal[5] = -1.59
  # Esperamos a que el robot llege a la trayectoria expecificada
  robot.go(joint_goal, wait=True)
  # Paramos los robot para no generar residuos
  robot.stop()
  current_joints = robot.get_current_joint_values()
  return all_close(joint_goal, current_joints, 0.01)
```

Figura 49: Función go_to_Home_Robot1

Se manda la posición al robot que debe de ejecutar la orden y se espera que realice la trayectoria completa, una vez en esa posición se borran los datos restantes y para no ir acumulando residuos de movimiento en el robot. Con el movimiento realizado se guardan los datos de posición verificando la existencia de colisiones, en caso de colisión, obtendríamos un fallo que nos impediría realizar el movimiento. Todo este calculo de posición lleva consigo además del tiempo que tarda en ejecutarse la trayectoria, el tiempo que tarda en calcular para realizar el movimiento de cada eje para alcanzar a la posición definida sin colisiones. Este tiempo se ve afectado por la matriz de colisión calculada con el asistente, cuanto mas pares de enlaces sea capaz de calcular la matriz de colisión mas rápido será el sistema a la hora de empezar la ejecución del movimiento deseado.

Con la misma estructura anteriormente definida se crean las funciones para los dos grupos de movimiento, cambiando solo el valor de la variable robot. Siempre teniendo en cuenta a la hora de dar valor a los ejes posibles colisiones o fallos de limite de ejes, ya que ocasionaría errores en el programa.

Para realizar el agarre de algún objeto hay que tener en cuenta que no se puede declarar igual que cualquier parte del entorno, debido a que si se genera como objeto

del entorno no podría relacionarse con ningún elemento propio del grupo de movimiento.

Por lo que durante la ejecución del programa hay que ejecutar funciones que añadan un objeto en la escena, esto se hace después del que el robot haya alcanzado la posición de destino donde se ha de realizar el agarre, una vez en dicha posición, se define el objeto que se quiere agregar, dándole un nombre, el cual hemos definido previamente en la clase de la función inicial. Se declara la escena donde se va a situar y su posición en este caso la única que existe y añadimos su función de geometría, además el nombre del elemento que es capaz de sujetar el objeto, el nombre tiene que ser el mismo que se uso para su definición en el macro XACRO.

```
def add_camara(self, timeout=4):

  camara_name = self.camara_name
  scene = self.scene

  camara_pose = geometry_msgs.msg.PoseStamped()
  camara_pose.header.frame_id = "gripper1_robotiq_arg2f_base_link"
  camara_pose.pose.orientation.w = 1.0
  camara_pose.pose.position.z = 0.08
  camara_pose.pose.position.x = -0.08
  camara_name = "camara"
  scene.add_box(camara_name, camara_pose, size=(0.2, 0.01, 0.01))

  self.camara_name=camara_name
  return self.wait_for_state_update_camara(camara_is_known=True, timeout=timeout)
```

Figura 50: Función add_camara

Para definir la posición es necesario utilizar un tipo de mensaje donde su estructura nos permita añadir la ubicación en cuaternios, además del nombre del enlace que hace la función de sistema de referencia para la posición.

Por ultimo, agregamos a la función que se encarga de añadir el objeto, con sus propiedades necesarias de la función en este caso son: el nombre, la posición creada y las dimensiones de la geometría que se requiera.

Existen diversas funciones para la creación de diferentes objetos cada una necesita unos parámetros obligatorios y opcionales.

Cuando un objeto es creado y se define su geometría, se añade también las dimensiones de colisión del propio objeto, por lo que si se realiza un movimiento donde pueda existir colisión con objeto recién añadido a la escena, el programa no será capaz de llegar a la posición de destino.

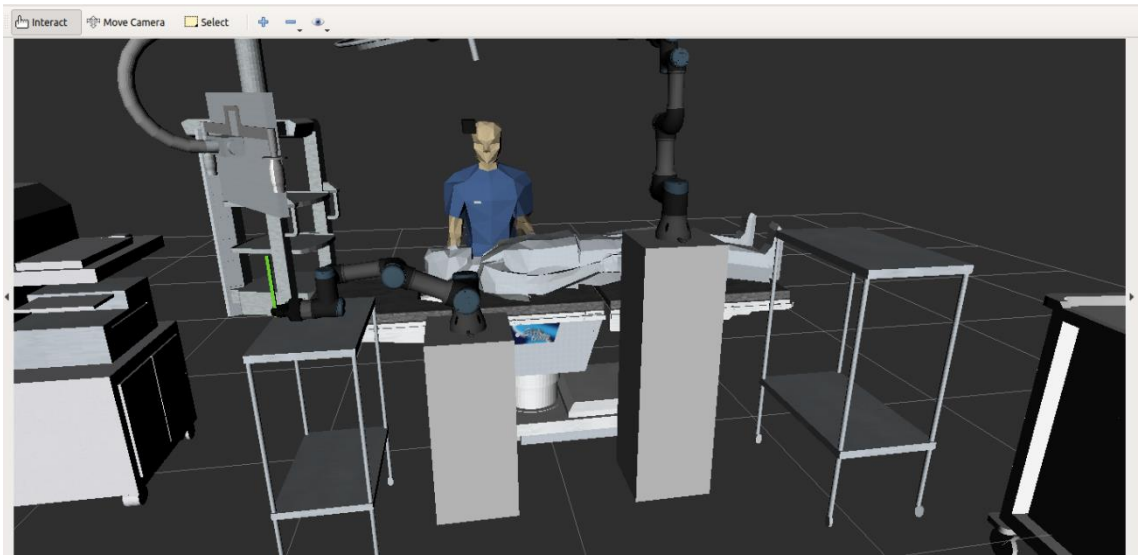


Figura 51: Ejecutando la función add_camara

Una vez creado el objeto se ejecuta la función de agarre, para lo cual se define el grupo de movimiento que va a sujetar el objeto, el nombre del objeto en cuestión y el enlace final de la cadena dinámica que se definió anteriormente, para saber de que enlace depende el objeto para realizar el movimiento.

```
def attach_camara(self, timeout=4):

  camara_name = self.camara_name
  robot = self.robot
  scene = self.scene
  eef_link = self.eef_link1
  group_names = self.group_names

  grasping_group = 'robot1'
  touch_links = robot.get_link_names(group=grasping_group)
  scene.attach_box(eef_link, camara_name, touch_links=touch_links)
  rospy.sleep(1)
```

Figura 52: Función attach_camara

Visualmente cuando se realiza correctamente el agarre del objeto definido, se produce un cambio de color de verde a morado, por lo cual cuando el objeto se crea tiene un color verde y cuando este agarrado por la garra cambia a morado manteniendo ese color hasta que se suelte el objeto.

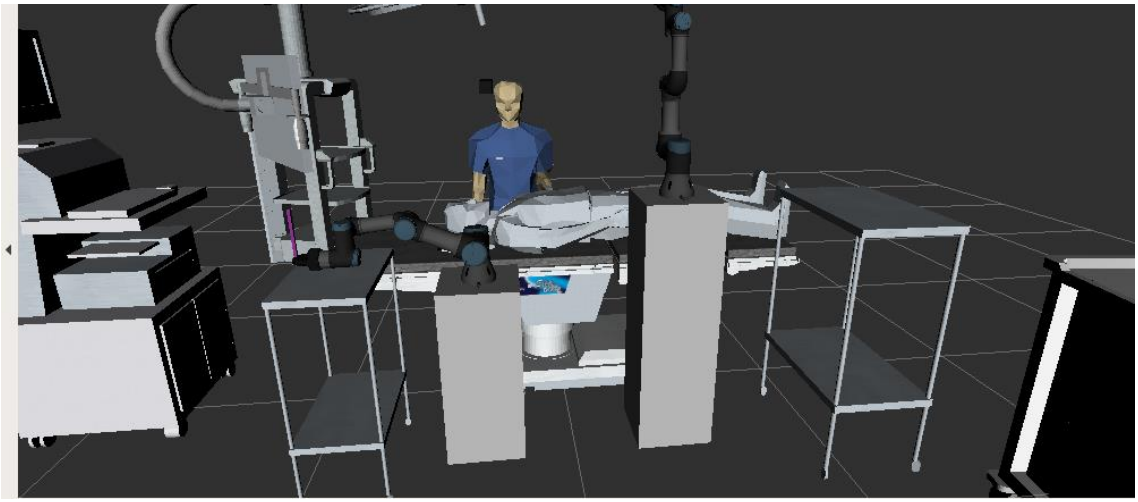


Figura 53: Ejecutando función attach_camara

Una vez que el objeto se quiera liberar hay que hacer un proceso similar, pero a la inversa, creando un programa de liberación del objeto con mismos parámetros de se usan para coger el objeto, solo que cambiando la función final y ejecutarlo cuando el robot este situado en la posición de dejada y a continuación, eliminar el objeto de la escena con el uso de otra función, muy sencilla donde solo es necesario especificar el nombre del objeto a eliminar y la escena.

Con esta base se crea el programa, donde los robots se inicializan en la posición HOME la cual coloca los dos robots en posición vertical hacia arriba, seguido de esta acción el robot posicionado mas cercano a la cabeza del paciente coge la cámara endoscópica y la posiciona dentro de la nariz del paciente para comenzar la operación.

```
def detach_instru2(self, timeout=4):

  instru2_name = self.instru2_name
  scene = self.scene
  eef_link = self.eef_link2

  scene.remove_attached_object(eef_link, name=instru2_name)
  return self.wait_for_state_update_instru2(instru2_is_known=True, instru2_is_attached=False, timeout=timeout)

def remove_instru2(self, timeout=4):
  instru2_name = self.instru2_name
  scene = self.scene
  scene.remove_world_object(instru2_name)

  return self.wait_for_state_update_instru2(instru2_is_attached=False, instru2_is_known=False, timeout=timeout)
```

Figura 54: Funciones detach_intru2 y remove_intru2

En este paso, el robot que se encuentra libre espera las ordenes de ejecución, para saber el elemento que tiene que acercar al cirujano, cuando recibe la orden se

posiciona coloca coge el instrumento indicado, y se le acerca al cirujano que en ese momento este realizando la operación.

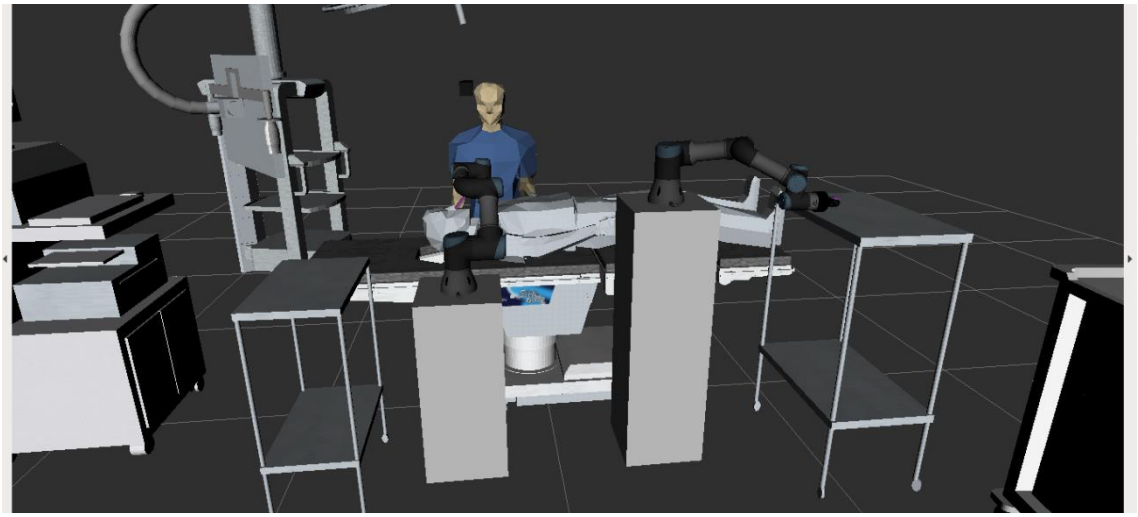


Figura 55: Programa en ejecución

Para no estorbar el robot se vuelve a su posición HOME hasta que el cirujano, de la orden de devolver el objeto, en cirujano da el objeto al robot que vuelve a posicionarlo en la mesa con los demás instrumentos. Así hasta que el se de por finalizada la operación, donde el robot que sujeta la cámara endoscópica vuelve a dejar el utensilio en su sitio y se vuelve a la posición HOME dando el programa por concluido.

6. Conclusiones

La robótica colaborativa tiene numerosas salidas en muchísimos campos debido a su simpleza y facilidad a la hora de colocarlos en cualquier medio; ya que al ser capaces de trabajar a velocidad colaborativa no es preciso de disponer mayores medidas de seguridad como puede ser en un robot convencional donde es necesario asegurar todo el recinto de trabajo del robot con medidas de seguridad y amplias distancias prudenciales.

Por lo que un robot colaborativo puede ser de gran utilidad en ámbitos como la medicina, para mejorar operaciones o hacerlas mas sencillas para unidad medica como se pretende hacer en este proyecto.

A mayores el uso de un sistema operativo robótico como es ROS que permite programar una amplia gama de robot desde un código abierto y general sin necesidad de conocer el lenguaje de programación especifico de cada gama de robot, lo que permite generar código para robot de diferentes marcas.

Además, ROS proporciona un sistema de simulación del entorno real y gratuito, para las diferentes pruebas para así poder evitar golpes y problemas cuando se utilice el robot de manera física.

Otra ventaja del sistema ROS frente al uso de la programación convencional es que un robot no es capaz de evitar colisionarse ya que desconoce su entorno e incluso si existe otro robot que trabaja compartiendo su mismo espacio, son incapaces de conocer la posición del robot que trabaja con ellos.

Esto provoca que en su programación sea necesario el uso de otro sistema que permita y acceda a las señales y al estado del robot, para poder evitar choques de los robots entre si, un ejemplo claro es la creación de un “semáforo” donde un robot deshabilita la zona para que los demás robots no puedan pasar y bloquear su movimiento mientras él se encuentre trabajando y así evita la colisión, pero estas señales del semáforo tienen que ser manejadas por otro elemento como puede ser un autómata y hacer de vía de enlace para la comunicación de los robots, además provocan retrasos en el proceso.

En cambio, mediante el uso de ROS es el sistema es el que se encarga de verificar la posición del todos robot y elementos del entorno para no sufrir colisiones, lo que permite evitar la creación de un programa especifico para evitar las colisiones en las zonas donde varios robots realicen diferentes trabajos y también poder prescindir del elemento que hace de vía de comunicación entre los diferentes robots y

elementos que forman la célula robótica. Haciendo que el trabajo del sistema sea mas rápido y fluido.

7. Bibliografía

- https://www.universal-robots.com/es/productos/robot-ur3/
- http://docs.ros.org/melodic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html
- https://robotiq.com/es/productos/pinza-adaptable-2f85-140
- http://wiki.ros.org/
- https://docs.aws.amazon.com/es_es/robomaker/latest/dg/simulation-tools-rviz.html
- https://github.com/ros-planning/moveit_tutorials/blob/master/doc/move_group_python_interface/scripts/move_group_python_interface_tutorial.py
- https://github.com/ros-planning/moveit_tutorials/blob/melodic-devel/doc/move_group_python_interface/scripts/move_group_python_interface_tutorial.py
- http://docs.ros.org/melodic/api/moveit_commander/html/classmoveit__commander_1_1move__group_1_1MoveGroupCommander.html
- http://docs.ros.org/melodic/api/moveit_commander/html/classmoveit__commander_1_1planning__scene__interface_1_1PlanningSceneInterface.html#af50ab6461a7c80451959b9a7a7668c77
- http://docs.ros.org/melodic/api/moveit_tutorials/html/doc/move_group_python_interface/move_group_python_interface_tutorial.html#the-entire-code
- https://www.elespanol.com/ciencia/salud/20180412/extrae-tumor-cerebro-traves-nariz-milagro-cirugia/298971162_0.html
- https://www.vozpopuli.com/altavoz/next/sacar-tumor-cerebro-nariz_0_1065494746.html
- https://www.aer-automation.com/mercados-emergentes/robotica-colaborativa/

8. Anexos

8.1. Anexo 1: Programa ur3.urdf

```
<?xml version="1.0"?>
<robot name="ur3">
  <link name="world"/>
  <link name="base">
    <visual>
      <geometry>
        <cylinder length="0.20" radius="0.2"/>
      </geometry>
      <material name="metal">
         <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </visual>
  </link>
  <joint name="union_base" type="fixed">
    <parent link="world"/>
    <child link="base"/>
  </joint>
  <link name="eje1">
        <visual>
                 <geometry>
                   <cylinder length="0.5" radius="0.15"/>
       </geometry>
       <origin rpy="0 0 0" xyz="0 0 0.25"/>
       <material name="blue">
          <color rgba="0 0 0.8 1"/>
       </material>
    </visual>
  </link>
  <joint name="eje1" type="continuous">
        <axis xyz="0 0 1"/>
    <parent link="base"/>
    <child link="eje1"/>
    <origin rpy="0 0 0" xyz="0 0 0.05"/>
```

```
</joint>
<link name="eje2">
       <visual>
                <geometry>
                  <cylinder length="0.35" radius="0.15"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0.15"/>
      <material name="metal">
         <color rgba="0.75 0.75 0.75 1"/>
      </material>
  </visual>
</link>
<joint name="eje2" type="continuous">
       <axis xyz="0 0 1"/>
  <parent link="eje1"/>
  <child link="eje2"/>
  <origin rpy="0 1.5708 0" xyz="0.15 0 0.35"/>
</joint>
<link name="brazo2">
  <visual>
    <geometry>
      <cylinder length="0.70" radius="0.15"/>
    </geometry>
    <material name="metal">
       <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 1.5708 0" xyz="-0.40 0 0.15"/>
  </visual>
</link>
<joint name="ejeBrazo2" type="fixed">
  <parent link="eje2"/>
  <child link="brazo2"/>
</joint>
<link name="eje3">
  <visual>
    <geometry>
      <cylinder length="0.45" radius="0.15"/>
    </geometry>
```

```
      <material name="blue">
           <color rgba="0 0 0.8 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </visual>
  </link>
  <joint name="eje3" type="continuous">
         <axis xyz="0 0 1"/>
    <parent link="brazo2"/>
    <child link="eje3"/>
    <origin rpy="0 0 0" xyz="-0.55 0 -0.1"/>
  </joint>
  <link name="brazo3">
    <visual>
      <geometry>
        <cylinder length="0.70" radius="0.13"/>
      </geometry>
      <material name="blue">
           <color rgba="0 0 0.8 1"/>
      </material>
      <origin rpy="0 1.5708 0" xyz="-0.30 0 -0.05"/>
    </visual>
  </link>
  <joint name="brazoeje3" type="fixed">
    <parent link="eje3"/>
    <child link="brazo3"/>
  </joint>
  <link name="eje4">
    <visual>
      <geometry>
        <cylinder length="0.45" radius="0.10"/>
      </geometry>
      <material name="metal">
         <color rgba="0.75 0.75 0.75 1"/>
      </material>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </visual>
  </link>
```

```
<joint name="eje4" type="continuous">
       <axis xyz="0 0 1"/>
  <parent link="brazo3"/>
  <child link="eje4"/>
  <origin rpy="0 0 0" xyz="-0.55 0 0.1"/>
</joint>
<link name="eje5">
  <visual>
    <geometry>
      <cylinder length="0.30" radius="0.10"/>
    </geometry>
    <material name="metal">
       <color rgba="0.75 0.75 0.75 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0"/>
  </visual>
</link>
<joint name="eje5" type="continuous">
       <axis xyz="0 0 1"/>
  <parent link="eje4"/>
  <child link="eje5"/>
  <origin rpy="0 1.5708 0" xyz="-0.15 0 0.1"/>
</joint>
<link name="eje6">
  <visual>
    <geometry>
      <cylinder length="0.30" radius="0.08"/>
    </geometry>
    <material name="blue">
         <color rgba="0 0 0.8 1"/>
    </material>
    <origin rpy="0 0 0" xyz="0 0 0"/>
  </visual>
</link>
<joint name="eje6" type="continuous">
       <axis xyz="0 0 1"/>
  <parent link="eje5"/>
  <child link="eje6"/>
```

```
    <origin rpy="0 1.5708 0" xyz="-0.08 0 -0.05"/>
  </joint>
</robot>
```

8.2. Anexo 2: Programa ur3.xacro

```
<?xml version="1.0" ?>
<robot name="ur3" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- world -->
  <link name="world"/>

  <!-- pedestal 1 -->
  <link name="pedestal1_link">
    <visual>
      <origin xyz="0 0 0.35"/>
      <geometry>
        <box size="0.3 0.3 0.7"/>
      </geometry>
      <material name="pedestal_color">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
    </visual>

    <collision>
      <origin xyz="0 0 0.35"/>
      <geometry>
        <box size="0.3 0.3 0.7"/>
      </geometry>
    </collision>

    <inertial>
      <mass value="500"/>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
    </inertial>
  </link>

  <!-- pedestal 2 -->
  <link name="pedestal2_link">
    <visual>
      <origin xyz="0 0 0.5"/>
```

```
      <geometry>
        <box size="0.3 0.3 1"/>
      </geometry>
      <material name="pedestal_color">
        <color rgba="0.75 0.75 0.75 1"/>
      </material>
    </visual>

    <collision>
      <origin xyz="0 0 0.5"/>
      <geometry>
        <box size="0.3 0.3 1"/>
      </geometry>
    </collision>

    <inertial>
      <mass value="500"/>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001"/>
    </inertial>
  </link>

  <!-- Robot1 -->
  <xacro:include filename="$(find ur3_description)/urdf/ur3.urdf.xacro"/>
  <xacro:ur3_robot prefix="robot1_" joint_limited="true"/>

   <!-- Robotiq Gripper1 -->
  <xacro:include filename="$(find robotiq_2f_85_gripper_visualization)/urdf/robotiq
_arg2f.xacro"/>
  <xacro:robotiq_arg2f_base_link prefix="gripper1_"/>

  <!-- Robot2 -->
  <xacro:include filename="$(find ur3_description)/urdf/ur3.urdf.xacro"/>
  <xacro:ur3_robot prefix="robot2_" joint_limited="true"/>

   <!-- Robotiq Gripper2 -->
  <xacro:include filename="$(find robotiq_2f_85_gripper_visualization)/urdf/robotiq
_arg2f.xacro"/>
```

```
  <xacro:robotiq_arg2f_base_link prefix="gripper2_"/>


   <!-- .dae quirofano -->
   <link name="quirofano">
    <visual>
      <origin rpy="1.58 0 0" xyz="0 0 -1.0"/>
      <geometry>
        <mesh filename="package://urdf_tutorial/meshes/quirofano.dae"/>
        <scale>0.4 0.4 0.4</scale>
      </geometry>
    </visual>
    <collision>
      <origin xyz="0 0.1 -0.75" rpy="0 0 0" />
      <geometry>
        <box size="2 0.5 0.75" />
      </geometry>
    </collision>
    <collision>
      <origin xyz="1.1 -0.55 -0.55" rpy="0 0 0" />
      <geometry>
        <box size="0.4 0.7 1" />
      </geometry>
    </collision>
    <collision>
      <origin xyz="-0.85 -0.55 -0.6" rpy="0 0 0" />
      <geometry>
        <box size="0.4 0.6 0.75" />
      </geometry>
    </collision>
  </link>
<!--camara -->
  <link name="camara">
      <visual>
        <origin xyz="0 0 0.025" rpy="0 0 0" />
        <geometry>
          <box size="0.05 0.2 0.05" />
        </geometry>
        <material name="dark_grey">
```

```
          <color rgba="0.1 0.1 0.1 1.0"/>
        </material>
      </visual>
      <collision>
        <origin xyz="0 0 0.025" rpy="0 0 0" />
        <geometry>
          <box size="0.05 0.2 0.05" />
        </geometry>
        <material name="dark_grey"/>
      </collision>
      <inertial>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <inertia ixx="0.1" ixy="0" ixz="0"
                 iyy="0.1" iyz="0"
                 izz="0.1" />
        <mass value="1" />
      </inertial>
    </link>
  <!-- Joints -->
  <!-- pedestal1 to mundo. -->
  <joint name="world_to_pedestal1" type="fixed">
    <parent link="world" />
    <child link="pedestal1_link" />
    <origin xyz="-0.35 -0.4 0.0" rpy="0.0 0.0 0.0"/>
  </joint>

  <!-- robot1 to pedestal1. -->
  <joint name="pedestal1_to_robot1" type="fixed">
    <parent link="pedestal1_link" />
    <child link="robot1_base_link" />
    <origin xyz="0 0 0.7" rpy="0 0 0" />
  </joint>

   <!-- gripper1 to robot1. -->
  <joint name="gripper1_to_robot1" type="fixed">
    <parent link="robot1_tool0" />
    <child link="gripper1_robotiq_arg2f_base_link" />
  </joint>
```

```
  <!-- pedestal2 to mundo. -->
  <joint name="world_to_pedestal2" type="fixed">
    <parent link="world" />
    <child link="pedestal2_link" />
    <origin xyz="0.4 -0.4 0.0" rpy="0.0 0.0 0.0"/>
  </joint>

  <!-- robot2 to pedestal1. -->
  <joint name="pedestal2_to_robot1" type="fixed">
    <parent link="pedestal2_link" />
    <child link="robot2_base_link" />
    <origin xyz="0 0 1" rpy="0 0 0" />
  </joint>

   <!-- gripper1 to robot1. -->
  <joint name="gripper2_to_robot1" type="fixed">
    <parent link="robot2_tool0" />
    <child link="gripper2_robotiq_arg2f_base_link" />
  </joint>

  <!-- quirofano to world. -->
  <joint name="quirofano_to_world" type="fixed">
    <parent link="world" />
    <child link="quirofano" />
    <origin xyz="0.0 0.0 1" rpy="0 0 0" />
  </joint>

<!-- camara to world. -->
  <joint name="world_to_camera_joint" type="fixed">
    <parent link="world"/>
    <child link="camara"/>
    <origin xyz="-0.3 0.1 1.35" rpy="0 0 0" />
  </joint>



</robot>
```

8.3. Anexo 3: Programa ur3moveit.xacro

```
<?xml version="1.0" encoding="utf-8" ?>

<!-- =================================================================================== -->

<!-- |    This document was autogenerated by xacro from /home/mariamolpe/Smros/src/simulaciones/urdf/ur3.xacro | -->

<!-- |    EDITING THIS FILE BY HAND IS NOT RECOMMENDED                                  | -->

<!-- =================================================================================== -->

<robot name="ur3">

    <!-- world -->

    <link name="world" />

    <!-- pedestal 1 -->

    <link name="pedestal1_link">

        <visual>

            <origin xyz="0 0 0.35" />

            <geometry>

                <box size="0.3 0.3 0.7" />

            </geometry>

            <material name="pedestal_color">

                <color rgba="0.75 0.75 0.75 1" />

            </material>

        </visual>

        <collision>

            <origin xyz="0 0 0.35" />

            <geometry>

                <box size="0.3 0.3 0.7" />

            </geometry>

        </collision>


<inertial>

            <mass value="500" />

            <origin rpy="0 0 0" xyz="0 0 0" />

            <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001" />

        </inertial>
```

```
    </link>
    <!-- pedestal 2 -->
    <link name="pedestal2_link">
        <visual>
            <origin xyz="0 0 0.5" />
            <geometry>
                <box size="0.3 0.3 1" />
            </geometry>
            <material name="pedestal_color">
                <color rgba="0.75 0.75 0.75 1" />
            </material>
        </visual>
        <collision>
            <origin xyz="0 0 0.5" />
            <geometry>
                <box size="0.3 0.3 1" />
            </geometry>
        </collision>

<inertial>
            <mass value="500" />
            <origin rpy="0 0 0" xyz="0 0 0" />
            <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001" iyz="0" izz="0.001" />
        </inertial>

    </link>
    <!-- measured from model -->
    <link name="robot1_base_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/base.dae
" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
```

```
<inertial>
            <mass value="2.0" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.0030531654454" ixy="0.0" ixz="0.0" iyy="0.0030531654454
" iyz="0.0" izz="0.005625" />
        </inertial>

    </link>
    <joint name="robot1_shoulder_pan_joint" type="revolute">
        <parent link="robot1_base_link" />
        <child link="robot1_shoulder_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.1519" />
        <axis xyz="0 0 1" />
        <limit effort="330.0" lower="-3.14159265359" upper="3.14159265359" velocity
="2.16" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot1_shoulder_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/shoulder
.dae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/shoul
der.stl" />
            </geometry>
        </collision>

<inertial>
            <mass value="2.0" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.0080931634294" ixy="0.0" ixz="0.0" iyy="0.0080931634294
" iyz="0.0" izz="0.005625" />
```

```
        </inertial>


    </link>
    <joint name="robot1_shoulder_lift_joint" type="revolute">
        <parent link="robot1_shoulder_link" />
        <child link="robot1_upper_arm_link" />
        <origin rpy="0.0 1.57079632679 0.0" xyz="0.0 0.1198 0.0" />
        <axis xyz="0 1 0" />
        <limit effort="330.0" lower="-3.14159265359" upper="3.14159265359" velocity
="2.16" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot1_upper_arm_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/upperarm
.dae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/upper
arm.stl" />
            </geometry>
        </collision>


<inertial>
            <mass value="3.42" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.121825" />
            <inertia ixx="0.0217284832211" ixy="0.0" ixz="0.0" iyy="0.0217284832211
" iyz="0.0" izz="0.00961875" />
        </inertial>


    </link>
    <joint name="robot1_elbow_joint" type="revolute">
        <parent link="robot1_upper_arm_link" />
```

```
        <child link="robot1_forearm_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 -0.0925 0.24365" />
        <axis xyz="0 1 0" />
        <limit effort="150.0" lower="-3.14159265359" upper="3.14159265359" velocity
="3.15" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot1_forearm_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/forearm.
dae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/forea
rm.stl" />
            </geometry>
        </collision>

<inertial>
            <mass value="1.26" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.106625" />
            <inertia ixx="0.00654680644378" ixy="0.0" ixz="0.0" iyy="0.006546806443
78" iyz="0.0" izz="0.00354375" />
        </inertial>

    </link>
    <joint name="robot1_wrist_1_joint" type="revolute">
        <parent link="robot1_forearm_link" />
        <child link="robot1_wrist_1_link" />
        <origin rpy="0.0 1.57079632679 0.0" xyz="0.0 0.0 0.21325" />
        <axis xyz="0 1 0" />
        <limit effort="54.0" lower="-3.14159265359" upper="3.14159265359" velocity=
"3.2" />
        <dynamics damping="0.5" friction="0.0" />
```

```
    </joint>
    <link name="robot1_wrist_1_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/wrist1.d
ae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/wrist
1.stl" />
            </geometry>
        </collision>


<inertial>
            <mass value="0.8" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.002084999166" ixy="0.0" ixz="0.0" iyy="0.002084999166" i
yz="0.0" izz="0.00225" />
        </inertial>


    </link>
    <joint name="robot1_wrist_2_joint" type="revolute">
        <parent link="robot1_wrist_1_link" />
        <child link="robot1_wrist_2_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.08505 0.0" />
        <axis xyz="0 0 1" />
        <limit effort="54.0" lower="-3.14159265359" upper="3.14159265359" velocity=
"3.2" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot1_wrist_2_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/wrist2.d
ae" />
```

```
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/wrist
2.stl" />
            </geometry>
        </collision>

<inertial>
            <mass value="0.8" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.002084999166" ixy="0.0" ixz="0.0" iyy="0.002084999166" i
yz="0.0" izz="0.00225" />
        </inertial>

    </link>
    <joint name="robot1_wrist_3_joint" type="revolute">
        <parent link="robot1_wrist_2_link" />
        <child link="robot1_wrist_3_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.08535" />
        <axis xyz="0 1 0" />
        <limit effort="54.0" lower="-3.14159265359" upper="3.14159265359" velocity=
"3.2" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot1_wrist_3_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/wrist3.d
ae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
```

```
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/wrist
3.stl" />
            </geometry>
        </collision>

<inertial>
            <mass value="0.35" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.000912187135125" ixy="0.0" ixz="0.0" iyy="0.00091218713
5125" iyz="0.0" izz="0.000984375" />
        </inertial>

    </link>
    <joint name="robot1_ee_fixed_joint" type="fixed">
        <parent link="robot1_wrist_3_link" />
        <child link="robot1_ee_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0819 0.0" />
    </joint>
    <link name="robot1_ee_link">
        <collision>
            <geometry>
                <box size="0.01 0.01 0.01" />
            </geometry>
            <origin rpy="0 0 0" xyz="-0.01 0 0" />
        </collision>

<inertial>
            <mass value="0.1" />
            <inertia ixx="0.03" iyy="0.03" izz="0.03" ixy="0.0" ixz="0.0" iyz="0.0"
/>
        </inertial>

    </link>
    <transmission name="robot1_shoulder_pan_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_shoulder_pan_joint">
```

```
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot1_shoulder_pan_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot1_shoulder_lift_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_shoulder_lift_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot1_shoulder_lift_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot1_elbow_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_elbow_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot1_elbow_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot1_wrist_1_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_wrist_1_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot1_wrist_1_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot1_wrist_2_trans">
        <type>transmission_interface/SimpleTransmission</type>
```

```
        <joint name="robot1_wrist_2_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot1_wrist_2_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot1_wrist_3_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_wrist_3_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot1_wrist_3_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <gazebo reference="robot1_shoulder_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot1_upper_arm_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot1_forearm_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot1_wrist_1_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot1_wrist_3_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
```

```
    <gazebo reference="robot1_wrist_2_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot1_ee_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <!-- ROS base_link to UR 'Base' Coordinates transform -->
    <link name="robot1_base1" />
    <joint name="robot1_base_link-base_fixed_joint" type="fixed">
        <!-- NOTE: this rotation is only needed as long as base_link itself is
                   not corrected wrt the real robot (ie: rotated over 180
                   degrees)
      -->
        <origin rpy="0 0 -3.14159265359" xyz="0 0 0" />
        <parent link="robot1_base_link" />
        <child link="robot1_base1" />
    </joint>
    <!-- Frame coincident with all-zeros TCP on UR controller -->
    <link name="robot1_tool0" />
    <joint name="robot1_wrist_3_link-tool0_fixed_joint" type="fixed">
        <origin rpy="-1.57079632679 0 0" xyz="0 0.0819 0" />
        <parent link="robot1_wrist_3_link" />
        <child link="robot1_tool0" />
    </joint>
    <gazebo reference="robot1_wrist_3_link-tool0_fixed_joint">
        <disableFixedJointLumping>true</disableFixedJointLumping>
    </gazebo>
    <link name="gripper1_robotiq_arg2f_base_link">


<inertial>
            <origin rpy="0 0 0" xyz="8.625E-08 -4.6583E-06 0.03145" />
            <mass value="0.22652" />
            <inertia ixx="0.00020005" ixy="-4.2442E-10" ixz="-2.9069E-10" iyy="0.00
017832" iyz="-3.4402E-08" izz="0.00013478" />
        </inertial>
```

```
        <visual>
            <origin rpy="0 0 0" xyz="0 0 0" />
            <geometry>
                <mesh filename="package://robotiq_2f_85_gripper_visualization/meshes/visual/robotiq_arg2f_85_base_link.dae" scale="0.001 0.001 0.001" />
            </geometry>
            <material name="">
                <color rgba="0.1 0.1 0.1 1" />
            </material>
        </visual>
        <collision>
            <origin rpy="0 0 0" xyz="0 0 0" />
            <geometry>
                <mesh filename="package://robotiq_2f_85_gripper_visualization/meshes/collision/robotiq_arg2f_base_link.stl" />
            </geometry>
        </collision>
    </link>
    <!-- measured from model -->
    <link name="robot2_base_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/base.dae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>


<inertial>
            <mass value="2.0" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.0030531654454" ixy="0.0" ixz="0.0" iyy="0.0030531654454" iyz="0.0" izz="0.005625" />
        </inertial>


    </link>
    <joint name="robot2_shoulder_pan_joint" type="revolute">
```

```
        <parent link="robot2_base_link" />
        <child link="robot2_shoulder_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.1519" />
        <axis xyz="0 0 1" />
        <limit effort="330.0" lower="-3.14159265359" upper="3.14159265359" velocity
="2.16" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot2_shoulder_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/shoulder
.dae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/shoul
der.stl" />
            </geometry>
        </collision>

<inertial>
            <mass value="2.0" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.0080931634294" ixy="0.0" ixz="0.0" iyy="0.0080931634294
" iyz="0.0" izz="0.005625" />
        </inertial>

    </link>
    <joint name="robot2_shoulder_lift_joint" type="revolute">
        <parent link="robot2_shoulder_link" />
        <child link="robot2_upper_arm_link" />
        <origin rpy="0.0 1.57079632679 0.0" xyz="0.0 0.1198 0.0" />
        <axis xyz="0 1 0" />
        <limit effort="330.0" lower="-3.14159265359" upper="3.14159265359" velocity
="2.16" />
```

```
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot2_upper_arm_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/upperarm
.dae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/upper
arm.stl" />
            </geometry>
        </collision>


<inertial>
            <mass value="3.42" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.121825" />
            <inertia ixx="0.0217284832211" ixy="0.0" ixz="0.0" iyy="0.0217284832211
" iyz="0.0" izz="0.00961875" />
        </inertial>


    </link>
    <joint name="robot2_elbow_joint" type="revolute">
        <parent link="robot2_upper_arm_link" />
        <child link="robot2_forearm_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 -0.0925 0.24365" />
        <axis xyz="0 1 0" />
        <limit effort="150.0" lower="-3.14159265359" upper="3.14159265359" velocity
="3.15" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot2_forearm_link">
        <visual>
            <geometry>
```

```
                <mesh filename="package://ur_description/meshes/ur3/visual/forearm.
dae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/forea
rm.stl" />
            </geometry>
        </collision>

<inertial>
            <mass value="1.26" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.106625" />
            <inertia ixx="0.00654680644378" ixy="0.0" ixz="0.0" iyy="0.006546806443
78" iyz="0.0" izz="0.00354375" />
        </inertial>

    </link>
    <joint name="robot2_wrist_1_joint" type="revolute">
        <parent link="robot2_forearm_link" />
        <child link="robot2_wrist_1_link" />
        <origin rpy="0.0 1.57079632679 0.0" xyz="0.0 0.0 0.21325" />
        <axis xyz="0 1 0" />
        <limit effort="54.0" lower="-3.14159265359" upper="3.14159265359" velocity=
"3.2" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot2_wrist_1_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/wrist1.d
ae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
```

```
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/wrist
1.stl" />
            </geometry>
        </collision>

<inertial>
            <mass value="0.8" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.002084999166" ixy="0.0" ixz="0.0" iyy="0.002084999166" i
yz="0.0" izz="0.00225" />
        </inertial>

    </link>
    <joint name="robot2_wrist_2_joint" type="revolute">
        <parent link="robot2_wrist_1_link" />
        <child link="robot2_wrist_2_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.08505 0.0" />
        <axis xyz="0 0 1" />
        <limit effort="54.0" lower="-3.14159265359" upper="3.14159265359" velocity=
"3.2" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot2_wrist_2_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/wrist2.d
ae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/wrist
2.stl" />
            </geometry>
```

```
        </collision>


<inertial>
            <mass value="0.8" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
            <inertia ixx="0.002084999166" ixy="0.0" ixz="0.0" iyy="0.002084999166" i
yz="0.0" izz="0.00225" />
        </inertial>


    </link>
    <joint name="robot2_wrist_3_joint" type="revolute">
        <parent link="robot2_wrist_2_link" />
        <child link="robot2_wrist_3_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.08535" />
        <axis xyz="0 1 0" />
        <limit effort="54.0" lower="-3.14159265359" upper="3.14159265359" velocity=
"3.2" />
        <dynamics damping="0.5" friction="0.0" />
    </joint>
    <link name="robot2_wrist_3_link">
        <visual>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/visual/wrist3.d
ae" />
            </geometry>
            <material name="LightGrey">
                <color rgba="0.7 0.7 0.7 1.0" />
            </material>
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://ur_description/meshes/ur3/collision/wrist
3.stl" />
            </geometry>
        </collision>


<inertial>
            <mass value="0.35" />
            <origin rpy="0 0 0" xyz="0.0 0.0 0.0" />
```

```
            <inertia ixx="0.000912187135125" ixy="0.0" ixz="0.0" iyy="0.00091218713
5125" iyz="0.0" izz="0.000984375" />
        </inertial>


    </link>
    <joint name="robot2_ee_fixed_joint" type="fixed">
        <parent link="robot2_wrist_3_link" />
        <child link="robot2_ee_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0819 0.0" />
    </joint>
    <link name="robot2_ee_link">
        <collision>
            <geometry>
                <box size="0.01 0.01 0.01" />
            </geometry>
            <origin rpy="0 0 0" xyz="-0.01 0 0" />
        </collision>


<inertial>
            <mass value="0.1" />
            <inertia ixx="0.03" iyy="0.03" izz="0.03" ixy="0.0" ixz="0.0" iyz="0.0"
/>
        </inertial>


    </link>
    <transmission name="robot2_shoulder_pan_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_shoulder_pan_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot2_shoulder_pan_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot2_shoulder_lift_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_shoulder_lift_joint">
```

```
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot2_shoulder_lift_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot2_elbow_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_elbow_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot2_elbow_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot2_wrist_1_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_wrist_1_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot2_wrist_1_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot2_wrist_2_trans">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_wrist_2_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot2_wrist_2_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="robot2_wrist_3_trans">
        <type>transmission_interface/SimpleTransmission</type>
```

```
        <joint name="robot2_wrist_3_joint">
            <hardwareInterface>hardware_interface/PositionJointInterface</hardwareI
nterface>
        </joint>
        <actuator name="robot2_wrist_3_motor">
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <gazebo reference="robot2_shoulder_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot2_upper_arm_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot2_forearm_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot2_wrist_1_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot2_wrist_3_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot2_wrist_2_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <gazebo reference="robot2_ee_link">
        <selfCollide>true</selfCollide>
        <implicitSpringDamper>1</implicitSpringDamper>
    </gazebo>
    <!-- ROS base_link to UR 'Base' Coordinates transform -->
    <link name="robot2_base" />
```

```
    <joint name="robot2_base_link-base_fixed_joint" type="fixed">
        <!-- NOTE: this rotation is only needed as long as base_link itself is
                   not corrected wrt the real robot (ie: rotated over 180
                   degrees)
      -->
        <origin rpy="0 0 -3.14159265359" xyz="0 0 0" />
        <parent link="robot2_base_link" />
        <child link="robot2_base" />
    </joint>
    <!-- Frame coincident with all-zeros TCP on UR controller -->
    <link name="robot2_tool0" />
    <joint name="robot2_wrist_3_link-tool0_fixed_joint" type="fixed">
        <origin rpy="-1.57079632679 0 0" xyz="0 0.0819 0" />
        <parent link="robot2_wrist_3_link" />
        <child link="robot2_tool0" />
    </joint>
    <gazebo reference="robot2_wrist_3_link-tool0_fixed_joint">
        <disableFixedJointLumping>true</disableFixedJointLumping>
    </gazebo>
    <link name="gripper2_robotiq_arg2f_base_link">


<inertial>
            <origin rpy="0 0 0" xyz="8.625E-08 -4.6583E-06 0.03145" />
            <mass value="0.22652" />
            <inertia ixx="0.00020005" ixy="-4.2442E-10" ixz="-2.9069E-10" iyy="0.00
017832" iyz="-3.4402E-08" izz="0.00013478" />
        </inertial>


        <visual>
            <origin rpy="0 0 0" xyz="0 0 0" />
            <geometry>
                <mesh filename="package://robotiq_2f_85_gripper_visualization/meshe
s/visual/robotiq_arg2f_85_base_link.dae" scale="0.001 0.001 0.001" />
            </geometry>
            <material name="">
                <color rgba="0.1 0.1 0.1 1" />
            </material>
        </visual>
```

```
        <collision>
            <origin rpy="0 0 0" xyz="0 0 0" />
            <geometry>
                <mesh filename="package://robotiq_2f_85_gripper_visualization/meshes/collision/robotiq_arg2f_base_link.stl" />
            </geometry>
        </collision>
    </link>
    <!-- .dae quirofano -->
    <link name="quirofano">
        <visual>
            <origin rpy="1.58 0 0" xyz="0 0 -1.0" />
            <geometry>
                <mesh filename="package://simulaciones/meshes/quirofano.dae" />
                <scale>0.4 0.4 0.4</scale>
            </geometry>
        </visual>
        <collision>
            <origin rpy="0 0 0" xyz="0 0.1 -0.75" />
            <geometry>
                <box size="2 0.5 0.75" />
            </geometry>
        </collision>
        <collision>
            <origin rpy="0 0 0" xyz="1.1 -0.55 -0.55" />
            <geometry>
                <box size="0.4 0.7 1" />
            </geometry>
        </collision>
        <collision>
            <origin rpy="0 0 0" xyz="-0.85 -0.55 -0.6" />
            <geometry>
                <box size="0.4 0.6 0.75" />
            </geometry>
        </collision>


<inertial>
            <mass value="0.1" />
```

```
            <inertia ixx="0.03" iyy="0.03" izz="0.03" ixy="0.0" ixz="0.0" iyz="0.0"
/>
        </inertial>


    </link>
    <!--camara -->
    <link name="camara">
        <visual>
            <origin rpy="0 0 0" xyz="0 0 0.025" />
            <geometry>
                <box size="0.05 0.2 0.05" />
            </geometry>
            <material name="dark_grey">
                <color rgba="0.1 0.1 0.1 1.0" />
            </material>
        </visual>
        <collision>
            <origin rpy="0 0 0" xyz="0 0 0.025" />
            <geometry>
                <box size="0.05 0.2 0.05" />
            </geometry>
            <material name="dark_grey" />
        </collision>


<inertial>
            <origin rpy="0 0 0" xyz="0 0 0" />
            <inertia ixx="0.1" ixy="0" ixz="0" iyy="0.1" iyz="0" izz="0.1" />
            <mass value="1" />
        </inertial>


    </link>
    <!-- Joints -->
    <!-- pedestal1 to mundo. -->
    <joint name="world_to_pedestal1" type="fixed">
        <parent link="world" />
        <child link="pedestal1_link" />
        <origin rpy="0.0 0.0 0.0" xyz="-0.35 -0.4 0.0" />
    </joint>
```

```
    <!-- robot1 to pedestal1. -->
    <joint name="pedestal1_to_robot1" type="fixed">
        <parent link="pedestal1_link" />
        <child link="robot1_base_link" />
        <origin rpy="0 0 0" xyz="0 0 0.7" />
    </joint>
    <!-- gripper1 to robot1. -->
    <joint name="gripper1_to_robot1" type="fixed">
        <parent link="robot1_tool0" />
        <child link="gripper1_robotiq_arg2f_base_link" />
    </joint>
    <!-- pedestal2 to mundo. -->
    <joint name="world_to_pedestal2" type="fixed">
        <parent link="world" />
        <child link="pedestal2_link" />
        <origin rpy="0.0 0.0 0.0" xyz="0.4 -0.4 0.0" />
    </joint>
    <!-- robot2 to pedestal1. -->
    <joint name="pedestal2_to_robot1" type="fixed">
        <parent link="pedestal2_link" />
        <child link="robot2_base_link" />
        <origin rpy="0 0 0" xyz="0 0 1" />
    </joint>
    <!-- gripper1 to robot1. -->
    <joint name="gripper2_to_robot1" type="fixed">
        <parent link="robot2_tool0" />
        <child link="gripper2_robotiq_arg2f_base_link" />
    </joint>
    <!-- quirofano to world. -->
    <joint name="quirofano_to_world" type="fixed">
        <parent link="world" />
        <child link="quirofano" />
        <origin rpy="0 0 0" xyz="0.0 0.0 1" />
    </joint>
    <!-- camara to world. -->
    <joint name="world_to_camera_joint" type="fixed">
        <parent link="world" />
        <child link="camara" />
```

```
        <origin rpy="0 0 0" xyz="-0.3 0.1 1.35" />
    </joint>
    <transmission name="trans_robot1_shoulder_pan_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_shoulder_pan_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot1_shoulder_pan_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot1_shoulder_lift_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_shoulder_lift_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot1_shoulder_lift_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot1_elbow_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_elbow_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot1_elbow_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot1_wrist_1_joint">
        <type>transmission_interface/SimpleTransmission</type>
```

```
        <joint name="robot1_wrist_1_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot1_wrist_1_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot1_wrist_2_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_wrist_2_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot1_wrist_2_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot1_wrist_3_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot1_wrist_3_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot1_wrist_3_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot2_shoulder_pan_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_shoulder_pan_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot2_shoulder_pan_joint_motor">
```

```
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot2_shoulder_lift_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_shoulder_lift_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot2_shoulder_lift_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot2_elbow_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_elbow_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot2_elbow_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot2_wrist_1_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_wrist_1_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot2_wrist_1_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
```

```
    <transmission name="trans_robot2_wrist_2_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_wrist_2_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot2_wrist_2_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <transmission name="trans_robot2_wrist_3_joint">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="robot2_wrist_3_joint">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
        </joint>
        <actuator name="robot2_wrist_3_joint_motor">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInt
erface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
    <gazebo>
        <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
            <robotNamespace>/</robotNamespace>
        </plugin>
    </gazebo>
</robot>
```

8.4. Anexo 4: Programa move.py

```
#!/usr/bin/env python
import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list


def all_close(goal, actual, tolerance):

  all_equal = True
  if type(goal) is list:
    for index in range(len(goal)):
      if abs(actual[index] - goal[index]) > tolerance:
        return False

  elif type(goal) is geometry_msgs.msg.PoseStamped:
    return all_close(goal.pose, actual.pose, tolerance)

  elif type(goal) is geometry_msgs.msg.Pose:
    return all_close(pose_to_list(goal), pose_to_list(actual), tolerance)

  return True



class MoveGroupPythonInteface(object):

  def __init__(self):
    super(MoveGroupPythonInteface, self).__init__()

    ## Inicializar`moveit_commander`_ y el nodo rospy:
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('move', anonymous=True)
```

```
    robot = moveit_commander.RobotCommander()
    scene = moveit_commander.PlanningSceneInterface()

    group_name1 = "robot1"
    move_group1 = moveit_commander.MoveGroupCommander(group_name1)
    group_name2 = "robot2"
    move_group2 = moveit_commander.MoveGroupCommander(group_name2)

    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_pat
h', moveit_msgs.msg.DisplayTrajectory, queue_size=20)

    planning_frame1 = move_group1.get_planning_frame()
    planning_frame2 = move_group2.get_planning_frame()

    eef_link1 = move_group1.get_end_effector_link()
    eef_link2 = move_group2.get_end_effector_link()

    group_names = robot.get_group_names()

    # Variables de la clase
    self.camara_name = ''
    self.instru1_name = ''
    self.instru2_name = ''
    self.instru3_name = ''
    self._name = ''
    self.robot = robot
    self.scene = scene
    self.move_group1 = move_group1
    self.move_group2 = move_group2
    self.display_trajectory_publisher = display_trajectory_publisher
    self.planning_frame1 = planning_frame1
    self.planning_frame2 = planning_frame2
    self.eef_link1 = eef_link1
    self.eef_link2 = eef_link2
    self.group_names = group_names



  def go_to_Home_Robot1(self):
```

```
    robot = self.move_group1
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = 0
    joint_goal[1] = -1.59
    joint_goal[2] = 0
    joint_goal[3] = -1.59
    joint_goal[4] = 0
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Mesa_Robot1(self):

    robot = self.move_group1
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = -3.14
    joint_goal[1] = -0.45
    joint_goal[2] = 0
    joint_goal[3] = 0.45
    joint_goal[4] = 1.59
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    rospy.sleep(0.5)
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Coger_Robot1(self):

    robot = self.move_group1
    joint_goal = robot.get_current_joint_values()
```

```
    joint_goal[0] = -3.14
    joint_goal[1] = -0.4
    joint_goal[2] = 0.5
    joint_goal[3] = -0.1
    joint_goal[4] = 1.59
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Colocar_Robot1(self):

    robot = self.move_group1
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = -2
    joint_goal[1] = -2.5
    joint_goal[2] = -0.3
    joint_goal[3] = 3
    joint_goal[4] = -1.8
    joint_goal[5] = 3.1
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Fin_Robot1(self):

    robot = self.move_group1
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = -1.55
    joint_goal[1] = -2.3
    joint_goal[2] = -0.7
    joint_goal[3] = 3
```

```
    joint_goal[4] = -1.8
    joint_goal[5] = 2.3
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Home_Robot2(self):

    robot = self.move_group2
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = 0
    joint_goal[1] = -1.59
    joint_goal[2] = 0
    joint_goal[3] = -1.59
    joint_goal[4] = 0
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos el robot para no generar residuos
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Mesa_Robot2_pose3(self):

    robot = self.move_group2
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = 0
    joint_goal[1] = -0.15
    joint_goal[2] = 0.6
    joint_goal[3] = -0.4
    joint_goal[4] = 1.59
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
```

```
    # Paramos los robot para no generar residuos
    rospy.sleep(0.5)
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Mesa_Robot2_pose2(self):

    robot = self.move_group2
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = -0.5
    joint_goal[1] = -0.15
    joint_goal[2] = 0.6
    joint_goal[3] = -0.4
    joint_goal[4] = 1
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    rospy.sleep(0.5)
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)

  def go_to_Mesa_Robot2_pose1(self):

    robot = self.move_group2
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = -0.85
    joint_goal[1] = -0.15
    joint_goal[2] = 0.7
    joint_goal[3] = -0.6
    joint_goal[4] = 0.7
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    rospy.sleep(0.5)
```

```
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)


  def go_to_Medico_Robot2(self):


    robot = self.move_group2
    joint_goal = robot.get_current_joint_values()
    joint_goal[0] = 1.6
    joint_goal[1] = -0.15
    joint_goal[2] = 0
    joint_goal[3] = 0
    joint_goal[4] = 1.59
    joint_goal[5] = -1.59
    # Esperamos a que el robot llege a la trayectoria expecificada
    robot.go(joint_goal, wait=True)
    # Paramos los robot para no generar residuos
    rospy.sleep(0.5)
    robot.stop()
    current_joints = robot.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)


  def wait_for_state_update_camara(self, camara_is_known=False, camara_is_attached=
False, timeout=4):


    camara_name = self.camara_name
    scene = self.scene
    start = rospy.get_time()
    seconds = rospy.get_time()
    while (seconds - start < timeout) and not rospy.is_shutdown():


      attached_objects = scene.get_attached_objects([camara_name])
      is_attached = len(attached_objects.keys()) > 0
      is_known = camara_name in scene.get_known_object_names()


      if (camara_is_attached == is_attached) and (camara_is_known == is_known):
        return True
```

```
      rospy.sleep(1)
      seconds = rospy.get_time()

    return False

  def wait_for_state_update_instru1(self, instru1_is_known=False, instru1_is_attach
ed=False, timeout=4):

    instru1_name = self.instru1_name
    scene = self.scene
    start = rospy.get_time()
    seconds = rospy.get_time()
    while (seconds - start < timeout) and not rospy.is_shutdown():

      attached_objects = scene.get_attached_objects([instru1_name])
      is_attached = len(attached_objects.keys()) > 0
      is_known = instru1_name in scene.get_known_object_names()

      if (instru1_is_attached == is_attached) and (instru1_is_known == is_known):
        return True

      rospy.sleep(1)
      seconds = rospy.get_time()

    return False

  def wait_for_state_update_instru2(self, instru2_is_known=False, instru2_is_attach
ed=False, timeout=4):

    instru2_name = self.instru2_name
    scene = self.scene
    start = rospy.get_time()
    seconds = rospy.get_time()
    while (seconds - start < timeout) and not rospy.is_shutdown():

      attached_objects = scene.get_attached_objects([instru2_name])
      is_attached = len(attached_objects.keys()) > 0
      is_known = instru2_name in scene.get_known_object_names()
```

```
      if (instru2_is_attached == is_attached) and (instru2_is_known == is_known):
        return True

      rospy.sleep(1)
      seconds = rospy.get_time()

    return False

  def wait_for_state_update_instru3(self, instru3_is_known=False, instru3_is_attach
ed=False, timeout=4):

    instru3_name = self.instru3_name
    scene = self.scene
    start = rospy.get_time()
    seconds = rospy.get_time()
    while (seconds - start < timeout) and not rospy.is_shutdown():

      attached_objects = scene.get_attached_objects([instru3_name])
      is_attached = len(attached_objects.keys()) > 0
      is_known = instru3_name in scene.get_known_object_names()

      if (instru3_is_attached == is_attached) and (instru3_is_known == is_known):
        return True

      rospy.sleep(1)
      seconds = rospy.get_time()

    return False

  def add_camara(self, timeout=4):

    camara_name = self.camara_name
    scene = self.scene

    camara_pose = geometry_msgs.msg.PoseStamped()
    camara_pose.header.frame_id = "gripper1_robotiq_arg2f_base_link"
    camara_pose.pose.orientation.w = 1.0
```

```
    camara_pose.pose.position.z = 0.08
    camara_pose.pose.position.x = -0.08
    camara_name = "camara"
    scene.add_box(camara_name, camara_pose, size=(0.2, 0.01, 0.01))

    self.camara_name=camara_name
    return self.wait_for_state_update_camara(camara_is_known=True, timeout=timeout)


  def attach_camara(self, timeout=4):

    camara_name = self.camara_name
    robot = self.robot
    scene = self.scene
    eef_link = self.eef_link1
    group_names = self.group_names

    grasping_group = 'robot1'
    touch_links = robot.get_link_names(group=grasping_group)
    scene.attach_box(eef_link, camara_name, touch_links=touch_links)
    rospy.sleep(1)

  def detach_camara(self, timeout=4):

    camara_name = self.camara_name
    scene = self.scene
    eef_link = self.eef_link1

    scene.remove_attached_object(eef_link, name=camara_name)
    return self.wait_for_state_update_camara(camara_is_known=True, camara_is_attach
ed=False, timeout=timeout)

  def remove_camara(self, timeout=4):
    camara_name = self.camara_name
    scene = self.scene
    scene.remove_world_object(camara_name)
```

```
    return self.wait_for_state_update_camara(camara_is_attached=False, camara_is_kn
own=False, timeout=timeout)


  def add_instru1(self, timeout=4):


    instru1_name = self.instru1_name

    scene = self.scene


    instru1_pose = geometry_msgs.msg.PoseStamped()

    instru1_pose.header.frame_id = "gripper2_robotiq_arg2f_base_link"

    instru1_pose.pose.orientation.w = 1.0

    instru1_pose.pose.position.z = 0.08

    instru1_pose.pose.position.x = -0.03

    instru1_name = "instru1"

    scene.add_box(instru1_name, instru1_pose, size=(0.07, 0.05, 0.05))


    self.instru1_name=instru1_name

    return self.wait_for_state_update_instru1(instru1_is_known=True, timeout=timeou
t)



  def attach_instru1(self, timeout=4):


    instru1_name = self.instru1_name

    robot = self.robot

    scene = self.scene

    eef_link = self.eef_link2

    group_names = self.group_names


    grasping_group = 'robot2'

    touch_links = robot.get_link_names(group=grasping_group)

    scene.attach_box(eef_link, instru1_name, touch_links=touch_links)

    rospy.sleep(1)


  def detach_instru1(self, timeout=4):


    instru1_name = self.instru1_name

    scene = self.scene
```

```
    eef_link = self.eef_link2


    scene.remove_attached_object(eef_link, name=instru1_name)
    return self.wait_for_state_update_instru1(instru1_is_known=True, instru1_is_att
ached=False, timeout=timeout)


  def remove_instru1(self, timeout=4):
    instru1_name = self.instru1_name
    scene = self.scene
    scene.remove_world_object(instru1_name)


    return self.wait_for_state_update_instru1(instru1_is_attached=False, instru1_is
_known=False, timeout=timeout)


  def add_instru2(self, timeout=4):


    instru2_name = self.instru2_name
    scene = self.scene


    instru2_pose = geometry_msgs.msg.PoseStamped()
    instru2_pose.header.frame_id = "gripper2_robotiq_arg2f_base_link"
    instru2_pose.pose.orientation.w = 1.0
    instru2_pose.pose.position.z = 0.08
    instru2_name = "instru2"
    scene.add_sphere(instru2_name, instru2_pose, radius=(0.02))


    self.instru2_name=instru2_name
    return self.wait_for_state_update_instru2(instru2_is_known=True, timeout=timeou
t)




  def attach_instru2(self, timeout=4):


    instru2_name = self.instru2_name
    robot = self.robot
    scene = self.scene
    eef_link = self.eef_link2
    group_names = self.group_names
```

```
    grasping_group = 'robot2'
    touch_links = robot.get_link_names(group=grasping_group)
    scene.attach_box(eef_link, instru2_name, touch_links=touch_links)
    rospy.sleep(1)


  def detach_instru2(self, timeout=4):
    instru2_name = self.instru2_name
    scene = self.scene
    eef_link = self.eef_link2


    scene.remove_attached_object(eef_link, name=instru2_name)
    return self.wait_for_state_update_instru2(instru2_is_known=True, instru2_is_att
ached=False, timeout=timeout)
  def remove_instru2(self, timeout=4):
    instru2_name = self.instru2_name
    scene = self.scene
    scene.remove_world_object(instru2_name)


    return self.wait_for_state_update_instru2(instru2_is_attached=False, instru2_is
_known=False, timeout=timeout)


  def add_instru3(self, timeout=4):


    instru3_name = self.instru3_name
    scene = self.scene


    instru3_pose = geometry_msgs.msg.PoseStamped()
    instru3_pose.header.frame_id = "gripper2_robotiq_arg2f_base_link"
    instru3_pose.pose.orientation.w = 1.0
    instru3_pose.pose.position.z = 0.08
    instru3_name = "instru3"
    scene.add_cylinder(instru3_name, instru3_pose, 0.08, 0.02)


    self.instru3_name=instru3_name
    return self.wait_for_state_update_instru3(instru3_is_known=True, timeout=timeou
t)




  def attach_instru3(self, timeout=4):
```

```
    instru3_name = self.instru3_name
    robot = self.robot
    scene = self.scene
    eef_link = self.eef_link2
    group_names = self.group_names

    grasping_group = 'robot2'
    touch_links = robot.get_link_names(group=grasping_group)
    scene.attach_box(eef_link, instru3_name, touch_links=touch_links)
    rospy.sleep(1)

  def detach_instru3(self, timeout=4):

    instru3_name = self.instru3_name
    scene = self.scene
    eef_link = self.eef_link2
    scene.remove_attached_object(eef_link, name=instru3_name)
    return self.wait_for_state_update_instru3(instru3_is_known=True, instru3_is_att
ached=False, timeout=timeout)

  def remove_instru3(self, timeout=4):
    instru3_name = self.instru3_name
    scene = self.scene
    scene.remove_world_object(instru3_name)

    return self.wait_for_state_update_instru3(instru3_is_attached=False, instru3_is
_known=False, timeout=timeout)
def main():
  try:
    # Inicializamos variables
    movement = MoveGroupPythonInteface()

    print "=== Presione `Enter` para ejecutar el programa ==="
    raw_input()
    movement.go_to_Home_Robot1()
    movement.go_to_Home_Robot2()
    movement.go_to_Mesa_Robot1()
```

```
    movement.go_to_Coger_Robot1()
    movement.add_camara()
    movement.attach_camara()
    movement.go_to_Mesa_Robot1()
    movement.go_to_Home_Robot1()
    movement.go_to_Colocar_Robot1()
    movement.go_to_Fin_Robot1()
    print "=== Elija material a coger: ==="
    print "=== Pulse 1 = Instrumento n 1 ==="
    print "=== Pulse 2 = Instrumento n 2 ==="
    print "=== Pulse 3 = Instrumento n 3 ==="
    print "=== Pulse 4 = Acabar ==="
    opcion=raw_input()

    while opcion!="4":

        if opcion=="1":
            movement.go_to_Home_Robot2()
            movement.go_to_Mesa_Robot2_pose1()
            movement.add_instru1()
            movement.attach_instru1()
            movement.go_to_Medico_Robot2()
            movement.detach_instru1()
            movement.remove_instru1()
            movement.go_to_Home_Robot2()
            print "=== Para devolver material pulse `Enter` ==="
            raw_input()
            movement.go_to_Medico_Robot2()
            movement.add_instru1()
            movement.attach_instru1()
            movement.go_to_Mesa_Robot2_pose1()
            movement.detach_instru1()
            movement.remove_instru1()
            movement.go_to_Home_Robot2()
            print "=== Elija material a coger: ==="
            print "=== Pulse 1 = Instrumento n 1 ==="
            print "=== Pulse 2 = Instrumento n 2 ==="
            print "=== Pulse 3 = Instrumento n 3 ==="
```

```
            print "=== Pulse 4 = Acabar ==="
            opcion=raw_input()


        elif opcion=="2":
            movement.go_to_Home_Robot2()
            movement.go_to_Mesa_Robot2_pose2()
            movement.add_instru2()
            movement.attach_instru2()
            movement.go_to_Medico_Robot2()
            movement.detach_instru2()
            movement.remove_instru2()
            movement.go_to_Home_Robot2()
            print "=== Para devolver material pulse `Enter` ==="
            raw_input()
            movement.go_to_Medico_Robot2()
            movement.add_instru2()
            movement.attach_instru2()
            movement.go_to_Mesa_Robot2_pose2()
            movement.detach_instru2()
            movement.remove_instru2()
            movement.go_to_Home_Robot2()
            print "=== Elija material a coger: ==="
            print "=== Pulse 1 = Instrumento n 1 ==="
            print "=== Pulse 2 = Instrumento n 2 ==="
            print "=== Pulse 3 = Instrumento n 3 ==="
            print "=== Pulse 4 = Acabar ==="
            opcion=raw_input()
        elif opcion=="3":
            movement.go_to_Home_Robot2()
            movement.go_to_Mesa_Robot2_pose3()
            movement.add_instru3()
            movement.attach_instru3()
            movement.go_to_Medico_Robot2()
            movement.detach_instru3()
            movement.remove_instru3()
            movement.go_to_Home_Robot2()
            print "=== Para devolver material pulse `Enter` ==="
            raw_input()
```

```
            movement.go_to_Medico_Robot2()
            movement.add_instru3()
            movement.attach_instru3()
            movement.go_to_Mesa_Robot2_pose2()
            movement.detach_instru3()
            movement.remove_instru3()
            movement.go_to_Home_Robot2()
            print "=== Elija material a coger: ==="
            print "=== Pulse 1 = Instrumento n 1 ==="
            print "=== Pulse 2 = Instrumento n 2 ==="
            print "=== Pulse 3 = Instrumento n 3 ==="
            print "=== Pulse 4 = Acabar ==="
            opcion=raw_input()

        else:
            opcion="4"
    movement.go_to_Colocar_Robot1()
    movement.go_to_Home_Robot1()
    movement.go_to_Mesa_Robot1()
    movement.go_to_Coger_Robot1()
    movement.detach_camara()
    movement.remove_camara()
    movement.go_to_Mesa_Robot1()
    movement.go_to_Home_Robot2()
    movement.go_to_Home_Robot1()
    print "==== Programa completado ===="
  except rospy.ROSInterruptException:
    return
  except KeyboardInterrupt:
    return

if __name__ == '__main__':
  main()
```

8.5. Presupuesto del proyecto

	E/h	1	1	1	1	1	50	60	45	50	40	40	40	35	60	40	40	40		
		Comerciales					Ingeniería			Fabricación						Motaje				
Concepto	Unidades	mat. comercial mec.	mat. comercial elec.	mat. comercial neum.	materia prima	herramientas/consumibles	proyectista	gestor proyecto	automatista/robotico	organización fabricacion	corte y preparación	soldador	pleg./curv./cizallas	pintura	fresas	mecánico	eléctrico	robotico	total unitario	TOTAL
		51.330	1.200	0	300	0	1000	300	0	200	320	800	0	280	720	320	0	4800		
Suministro	x	51.330	1.200	0	300	0	1000	300	0	200	320	800	0	280	720	320	0	4.800		61.570
Robot UR3e	2	22.950																	22.950	45.900
Garra 2F/85 Robotiq	2	2.715																	2.715	5.430
Pedestales	2				150		10			2	4	10		4	6				1.810	3.620
Equipo informatico	1		1.200													8			1.520	1.520
Montaje	2																		0	0
Programacion y pruebas	15																	8	320	4.800
Gestion de proyecto	1							5											300	300