+ ROS

Tinkerkit Braccio y uso del framework ROS para el aprendizaje y desarrollo de robots manipuladores

Augusto Hernández Elvira

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Desarrollo de aplicaciones electrónicas

Nombre Consultor/a: Aleix López Antón
Nombre Profesor/a responsable de la asignatura: Carlos Monzo Sánchez

07/06/2021

Logo de la portada:

Autor: *Willow Garage*
Fecha: 20/10/2018
Fuente: http://www.ros.org/press-kit/
Licencia: https://commons.wikimedia.org/wiki/File:Ros_logo.svg

Imagen de la portada: *Tinkerkit Braccio*

Fuente: https://store.arduino.cc/tinkerkit-braccio-robot

a mi esposa Claudia y mi hijo Hugo
zu meiner Frau Claudia und meinem Sohn Hugo

a mi “hermano” mayor Rafael

Hace mucho tiempo que escribí las siguientes palabras:

“Me dirijo seguro al mismo lugar de donde vine, consciente de que la piel, al arrugarse, se asemeja cada vez más a la de un niño antiguo y respetable. ¡Es sin duda alguna el amor reunido, el equipaje más valioso del ser humano!”.

Ich habe vor langer Zeit die folgenden Wörter geschrieben:

“Ich gehe sicher zu dem Ort, von dem ich gekommen bin, und weiß, dass die faltige Haut mehr und mehr der eines alten und anständigen Kindes ähnelt. Liebe ist ohne Zweifel das wertvollste Gepäck der Menschen! “.

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Tinkerkit Braccio y uso del framework ROS para el aprendizaje y desarrollo de robots manipuladores*
Nombre del autor:	Augusto Hernández Elvira
Nombre del consultor/a:	Aleix López Antón
Nombre del PRA:	Carlos Monzo Sánchez
Fecha de entrega (mm/aaaa):	06/2021
Titulación::	Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Área del Trabajo Final:	Desarrollo de aplicaciones electrónicas
Idioma del trabajo:	Español
Palabras clave	Arduino, ROS, robots

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

ROS se está convirtiendo, sin duda alguna, en un estándar de *facto* en el campo de la robótica. De forma paralela, la placa Arduino sigue siendo fuente de inspiración para los distintos proyectos de investigadores, estudiantes y *makers*. Resulta conveniente, por lo tanto, realizar un estudio sobre dicha plataforma de software y el *hardware* que nos proporciona Arduino junto con el robot manipulador de *Tinkerkit Braccio*.

Para ello, el proyecto plantea algunos aspectos fundamentales. Inicialmente se presenta el diseño propuesto a nivel de *hardware/software*, y se hace una introducción del ecosistema ROS. Luego se explica detalladamente toda la teoría cinemática necesaria para entender un trabajo de este tipo. Seguidamente se continúa profundizando en la cinemática (directa e inversa), pero esta vez a nivel más práctico, con ayuda de algunas herramientas y librerías de ROS como RViz y *MoveIt*! El hilo conductor entre los distintos capítulos será el elemento visual, haciendo lo más agradable e intuitivo posible, el recorrido conceptual del lector a lo largo del TFG. Finalmente se hará una prueba de funcionamiento y algunos movimientos básicos con el robot.

Abstract (in English, 250 words or less):

ROS is undoubtedly becoming a de facto standard in robotics. In parallel, the Arduino board continues to be a source of inspiration for the different projects of researchers, students and makers. Therefore, it is convenient to carry out a study on this software platform and the hardware provided by Arduino together with the Tinkerkit Braccio manipulator robot.

For this, the project raises some fundamental aspects. Initially, the proposed design at the hardware / software level is presented, and an introduction of the ROS ecosystem is made. Then, all the kinematic theory necessary to understand a work of this type is explained in detail. Then we continue to delve into kinematics (direct and inverse), but this time at a more practical level, with the help of some ROS tools and libraries such as RViz and MoveIt! The common thread between the different chapters will be the visual element, making the reader's conceptual journey throughout the Bachelor Thesis as pleasant and intuitive as possible. Finally, there will be a test run and some basic movements with the robot.

Índice

Lista de figuras

Lista de tablas

1. Introducción

1.1 Contexto y justificación del Trabajo

ROS es una plataforma fantástica para el desarrollo y programación de robots, pero desafortunadamente la pendiente de su curva de aprendizaje comienza de forma exponencial y se aplana al final, como se puede observar en la Figura 1. [1] Una curva de este tipo señala que al principio nos cuesta mucho trabajo aprender, pero una vez tengamos determinado conocimiento podemos avanzar más rápido. Sin embargo, para llegar a un nivel de experto nos vuelve a costar mucho tiempo y esfuerzo.

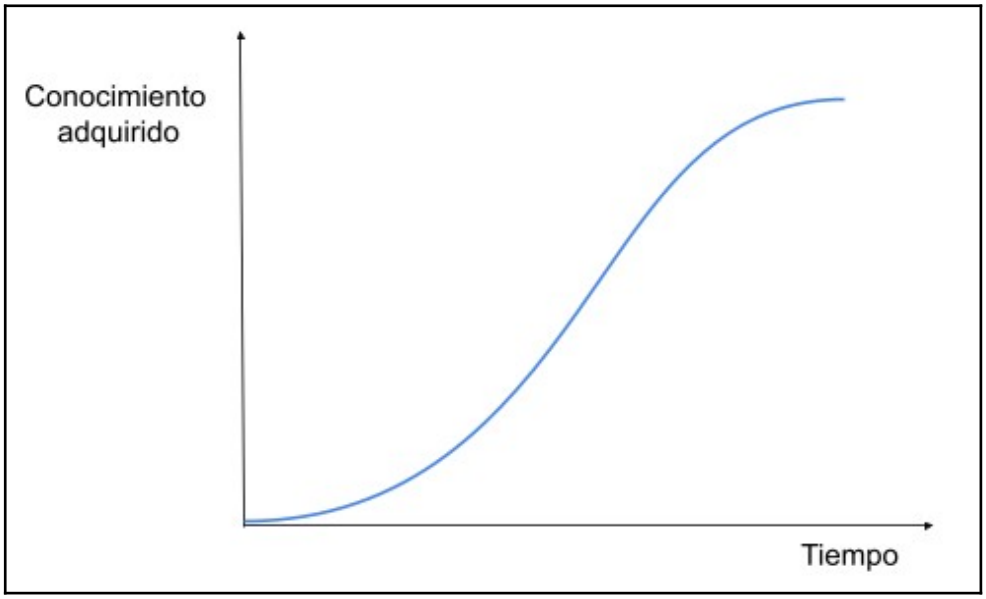


Figura 1. Curva de aprendizaje en ROS. Fuente: elaboración propia

El primer contacto con ROS puede resultar algo frustrante y tedioso, sobretodo en la parte de instalación y configuración de todo el sistema de archivos y dependencias. Además, ROS en su forma nativa tiene un alto porcentaje de *shell* (alto porcentaje de uso de terminales y línea de comandos), lo que puede suponer una barrera de entrada a un gran número de futuros usuarios. Este proyecto intentará acercar, el mundo de ROS, a todas aquellas personas interesadas en el aprendizaje y desarrollo de manipuladores robóticos, independientemente de los conocimientos previos que se tengan sobre el tema.

1.2 Objetivos del Trabajo

Los objetivos principales de este proyecto son los siguientes:

- Introducir los conocimientos básicos de cinemática directa e inversa.

- Estudio detallado del ecosistema de ROS (URDF, RViz, *MoveIt*!) y sus conceptos fundamentales.

- Realizar una demostración práctica con el robot manipulador *Braccio* de *Tinkerkit*. En este último objetivo, se hará uso del lenguaje de programación C++ y de la librería de álgebra *Eigen*.

1.3 Enfoque y método seguido

En este trabajo se sigue un enfoque didáctico, intentando que el lector aprenda lo máximo posible. El material pedagógico de partida serán los tutoriales de la *wiki* oficial de ROS y código libre de *Github*, pero debido a la falta de detalles técnicos, pasos lógicos, erratas, etc. también se recurrirá a métodos de ingeniería inversa para completar dicho contenido. Los elementos visuales y de expresión gráfica serán ampliamente desarrollados, tanto para una mayor comprensión de los aspectos cinemáticos, como para una ágil representación de algunos conceptos clave.

1.4 Planificación del Trabajo

La planificación del trabajo se ha realizado a través de entregables a nivel de producto:

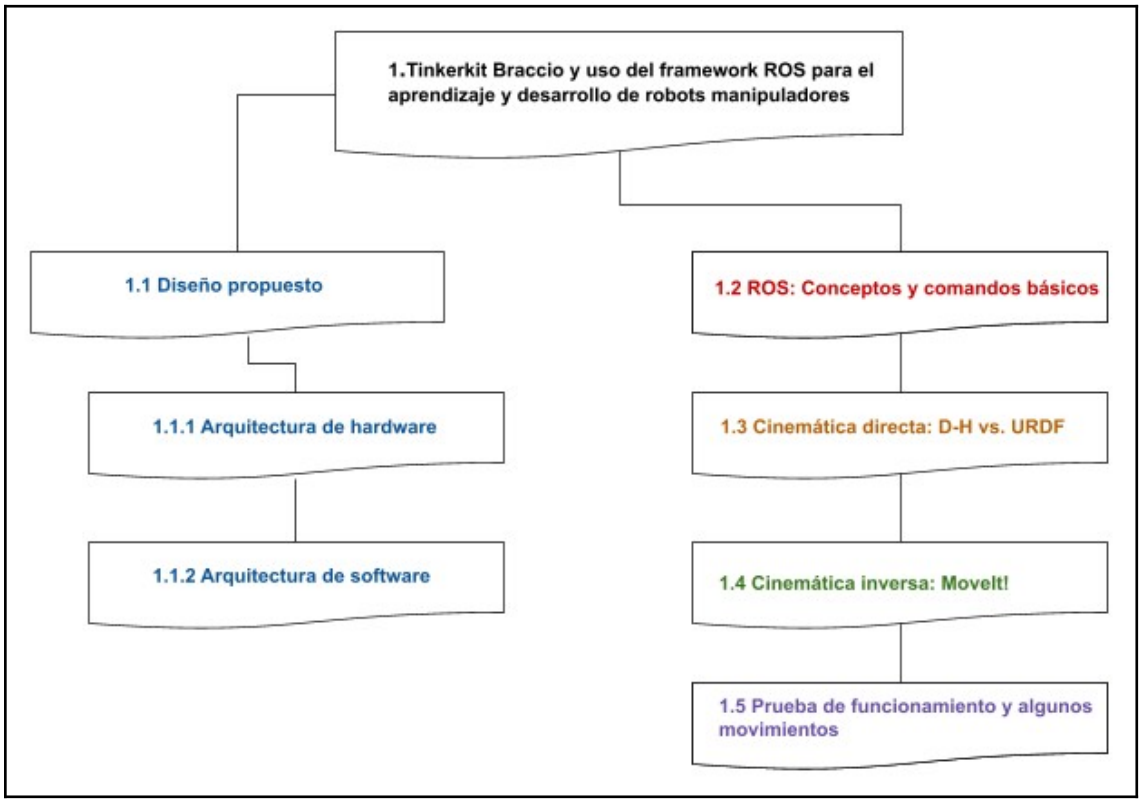


Figura 2. Entregables (wbs) a nivel de producto. Fuente: elaboración propia

1.5 Breve descripción de los otros capítulos de la memoria

- Capítulo 2. Estado del arte. Conocimiento general de la tecnología relacionada o implementada.
- Capítulo 3. Diseño propuesto. Introducción a la arquitectura de hardware y software propuesta.
- Capútulo 4. ROS: conceptos y comandos básicos. Breve descripción del sistema ROS a nivel conceptual y de comandos.
- Capítulo 5. Cinemática directa: D-H vs. URDF. Se analizan y comparan dos modelos distintos de cinemática directa.

- Capítulo 6. Cinemática inversa: MoveIt! Se configura la cinemática inversa del robot manipulador y se genera automáticamente todo el código necesario para poder implementarla.

- Capítulo 7. Prueba de funcionamiento y algunos movimientos. Finalmente se prueban en el robot real la cinemática directa e indirecta.

2. Estado del arte

Robot Operating System (ROS) es un *framework* de código abierto (*open source*) que se está convirtiendo en un estándar de *facto* en el campo de la robótica. ROS sigue una arquitectura distribuida, robusta, altamente modularizada, proporciona el software y las librerías necesarias que facilitan la abstracción del hardware y resulta de fácil expansión. No obstante, no es un sistema operativo como tal; ya que corre por encima del sistema operativo nativo *Ubuntu* (sistemas operativos basados en *Unix/Linux*, y ahora también por *Windows* 10, con la reciente versión ROS 2).

2.1 Distribuciones de ROS

Según la versión del sistema operativo *Ubuntu* que tengamos instalada en nuestro ordenador o sistema embebido, ROS tiene una o varias versiones compatibles que pueden utilizarse. En nuestro caso, como disponemos de un ordenador portátil *Toshiba Satellite* C70D-A con el sistema operativo *Ubuntu* 16.04, vamos a trabajar con la versión ROS Kinetic Kame, que es una versión LTS (*Long Term Suppor*t) lanzada en 2017, y que dispone de soporte hasta el año 2021.

Figura 3. ROS Kinetic Kame. Fuente: [2]

A continuación se introducen las distribuciones ROS más recomendadas:

- **ROS Noetic**:
  - Publicada: 23/05/2020
  - Plataformas:
    - Ubuntu: Focal
    - Debian: Scretch
  - Cambios:
    http://wiki.ros.org/noetic/Migration

- **ROS Melodic**:
  - Publicada: 23/05/2018
  - Plataformas:
    - Ubuntu: Artful, Bionic
    - Debian: Scretch
  - Cambios:
    http://wiki.ros.org/melodic/Migration

- **ROS Lunar**:
  - Publicada: 23/05/2017
  - Plataformas:
    - Ubuntu: Willy; Xenial
    - Debian: Scretch
  - Cambios:
    http://wiki.ros.org/lunar/Migration

- **ROS Kinetic**:
  - Publicada: 23/05/2016
  - Plataformas:
    - Ubuntu: Willy; Xenial
    - Debian: Jessie
    - OS X(Homebrew)
    - Gentoo
    - OpenEmbedded/Yocto
  - Cambios:
    http://wiki.ros.org/kinetic/Migration

- **ROS Jade**:
  - Publicada: 23/05/2015
  - Plataformas:
    - Ubuntu: Willy; Xenial
    - Debian: Jessie
    - OS X(Homebrew)
    - Gentoo
    - Android (NDK)
  - Cambios:
    http://wiki.ros.org/jade/Migration

- **ROS Indigo**:
  - Publicada: 22/04/2014
  - Plataformas:
    - Ubuntu: Willy; Xenial
    - Debian: Wheezy
    - OS X(Homebrew)
    - Gentoo
    - OpenEmbedded/Yocto
    - Android (NDK)
  - Cambios:
    http://wiki.ros.org/indigo/Migration

2.2 Robots manipuladores compatibles con ROS

El Tinkerkit Braccio de Arduino, por otra parte, contiene todas los elementos necesarios para crear un manipulador robótico programable usando un microcontrolador Arduino, los servomotores y la *shield* incluida en el *kit*. El Braccio es un robot de bajo coste (230 € aproximadamente) que está diseñado para proyectos de estudiantes, investigadores y *makers*. Se puede ensamblar de muchas formas, pero para este trabajo se utiliza la configuración que le da

al brazo robótico 5 grados de libertad (+1 GdL Pinza), tal y como puede apreciarse en la siguiente figura:

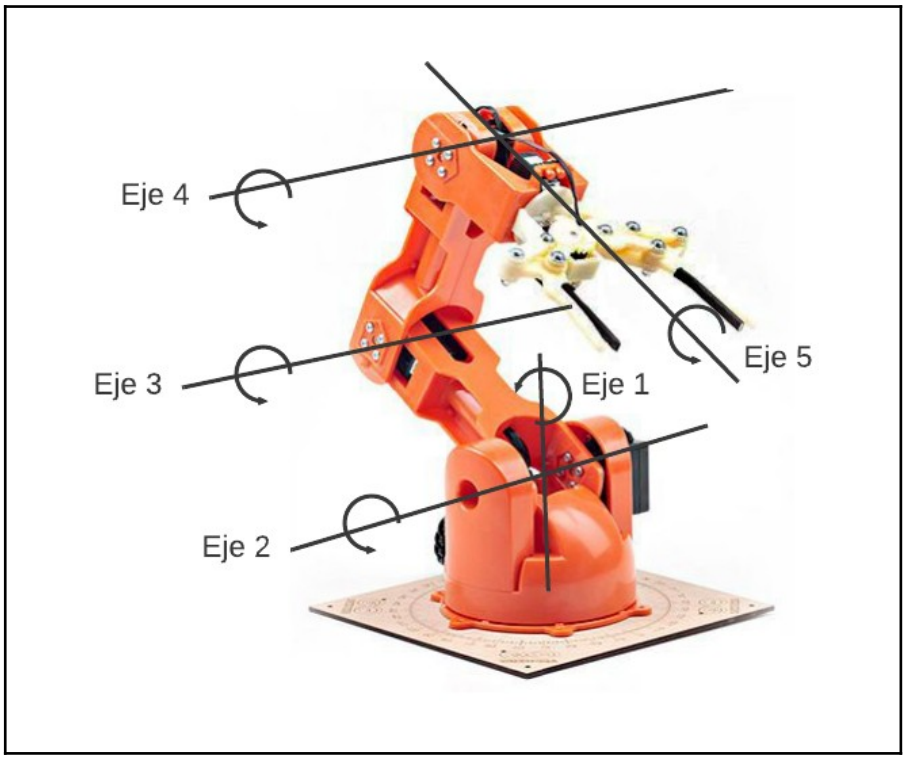


Figura 4. Braccio 5-GdL (Grados de Libertad). Fuente: elaboración propia

Seguidamente, se presentan algunos robots manipuladores compatibles con el «sistema operativo» ROS:

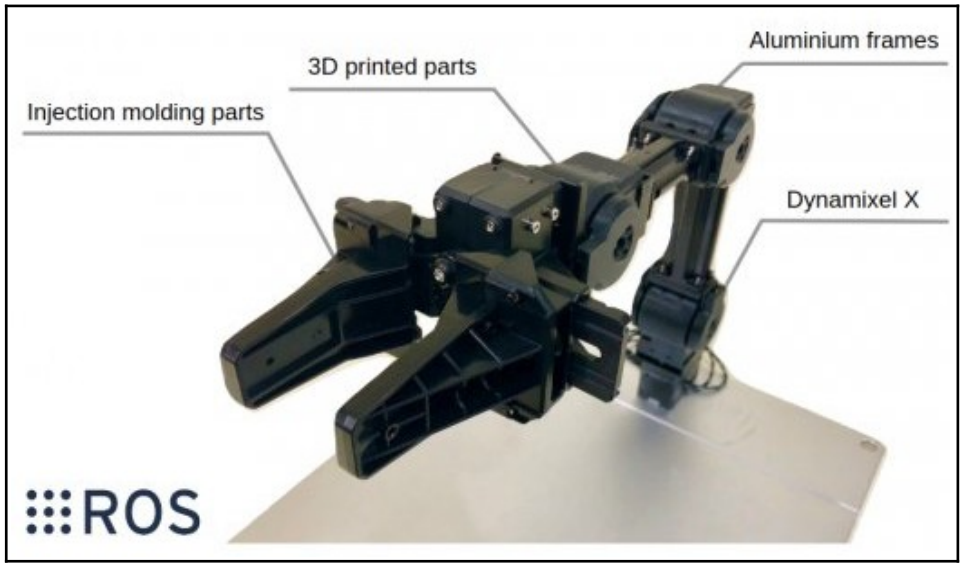


Figura 5. OpenManipulator

- **OpenManipulator**:
  - Fabricante: Robotis
  - Precio: 799,95 €
  - Grados de libertad: 4 GdL (+1 GdL Pinza)
  - Fuente (Figura 5 incluída): https://www.mybotshop.de/ROBOTIS-OpenManipulator_1

Figura 6. Uarm Swift Pro

- **Uarm Swfit Pro**
  - Fabricante: Ufactori
  - Precio: 880 €
  - Grados de libertad: 4 GdL
  - Fuente (Figura 6 incluída): https://www.mybotshop.de/UFactory-uArm-Swift-Pro

Figura 7. PhantomX Reactor Robot Arm

- **PhantomX Reactor Robot Arm**
  - Fabricante: Trossenrobotics
  - Precio: 749,95 €
  - Grados de libertad: 5 GdL
  - Fuente (Figura 7 incluída): https://www.mybotshop.de/TrossenRobotics-PhantomX-Reactor-Robot-Arm

Figura 8. AL5D

- **ALD50**
  - Fabricante: Lynxmotion
  - Precio: 158,95 €
  - Grados de libertad: 5 GdL
  - Fuente (Figura 8 incluída): https://www.mybotshop.de/Lynxmotion-AL5D

La lista de robots manipuladores disponibles en el mercado actualmente es muy amplia, nosotros nos hemos limitado a seleccionar aquellos con un precio más asequible (menor de 1000 €) y compatibles con ROS. No obstante, también existe la posibilidad de adquirir cualquier robot barato y "rosificarlo" (*rosify*), es decir, crear los paquetes y archivos CAD (.stl, por ejemplo) necesarios para hacer el robot compatible con ROS. Además, si se dispone de una impresora 3D nada nos impide crear un robot propio sin tener que partir de cero, en cuanto a la parte de software se refiere. ¡Esa es la magia de ROS!

Finalizar comentando que ROS se está introduciendo en el terreno de los microcontroladores. *eProsima*, una empresa española con base en Madrid, ofrece la infraestructura *middleware* necesaria (Micro-ROS) para poder integrar proyectos de robótica con este tipo de hardware en ROS 2.

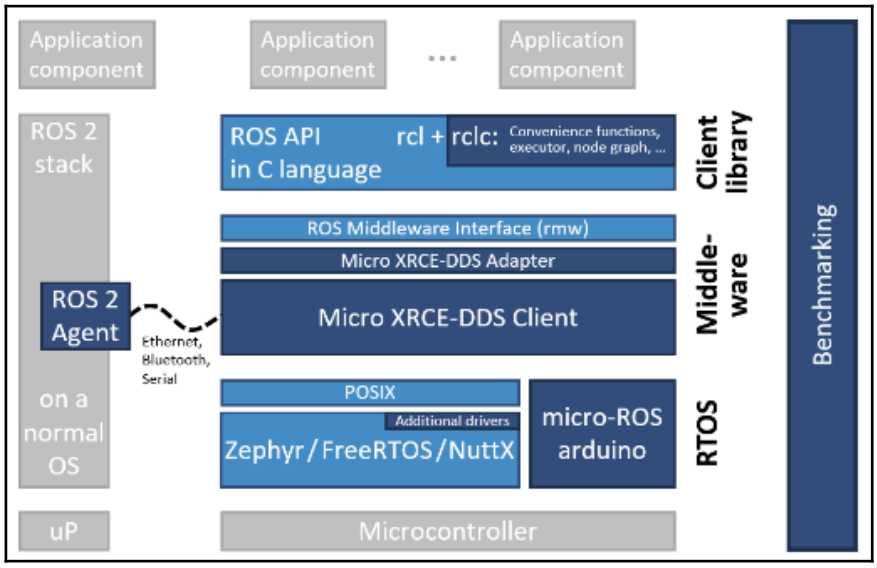


Figura 9. Arquitectura Micro-ROS. Fuente: [9]

3. Diseño propuesto

3.1 Arquitectura de hardware

Nuestro Tinker Braccio es un brazo robótico que se vende desmontado, siguiendo las instrucciones del mismo, y tras unas 3 horas de montaje y la configuración del primer *sketch* (testbraccio90.ino) para alinear los servo-motores, el robot queda totalmente operativo.

Figura 10. Tinker Braccio *unboxed*. Fuente: [3]

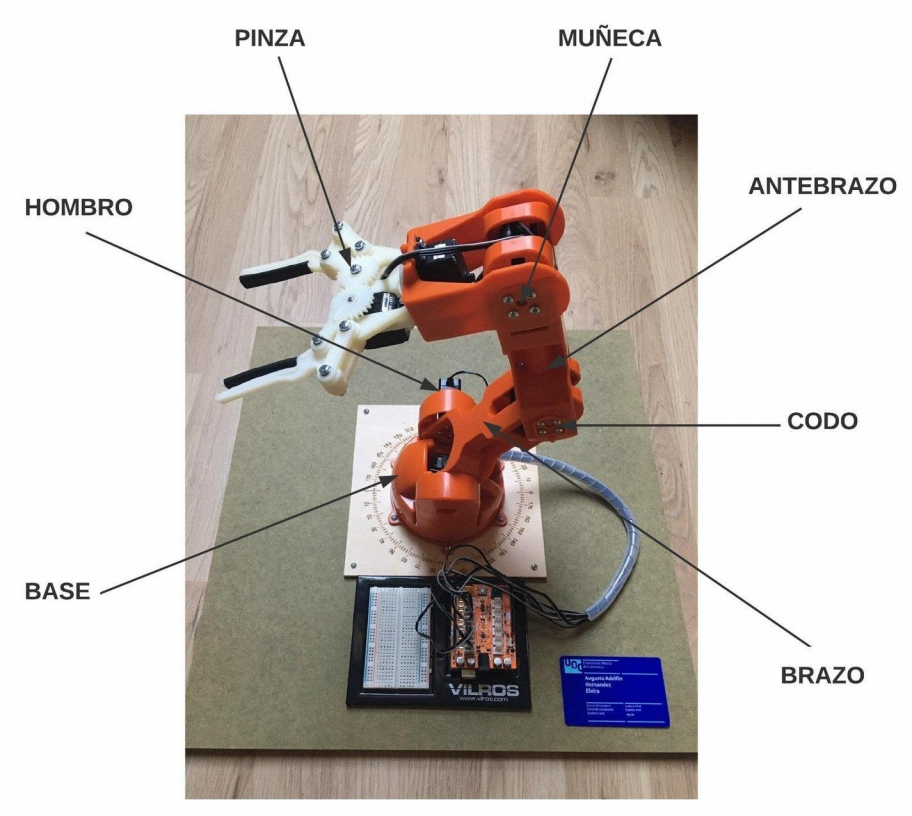


Figura 11. Tinker Braccio montado. Fuente: elaboración propia

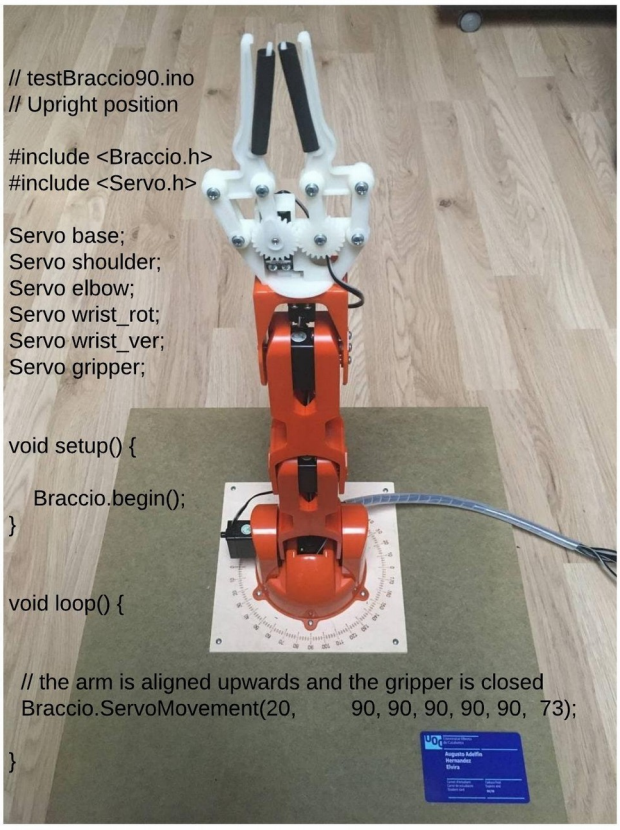


Figura 12. Sketch testBraccio90.ino: posición «*upright*». Fuente: elaboración propia

En la siguiente figura se pueden identificar las diferentes partes del hardware que componen el sistema:

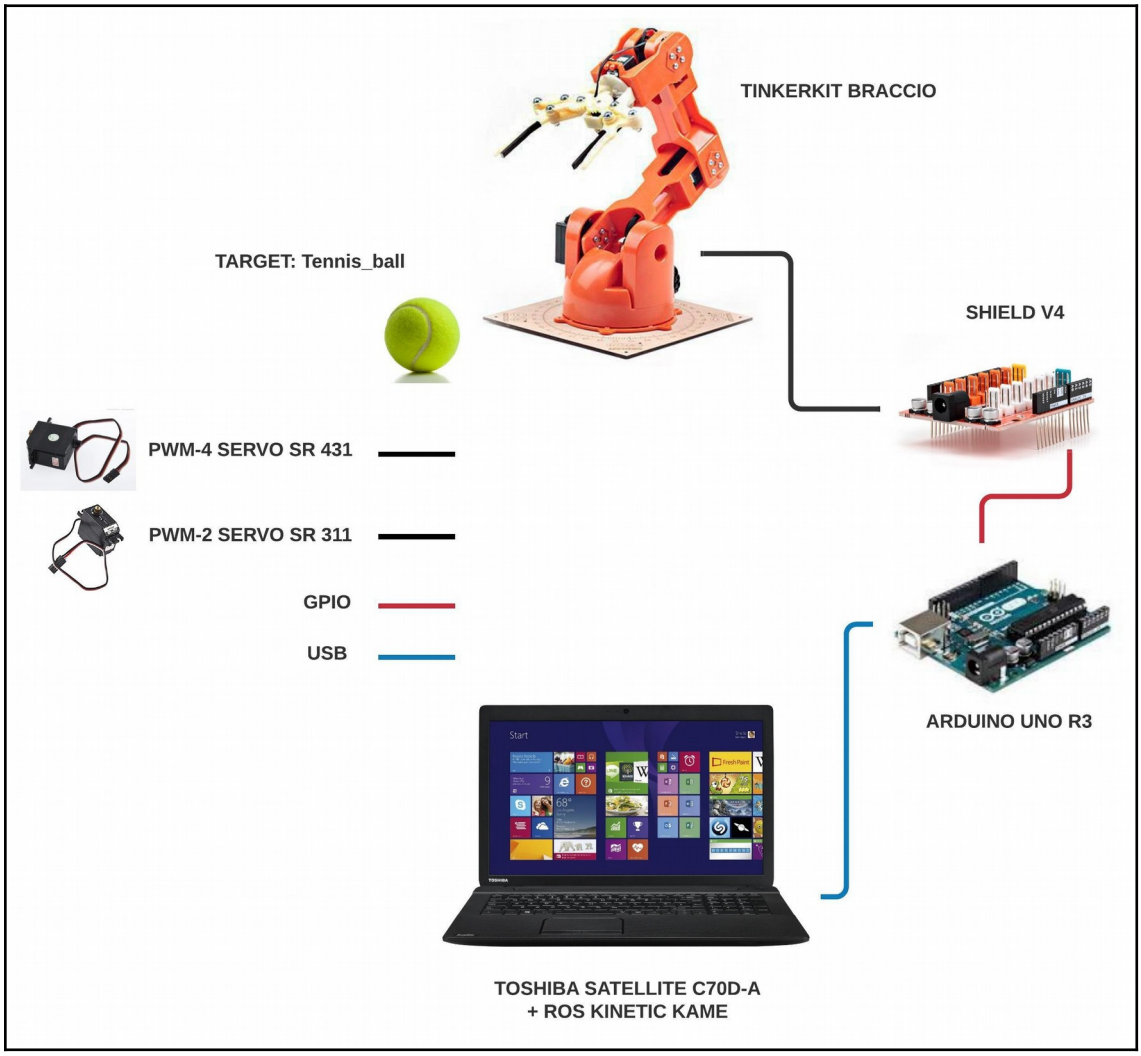


Figura 13. Arquitectura de hardware. Fuente: elaboración propia

Como puede observarse en la anterior figura, a partir de un ordenador portátil el programador puede acceder al sistema operativo Ubuntu 16.04 LTS, y a la versión de ROS Kinetic Kame previamente instalada. La conexión entre el ordenador y el microcontrolador Arduino se hará físicamente mediante un cable USB. La conexión entre el microcontrolador Arduino y la *shield* se realiza a través de los pines GPIO (***G**eneral-**P**urpose **I**nput **O**utput*), mientras que una señal PWM (***P**ulse **W**idth **M**odulation*) se encarga de controlar los actuadores o servomotores.

La pelota de tenis, si bien no es un elemento de hardware, sí que es una parte importante del sistema para poder realizar algunas pruebas y movimientos con el robot manipulador. La elección de dicho objeto no fue trivial, la pinza del

Arduino Braccio es muy resbaladiza, la superficie de contacto es insuficiente y no permite agarrar cualquier objeto de reducido diámetro, grandes dimensiones o demasiado peso. Después de varios experimentos con diferentes objetos (manzana, vaso, botella, figuras geométricas, etc.) se comprobó que la pelota de tenis era la mejor alternativa disponible.

A continuación se describen las características técnicas cada uno de los componentes: [3], [4]

Tabla 1. Características técnicas del TinkerKit Braccio

Distancia de funcionamiento máxima	80 cm
Altura máxima	52 cm
Anchura de la base	14 cm
Ancho de la pinza	9 cm
Peso total	792 g
Capacidad de carga máxima/peso a 32 cm distancia de funcionamiento	150 g
Peso máximo en la base de configuración Braccio	400 g

Tabla 2. Características técnicas de los servomotores

Servomotor	**Articulación**	**Rango de movimiento permitido (ángulo en grados)**	**Modelo**
M1	Base	0-180	SR 431 (PWM)
M2	Hombro	15-165	SR 431 (PWM)
M3	Codo	0-180	SR 431 (PWM)
M4	Muñeca_elevación	0-180	SR 431 (PWM)
M5	Muñeca_giro	0-180	SR 311 (PWM)
M6	Pinza	10-73	SR 311 (PWM)

Tabla 3. Características técnicas de la Tinkerkit Shield

Versión	V4
Voltaje de funcionamiento	5 V
Consumo de potencia	20 mV
Corriente máxima	1,1 A desde los conectores M1 a M4 750 mA desde los conectores M5 a M6

Tabla 4. Características técnicas de la placa Arduino Uno

Microcontrolador	ATmega328P
Voltaje de funcionamiento	5 V
Voltaje de entrada (recomendado)	7-12 V
Voltaje de entrada (límite)	6-20 V
Pines de E/S digitales	14 (de los cuales 6 proporcionan salida PWM)
Pines de entrada analógica	6
Pines PWM Digital E/S	6
Corriente DC por Pin de E/S	20 mA
Corriente DC para Pin de 3.3V	50 mA
Memoria Flash	32 KB de los cuales 0.5 KB utilizados por el gestor de arranque
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Velocidad del reloj	16 MHz
LEDS	13
Longitud	68.6 mm
Ancho	53.4 mm
Peso	25 g

3.2 Arquitectura de software

El componente de software más importante de este proyecto es la versión Kinetic Kame de ROS, pero también se hará uso de la IDE (***I****ntegrated* ***D****evelopment* ***E****nvironment*) de Arduino.

Como ya se comentó anteriormente, ROS es un software distribuido que contiene una gran variedad de librerías, herramientas y paquetes. Para nuestro trabajo vamos a necesitar el simulador de tres dimensiones RViz, y la aplicación para la manipulación de robots móviles *MoveIt*!

El resto de componentes de software que completan nuestra arquitecura, y que serán desarrollados más adelante son: URDF, TF, ROSserial, *Robot State Publisher* y *Joint State Publisher*.

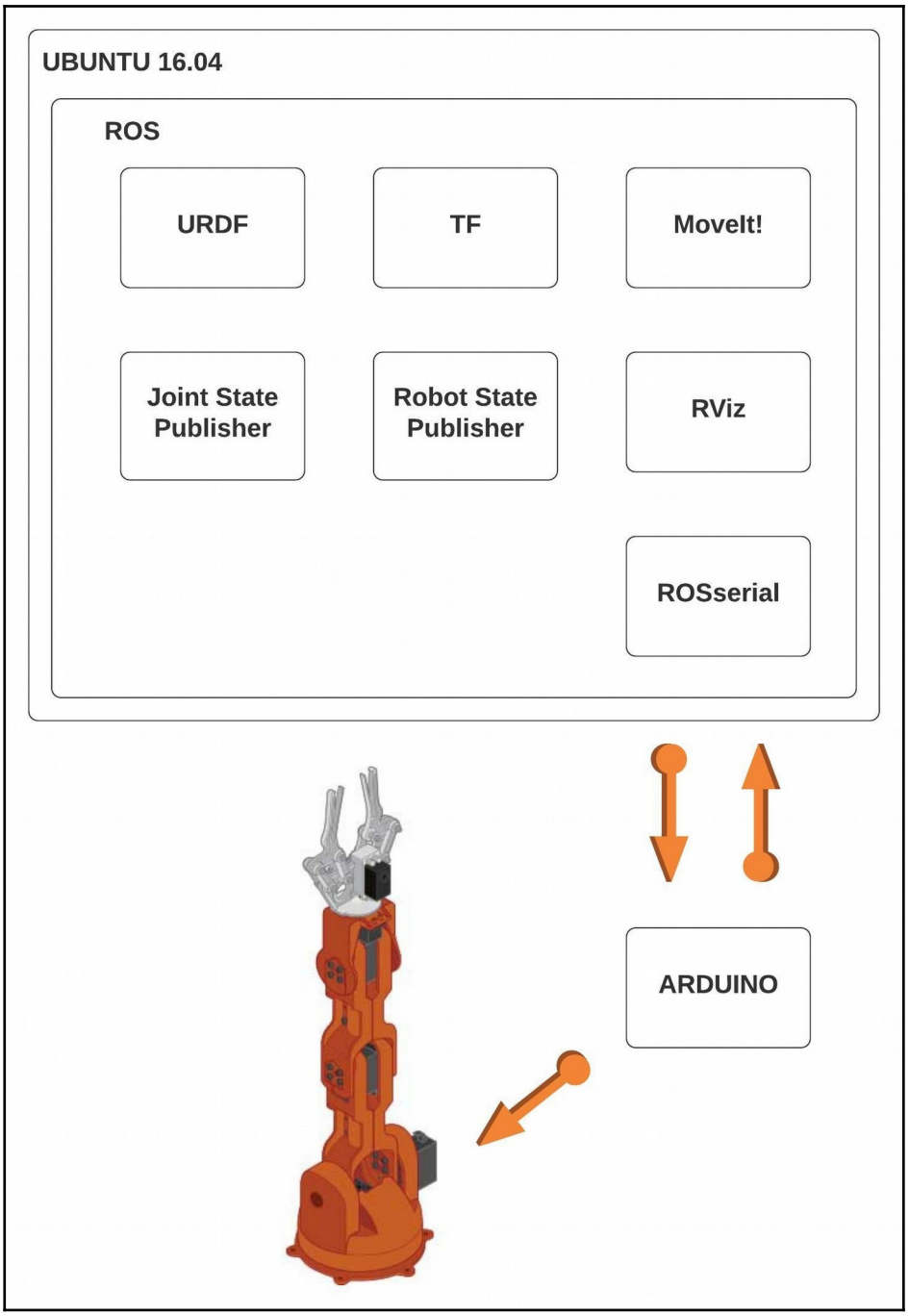


Figura 14. Diagrama de la arquitectura de software. Fuente: elaboración propia

4. ROS: conceptos y comandos básicos

En este capítulo se introducirán los conceptos y comandos más relevantes para entender este trabajo. No se presenta una lista exhaustiva ni detallada, puesto que la wiki oficial de ROS contiene toda la documentación necesaria.

La arquitectura de ROS se puede dividir en tres niveles conceptuales: [5]

- ***Filesystem level***: Grupo de conceptos que son usados para explicar cómo está formado ROS, estructura de carpetas y archivos necesarios para funcionar.

- ***Computation Graph level***: Lugar donde sucede la comunicación entre procesos y sistemas. Grupo de conceptos para configurar sistemas, gestionar procesos y comunicaciones.

- ***Community level***: Grupo de herramientas y conceptos usados para compartir conocimientos, algoritmos y código de cualquier desarrollador.

Del primer nivel nos interesan sobretodo los siguientes conceptos:

Packages: Los paquetes son la unidad más básica de la organización del software y contienen la información de los nodos. Es lo primero que se debe crear cuando se comunican los nodos entre sí. Dentro del paquete creado se encontrarán todos los códigos de los nodos, así como de los mensajes que se hayan creado exclusivamente para los mismos.

Messages types: Los mensajes son archivos de diferentes tipos que se envían entre nodos. El tipo de mensaje será importante a la hora de crear un nodo o un tópico. Cada tópico solo será capaz de transmitir un tipo de mensaje y los nodos trasmitirán o leerán el tipo de mensaje que se le ordene. Los diferentes tipos de mensajes según su complejidad:

  - ***Standard primitive***: números enteros, números reales, booleanos, strings, etc.

- ***Complex***: creados por el usuario, como *arrays*, por ejemplo, que pueden estar a su vez compuestos de tipos primitivos o complejos.

Del segundo nivel, por otra parte, podemos destacar:

Nodes: un nodo es la unidad básica del edificio ROS. Los nodos son procesos que pueden realizar cálculos, ejecutar algunas tareas y comunicarse gracias a la red ROS. Cada nodo se registra en la red con una identificación única y una lista de temas y servicios de los que desea enviar o recibir mensajes y algunos parámetros de conexión adicionales. ROS proporciona bibliotecas para escribir los nodos con los lenguajes C++ o Python.

- ***Publisher***: Estos nodos son de código secuencial, es decir, cada instrucción sigue a otra anterior. Se les denomina nodos publicadores porque publican mensajes en un tópico, con intención de que un suscriptor los reciba.

- ***Subscriber***: Utilizan un código basado en eventos, es decir, su ejecución dependerá de un suceso ocurrido en el sistema que estará definido por el usuario. Se les denomina nodos suscriptores porque están suscritos a un tópico y esperan a que éste reciba mensajes para poder leerlos.

Topics: Los dos grupos de nodos expuestos anteriormente utilizan este canal para compartir los mensajes. Es de gran importancia que tanto, el nodo publicador como el suscriptor, se comuniquen mediante el mismo tópico, para que dicha comunicación sea satisfactoria.

Messages: los nodos se comunican enviando mensajes entre sí, compartiendo la información y el estado del sistema. Un mensaje es simplemente una estructura de datos que contiene diversos campos. Se admiten los tipos primitivos estándar (entero, flotante, booleano, etc.), al igual que los arrays de tipos primitivos. Los mensajes pueden incluir (al igual que las estructuras de C) estructuras y arrays anidados arbitrariamente.

ROS *Master*: es un tipo especial de nodo, que ofrece una vista general del sistema, y proporciona una tabla de búsqueda única con todos los elementos

presentes en el mismo. Es el encargado de permitir la comunicación entre todos los nodos, es decir, si no ejecutamos este nodo no será posible tal comunicación. El maestro será el que permita la suscripción de los nodos en el sistema. Una vez creados dentro del sistema, el nodo publicador tendrá que pedir permiso para convertirse en publicador de un tópico y a su vez el suscriptor pedirá permiso para suscribirse al tópico. Una vez dichos permisos han sido concedidos por el Maestro el sistema está listo para funcionar, y las comunicaciones entre nodos estarán creadas.

Service/Client: cuando se necesita una comunicación de solicitud/respuesta los nodos pueden implementar una arquitectura tipo *service/client*. Un nodo ofrecerá un servicio a otros nodos (clientes) que así lo requieran, solicitar cierta información o que se realicen algunos cálculos en su nombre.

Algunos comandos básicos que nos pueden resultar útiles son: [6]

Una vez instalado ROS habrá que ejecutar el comando ***source*** en la terminal de Ubuntu, de forma que ROS pueda acceder a los diferentes paquetes que lo componen. Deberá ejecutar este comando en cada nueva terminal que abra para tener acceso a los comandos ROS, a no ser que haya configurado previamente el archivo .bashrc para no tener que repetir este proceso.

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

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

El siguiente paso es crear un nuevo espacio de trabajo con el comando ***catkin_make*** en un lugar del escritorio que nos interese:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Adicionalmente, para poder utilizar cualquiera de los paquetes que haya en el espacio de trabajo, habrá que usar nuevamente el comando ***source*** con el archivo *devel/setup.bash* que se encuentre dentro de dicho espacio de trabajo.

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

Ahora ya estamos en condiciones de crear un nuevo paquete o descargarlo de alguna página web como *Github*. Para ejecutar un nodo tenemos que utilizar el comando ***rosrun***. Los parámetros necesarios para este comando son el nombre del paquete, y el archivo del nodo que se quiere ejecutar. Para poder ejecutar dicho comando es necesario haber ejecutado anteriormente, en otra terminal, el comando ***roscore***, que es el que pone en marcha todo el ecosistema de ROS Master.

```
roscore
```

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/roslaunch-machine_name-130
39.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://machine_name:33919/
ros_comm version 1.4.7

SUMMARY
======

PARAMETERS
 * /rosversion
 * /rosdistro

NODES

auto-starting new master
process[master]: started with pid [13054]
ROS_MASTER_URI=http://machine_name:11311/

setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
process[rosout-1]: started with pid [13067]
started core service [/rosout]
```

```
$ rosrun [package_name] [node_name]
```

Por último, comentar que con el comando ***roslunch*** y los archivos de configuración creados para él, se pueden inicializar un grupo de nodos de forma simultánea y coordinada.

```
$ roslaunch package_name file.launch
```

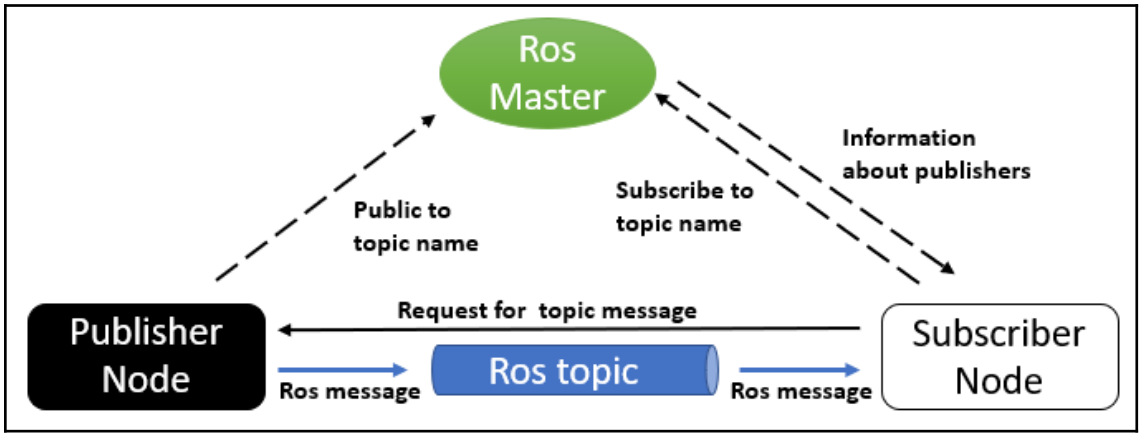


Figura 15. Componentes del sistema ROS. Fuente: [24]

5. Cinemática directa: D-H vs. URDF

La cinemática directa se refiere al uso de las ecuaciones cinemáticas de un robot para calcular la posición del efector final a partir de valores especificados para los parámetros de la articulaciones. [7]

El problema de la cinemática directa se puede resolver calculando las matrices de transformación homogénea, en las que se utiliza la notación de Denavit-Hartenberg (D-H), la cual expresa la posición y orientación de la articulación i respecto a la articulación i-1 mediante los siguientes parámetros:

θi: ángulo de Xi-1 a Xi medido sobre el eje de rotación Zi-1.

di: distancia de Xi-1 a Xi medida a lo largo del eje Zi-1.

ai: distancia de Zi-1 a Zi medida a lo largo del eje Xi .

αi: ángulo de Zi-1 a Zi medido sobre el eje de rotación Xi.

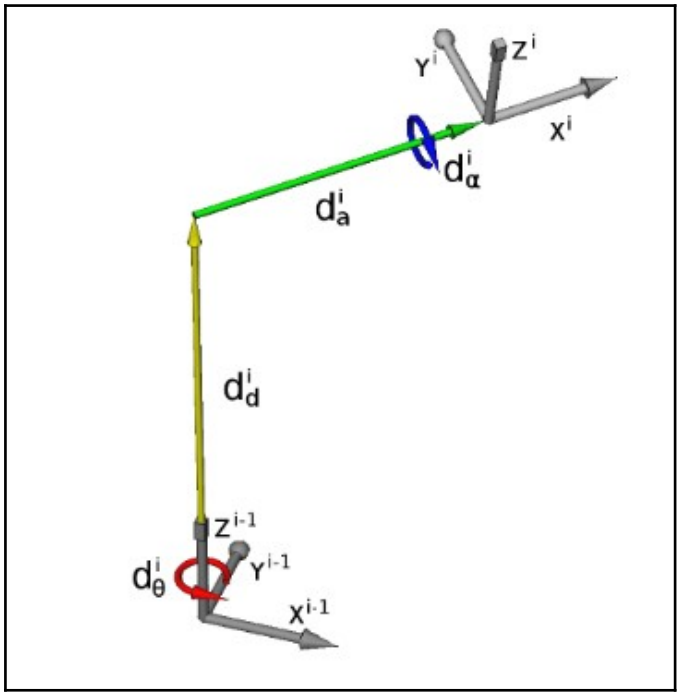


Figura 16. Parámetros Denavit-Hartenberg de las articulaciones. Fuente: [8]

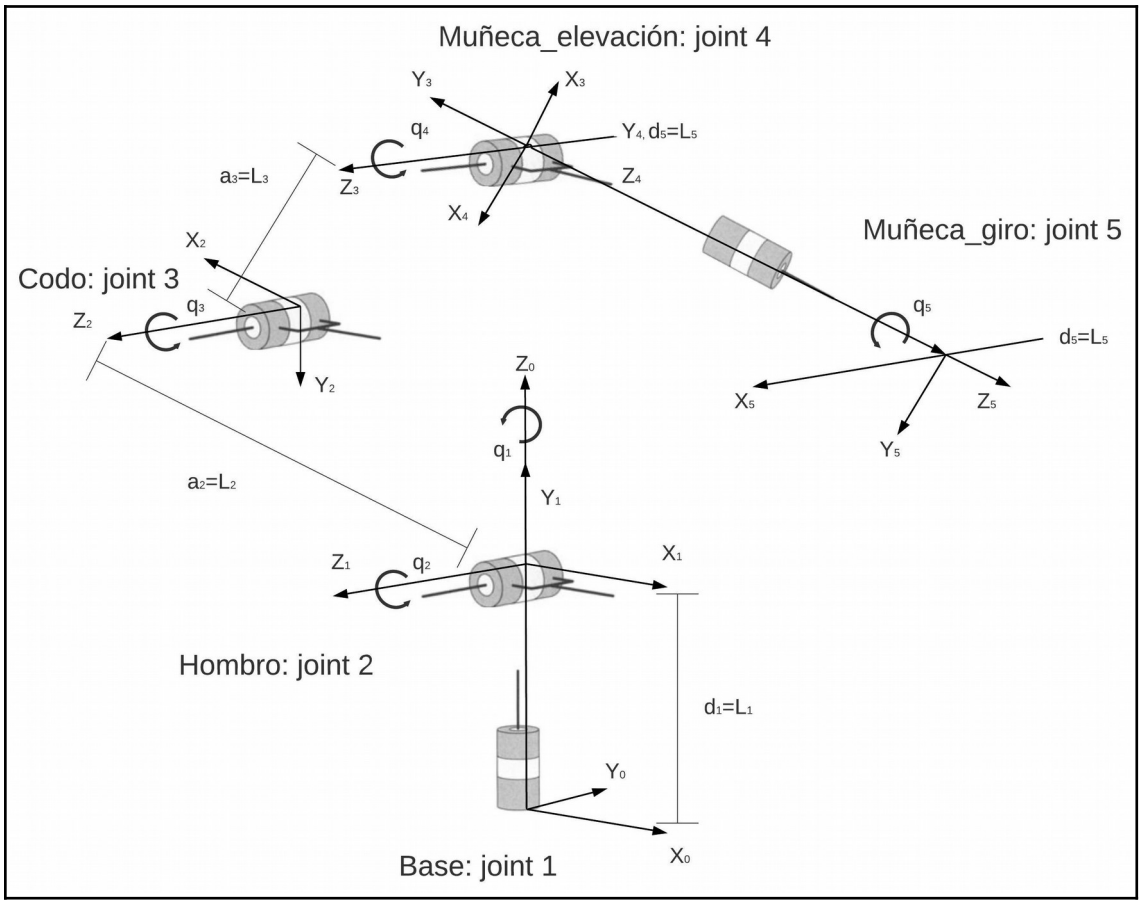


Figura 17. Cinemática directa. Asignación de las articulaciones y sistemas de coordenadas según la posición «*safety*» y el algoritmo de Denavit-Hartenberg. Fuente: elaboración propia

Figura 18. Posición «*safety*». Fuente: [3]

Para poder resolver el problema cinemático directo del manipulador Tinkerkit Braccio, hay que que localizar los sistemas de coordenadas de cada una de las articulaciones del mismo, según la pose inicial elegida (Figura 17 y 18). Posteriormente, se especifican los parámetros de Denavit-Hartenberg del robot y se completa la Tabla 5.

A continuación se presenta un resumen del procedimiento D-H a seguir, para asignar correctamente los sistemas de coordenadas de los *links* o vínculos entre las articulaciones [10]:

"1. Identifique los ejes de articulación e imagine (o dibuje) líneas infinitas sobre ellos. Para los pasos del 2 al 5, considere dos de estas líneas adyacentes (en los ejes i e i + 1).

2. Identifique la perpendicular común entre ellos, o el punto de intersección. En el punto de intersección, o en el punto en el que la perpendicular común se encuentra con el i-ésimo eje, asigne el origen de la trama asociada al vínculo.

3. Asigne el eje Zi para que apunte sobre el i-ésimo eje de articulación.

4. Asigne el eje Xi para que apunte sobre la perpendicular común o, si los ejes se intersectan, asigne Xi para que sea normal al plano que contiene los dos ejes.

5. Asigne el eje Yi para completar un sistema de coordenadas de mano derecha.

6. Asigne {0} para que concuerde con {1} cuando la primera variable de articulación sea cero. Para {N} seleccione la ubicación del origen y la dirección de XN libremente, pero generalmente de manera que haga que la mayor parte de los parámetros de los vínculos sean cero".

Aplicar correctamente el algoritmo D-H no resulta trivial y requiere algo de práctica. Es importante tener en cuenta, que la mayoría de manipuladores que nos vamos a encontrar suelen tener 5, 6 o 7 grados de libertad (GdL/DoF), y varios sistemas de coordenadas en articulaciones perpendiculares entre sí. El objetivo de dicho algoritmo consiste, básicamente, en que partiendo de un

frame i-1 (sistemas de coordenadas) de origen que se puede mover en el espacio (dos traslaciones, dos rotaciones) se alcance la posición y orientación del *frame* i de destino que es fijo.

En nuestro esquema cinemático de la Figura 16 puede observarse que los *frames* (sistemas de coordenadas) de la base y el hombro son perpendiculares (se intersectan los ejes de rotación z0 y z1), por lo que es conveniente elegir los ejes x0, x1 con el mismo ángulo medido en un plano perpendicular al el eje zi-1=z0 (evitando el *offset* inicial). De tal forma que partiendo del *frame*_origen {i-1=0} (base) podamos alcanzar la posición y orientación del *frame*_destino (hombro) {i=1} por medio de los parámetros de traslación di=d1=L1 (distancia a lo largo de zi−1=z0 desde el origen del *frame* {i-1=0} hasta el punto de intersección del eje zi-1=z0 con el eje xi=x1) y de rotación αi=α1=90° (ángulo de rotación alrededor del eje xi desde zi-1=z0 a zi=z1, usando la regla de la mano dereccha). Llegados a este punto, ya hemos relacionado el *frame* inicial o de origen {i-1=0} con el *frame* final o de destino, no obstante, la cadena cinemática continua y es ahora el *frame* {1}, el *frame*_origen, mientras que el frame {2} es el *frame*_destino. Ahora la relación entre *frames* es distinta, los ejes de rotación zi-1=z1 y zi=z2 ya no son mutuamente perpendiculares, sino que están colocados en paralelo. Señalar que según el algoritmo D-H, el eje de rotación de la articulación tiene que ser siempre el mismo (zi-1, zi). Por lo que, para poder alinear los dos ejes zi-1 y zi tenemos que realizar una traslación ai=a2 (distancia de traslación a lo largo de xi) y una rotación θi=θ2=180° (ángulo de rotación alrededor del eje zi−1 desde xi-1 a x, usando la regla de la mano derecha). El resto de la cadena cinemática se resuelve de forma similar y queda completada con la transición del *frame* {4} al {5}.

Tabla 5. Parámetros Denavit-Hartenberg

	i	$\theta_i + offset_i$	d_i	a_i	α_i
$0 \rightarrow 1$	1	θ_1	$L_1 = 6.5\ cm$	0	$\pi/2$
$1 \rightarrow 2$	2	$\theta_2 + \pi$	0	$L_2 = 12.5\ cm$	0
$2 \rightarrow 3$	3	$\theta_3 - \pi/2$	0	$L_3 = 12.5\ cm$	0
$3 \rightarrow 4$	4	$\theta_4 + \pi$	0	0	$-\pi/2$
$4 \rightarrow 5$	5	θ_5	$L_5 = 19\ cm$	0	0

Una vez obtenidos los parámetros D-H, el cálculo de las relaciones entre todos los *frames* consecutivos de la cadena cinemática es inmediato, ya que vienen dadas por el siguiente producto entre matrices en el orden indicado (recordar que el producto de matrices no cumple la propiedad conmutativa) [11]:

$$^{i-1}\boldsymbol{A}_i = \boldsymbol{Rotz}(\theta_i)\cdot\boldsymbol{T}(0,0,d_i)\cdot\boldsymbol{T}(a_i,0,0)\cdot\boldsymbol{Rotx}(\alpha_i) \qquad (1)$$

$$^{i-1}\boldsymbol{A}_i = \begin{bmatrix} C\theta_i & -S\theta_i & 0 & 0 \\ S\theta_i & C\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\alpha_i & -S\alpha_i & 0 \\ 0 & S\alpha_i & C\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} C\theta_i & -C\alpha_i S\theta_i & S\alpha_i S\theta_i & a_i C\theta_i \\ S\theta_i & C\alpha_i C\theta_i & -S\alpha_i C\theta_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (2)$$

donde $S\theta_i = \sin(\theta_i), C\theta_i = \cos(\theta_i), S\alpha_i = \sin(\alpha_i), C\alpha_i = \cos(\alpha_i)$

Ahora sustituyendo los parámetros D-H de la Tabla 5 en (2), podemos calcular las matrices de transformación de cada enlace:

$$
{}^{0}\boldsymbol{A}_{1}=\begin{bmatrix} C\theta_1 & -C\alpha_1 S\theta_1 & S\alpha_1 S\theta_1 & a_1 C\theta_1 \\ S\theta_1 & C\alpha_1 C\theta_1 & -S\alpha_1 C\theta_1 & a_1 S\theta_1 \\ 0 & S\alpha_1 & C\alpha_1 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} C\theta_1 & 0 & S\theta_1 & 0 \\ S\theta_1 & 0 & -C\theta_1 & 0 \\ 0 & 1 & 0 & L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (3)
$$

donde $S\alpha_1=\sin(\pi/2)=1, C\alpha_1=\cos(\pi/2)=0, d_1=L_1, a_1=0$

$$
{}^{1}\boldsymbol{A}_{2}=\begin{bmatrix} C\theta_2 & -C\alpha_2 S\theta_2 & S\alpha_2 S\theta_2 & a_2 C\theta_2 \\ S\theta_2 & C\alpha_2 C\theta_2 & -S\alpha_2 C\theta_2 & a_2 S\theta_2 \\ 0 & S\alpha_2 & C\alpha_2 & d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} C\theta_2 & -S\theta_2 & 0 & L_2 C\theta_2 \\ S\theta_2 & C\theta_2 & 0 & L_2 C\theta_2 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (4)
$$

donde $S\alpha_2=\sin(0)=0, C\alpha_2=\cos(0)=1, d_2=0, a_2=L_2$

$$
{}^{2}\boldsymbol{A}_{3}=\begin{bmatrix} C\theta_3 & -C\alpha_3 S\theta_3 & S\alpha_3 S\theta_3 & a_3 C\theta_3 \\ S\theta_3 & C\alpha_3 C\theta_3 & -S\alpha_3 C\theta_3 & a_3 S\theta_3 \\ 0 & S\alpha_3 & C\alpha_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} C\theta_3 & -S\theta_3 & 0 & L_3 C\theta_3 \\ S\theta_3 & C\theta_3 & 0 & L_3 C\theta_3 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (5)
$$

donde $S\alpha_3=\sin(0)=0, C\alpha_3=\cos(0)=1, d_3=0, a_3=L_3$

$$
{}^{3}\boldsymbol{A}_{4}=\begin{bmatrix} C\theta_4 & -C\alpha_4 S\theta_4 & S\alpha_4 S\theta_4 & a_4 C\theta_4 \\ S\theta_4 & C\alpha_4 C\theta_4 & -S\alpha_4 C\theta_4 & a_4 S\theta_4 \\ 0 & S\alpha_4 & C\alpha_4 & d_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} C\theta_4 & 0 & -S\theta_4 & 0 \\ S\theta_4 & 0 & C\theta_4 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (6)
$$

donde $S\alpha_4=\sin(-\pi/2)=-1, C\alpha_4=\cos(-\pi/2)=0, d_4=0, a_4=0$

$$
{}^{4}\boldsymbol{A}_{5}=\begin{bmatrix} C\theta_5 & -C\alpha_5 S\theta_5 & S\alpha_5 S\theta_5 & a_5 C\theta_5 \\ S\theta_5 & C\alpha_5 C\theta_5 & -S\alpha_5 C\theta_5 & a_5 S\theta_5 \\ 0 & S\alpha_5 & C\alpha_5 & d_5 \\ 0 & 0 & 0 & 1 \end{bmatrix}=\begin{bmatrix} C\theta_5 & -S\theta_5 & 0 & 0 \\ S\theta_5 & C\theta_5 & 0 & 0 \\ 0 & 1 & 1 & L_5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (7)
$$

donde $S\alpha_5=\sin(0)=0, C\alpha_5=\cos(0)=1, d_5=L_5, a_5=0$

A partir de las matrices 4x4 anteriores (matriz general D-H más cinco matrices del Tinkerkit Braccio) ya podemos hacer los cálculos en Matlab:

-Primero implementamos la función MDH en el script MDH.m: [11]

```
function dh=MDH(teta,d,a,alfa)

dh=[cos(teta) -cos(alfa)*sin(teta) sin(alfa)*sin(teta) a*cos(teta);

   sin(teta) cos(alfa)*cos(teta) -sin(alfa)*cos(teta) a*sin(teta);

   0 sin(alfa) cos(alfa) d;

   0 0 0 1];
```

-Posteriormente se definen las variables simbólicas mediante la siguiente secuencia de comandos, y se obtiene el producto total T de las 5 matrices 4x4 para el Tinkerkit Braccio:

```
>> syms q1 q2 q3 q4 q5 l1 l2 l3 l5

>> pi1=sym(pi);

>> A01=MDH(q1,l1,0,pi1/2);

>> A12=MDH(q2+pi1,0,l2,0);

>> A23=MDH(q3-pi1/2,0,l3,0);

>> A34=MDH(q4+pi1,0,0,-pi1/2);

>> A45=MDH(q5,l5,0,0);

>> A02=simplify(A01*A12);

>> A03=simplify(A02*A23);

>> A04=simplify(A03*A34);

>> A05=simplify(A04*A45);

>> T=A05
```

[sin(q2 + q3 + q4)*cos(q1)*cos(q5) - sin(q1)*sin(q5), - cos(q5)*sin(q1) - sin(q2 + q3 + q4)*cos(q1)*sin(q5), cos(q2 + q3 + q4)*cos(q1), l5*cos(q2 + q3 + q4)*cos(q1) - cos(q1)*(l3*sin(q2 + q3) + l2*cos(q2))]

[cos(q1)*sin(q5) + sin(q2 + q3 + q4)*cos(q5)*sin(q1), cos(q1)*cos(q5) - sin(q2 + q3 + q4)*sin(q1)*sin(q5), cos(q2 + q3 + q4)*sin(q1), l5*cos(q2 + q3 + q4)*sin(q1) - sin(q1)*(l3*sin(q2 + q3) + l2*cos(q2))]

[-cos(q2 + q3 + q4)*cos(q5), cos(q2 + q3 + q4)*sin(q5), sin(q2 + q3 + q4), 1 + l3*cos(q2 + q3) - l2*sin(q2) + l5*sin(q2 + q3 + q4)]

[0, 0, 0,1]

Ahora ya somos capaces de hallar la orientación y posición del centro de la pinza o TCP (*Tool Center Point*) con respecto al origen, e introduciendo los ángulos de interés se puede evaluar la matriz T:

>> q1=0;q2=0;q3=0;q4=0;q5=0;

>> eval(T)

ans =

[0, 0, 1, l5 - l2]

[0, 1, 0, 0]

[-1, 0, 0, l1 + l3]

[0, 0, 0, 1]

La Toolbox *Robotics* de Peter Corke para Matlab incluye la función *Link* (con L mayúscula, ***case sensitive***) para la definición de los parámetros cinemáticos D-H de cada *link* o enlace, y la función SerialLink para la concatenación de los mismos conformando así el robot: [12]

>> L1=Link([0 6.5 0 pi/2 0 0])

L1 =

Revolute(std): theta=q, d=6.5, a=0, alpha=1.5708, offset=0

```
>> L2=Link([0 0 12.5 0 0 pi])

L2 =
Revolute(std): theta=q, d=0, a=12.5, alpha=0, offset=3.14159

>> L3=Link([0 0 12.5 0 0 -pi/2])

L3 =
Revolute(std): theta=q, d=0, a=12.5, alpha=0, offset=-1.5708

>> L4=Link([0 0 0 -pi/2 0 pi])

L4 =
Revolute(std): theta=q, d=0, a=0, alpha=-1.5708, offset=3.14159

>> L5 = Link ([0 19 0 0 0 0])

L5 =
Revolute(std): theta=q, d=19, a=0, alpha=0, offset=0

>> bot=SerialLink([L1 L2 L3 L4 L5], 'name', 'Tinkerkit Braccio')

bot =

Tinkerkit Braccio:: 5 axis, RRRRR, stdDH, slowRNE
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|        q1|        6.5|          0|     1.5708|          0|
|  2|        q2|          0|       12.5|          0|3.14159|
|  3|        q3|          0|       12.5|          0| -1.5708|
|  4|        q4|          0|          0| -1.5708| 3.14159|
|  5|        q5|         19|          0|          0|          0|
+---+-----------+-----------+-----------+-----------+-----------+
```

Según las instrucciones del manipulador *Tinkerkit Braccio* la pose o posición *"safety"* se configura con los servomotores M1=90°, M2=45°, M3=180°, M4=180°, M5=90° y M6=10°. El servomotor M6 no contabiliza como grado de libertad de rotación ya que se trata de la apertura (10°) y cierre (73°) de la pinza. En cuanto al servomotor M2=45° se traduce como M2=-135° según la convención D-H de la Toolbox *Robotics*. Introduciendo los ángulos [90°, -135°, 180°, 180°, 90°] en radianes por medio de la función *bot.plot* se obtiene la imagen de la posición inicial:

```
>> bot.plot([1.5708 -2.3562 3.14159 3.14159 1.5708])
```

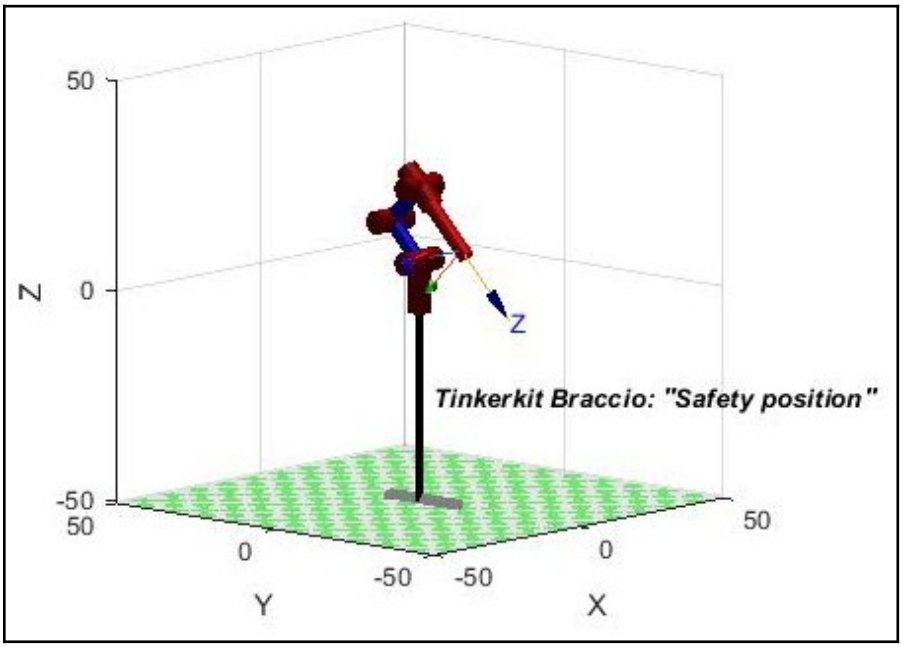


Figura 19. Posición «*safety*» en la Toolbox *Robotics*. Fuente: elaboración propia

Además, una vez definido el robot podemos usar la función *fkine (forward kinematics)* para obtener la matriz de transformación homogénea que localiza el TCP (*Tool Center Point*) para un vector de ángulos determinados:

```
>> fkine(bot,[1.5708 -2.3562 3.14159 3.14159 1.5708])
ans =
   -1.0000    0.0000    0.0000 4.935e-05
   -0.0000    0.7071   -0.7071   -13.44
   -0.0000   -0.7071   -0.7071    10.74
         0         0         0         1
```

Esta Toolbox tiene otras funciones y métodos interesantes como *bot.teach(),* que nos proporciona una GUI 3D (Graphical User Interface) interactiva en la que podemos introducir directamente los ángulos de las articulaciones y observar los cambios de posición (x,y,z) y orientación (**R**oll,**P**itch,**Y**aw) del robot:

```
>> bot.teach()
```

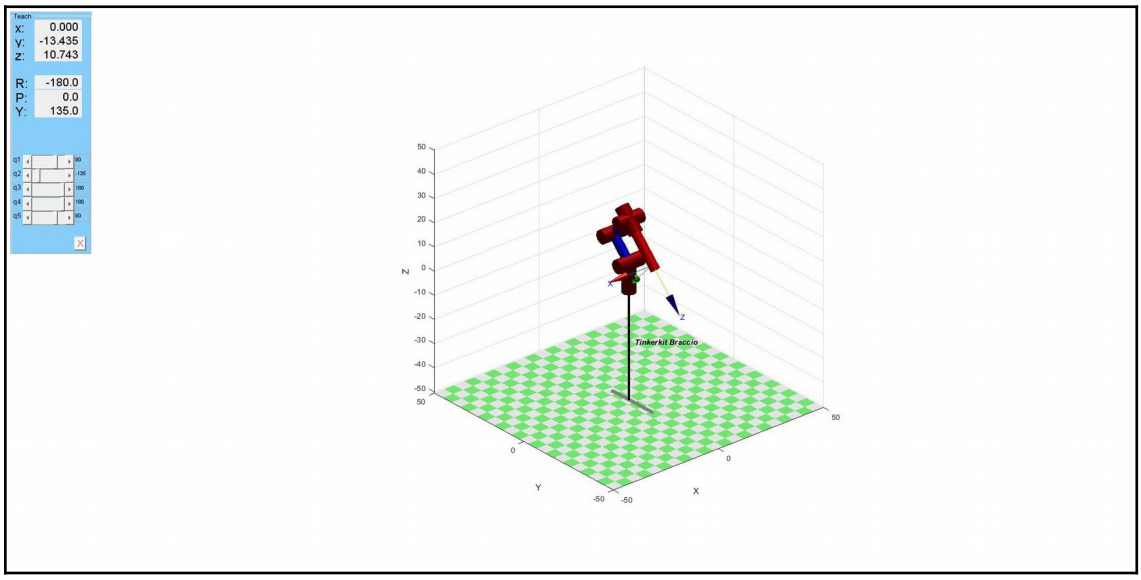


Figura 20. GUI Toolbox *Robotics*. Fuente: elaboración propia

Los parámetros D-H si bien son de uso obligado en la mayoría de manipuladores industriales, no resultan de gran utilidad en el *framework* ROS que usa otros parámetros denominados URDF (Universal Robot Description Format). Dichos parámetros se configuran en un archivo de formato XML (*eXtensible Markup Language*) para describir la cinemática, las propiedades inerciales y la geometría de enlaces de los robots. Un archivo URDF describe las articulaciones (*joints*) y enlaces (*links*) de un robot: [13]

-**Articulaciones**: Las articulaciones conectan dos enlaces: un enlace principal (*parent link*) y un enlace secundario (*child link*). Cada articulación (*joint*) tiene un sistema de referencia (*frame*) origen que define la posición y orientación del *frame* del *link* hijo (*child*) relativo al frame del *link* padre (*parent*) cuando la variable de articulación es cero. El origen está situado en el eje de la articulación. Cada articulación tiene un eje de rotación, un vector unitario expresado en el sistema de coordenadas del link hijo, en la dirección de rotación positiva para articulaciones de rotación (un grado de libertad).

Al describir la parametrización de URDF, tiene sentido considerar tres sistemas de coordenadas para cada articulación, el del *link* padre, el del *link* hijo y uno intermedio que se asigna a la articulación. En la Figura 20 el *link* padre es {i -

1}, los sistemas de coordenadas hijo y de la articulación {i} son solidarios, la articulación está en la posición cero.

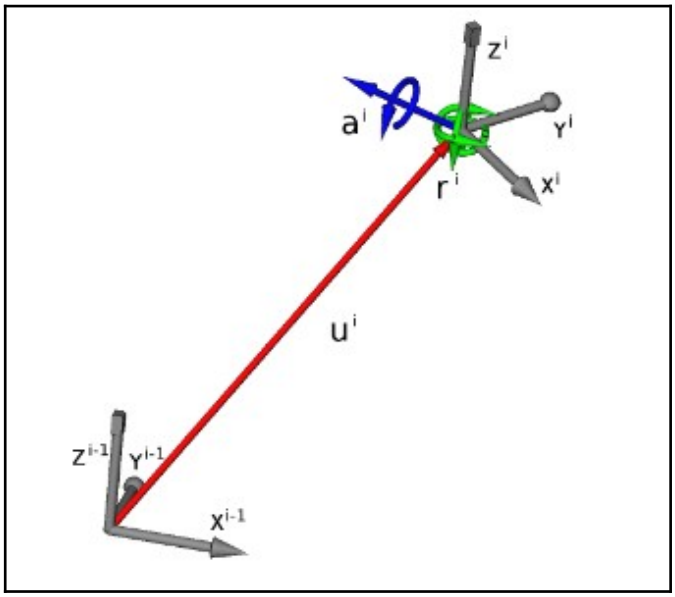


Figura 21. Parámetros URDF de las articulaciones. Fuente: [8]

Los sistemas de coordenadas se muestran en color gris. El vector rojo describe el desplazamiento del origen del sistema de coordenadas de la articulación con respecto al origen del *frame* padre. La inclinación relativa del sistema de coordenadas de la articulación está representada en verde, y consiste en los ángulos *Roll-Pitch-Yaw* (Alabeo-Cabeceo-Guiñada) de la Figura 21 expresados en radianes. Si la articulación es móvil, el vector del eje de la articulación (azul) es un vector unitario (módulo igual a uno), y se da en relación con el sistema de coordenadas de la articulación.

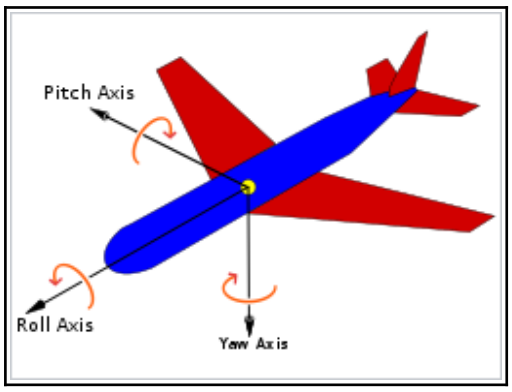


Figura 22. Posición de los tres ejes para describir su ángulo de rotación. Fuente: [14]

-**Enlaces**: Mientras que las articulaciones describen completamente la cinemática del robot, los enlaces definen sus propiedades dinámicas. Los elementos de un enlace incluyen su masa; un *frame* de origen que define la posición y orientación de un *frame* en el centro de masa del enlace en relación con el *frame* de articulación del enlace descrito anteriormente; y una matriz de inercia.

Como ya comentamos anteriormente ROS se divide en tres niveles conceptuales, volviendo al nivel que más nos interesa (***File System Level***) vemos que se organiza en una serie de carpetas y archivos.

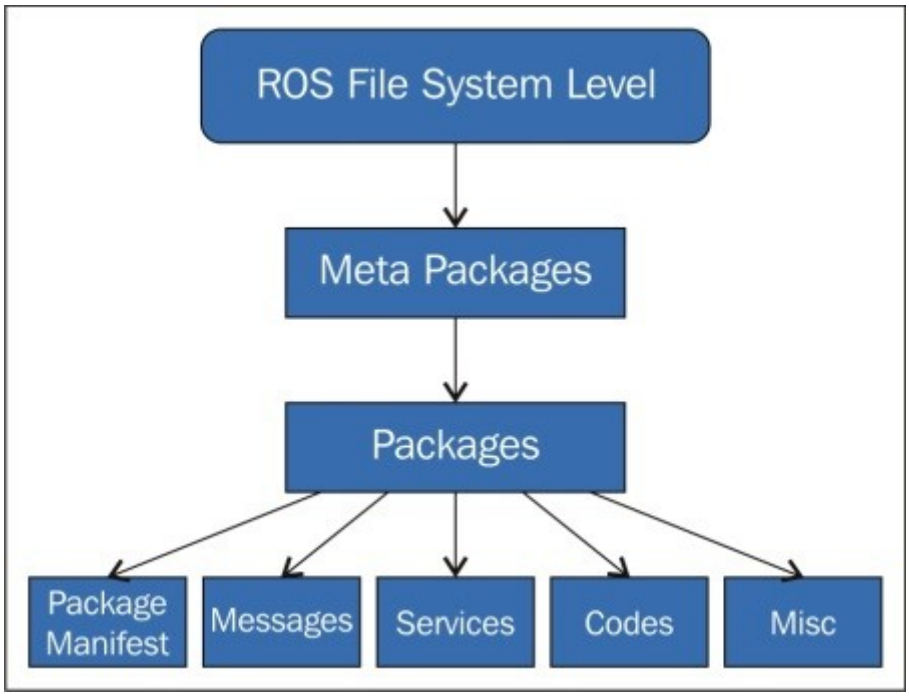


Figura 23. Nivel del sistema de archivos de ROS. Fuente: [15]

En este capítulo vamos a trabajar con el paquete (***Package***) exclusivamente creado para el Tinkerkit Braccio y disponible en el repositorio de Github: https://github.com/ohlr/braccio_arduino_ros_rviz. En concreto, vamos a explicar en detalle el contenido de las carpetas **urdf** y **stl** que aparecen en la Figura 24.

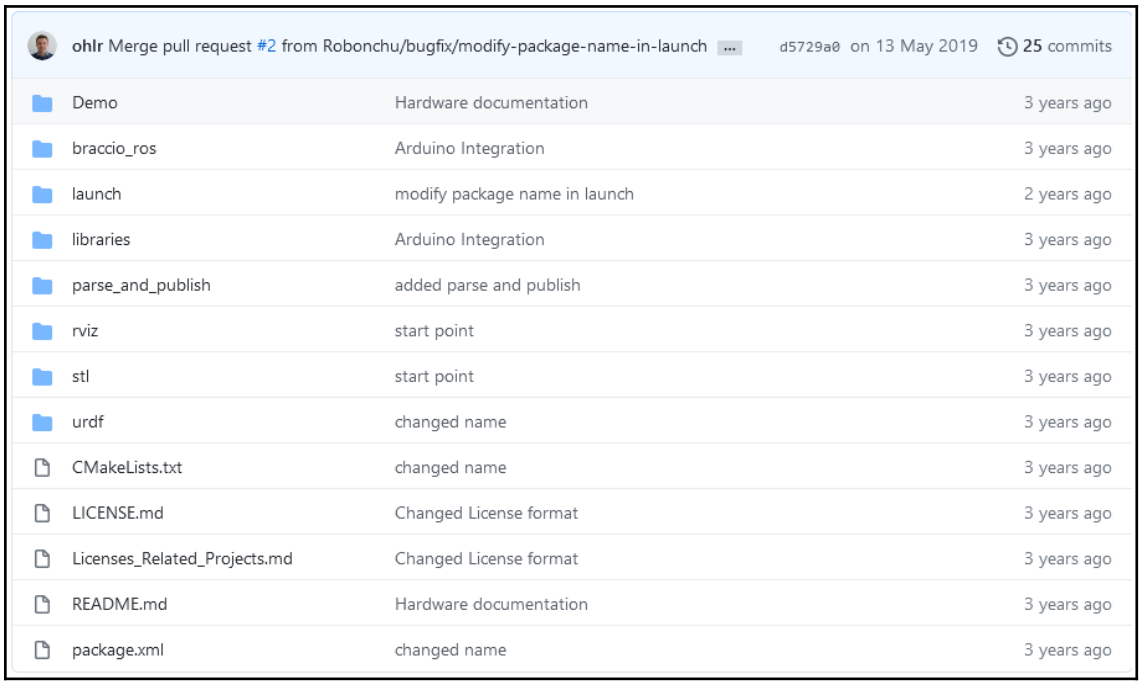


Figura 24. Nivel del sistema de archivos de ROS. Fuente: [16]

A continuación, pasamos a analizar el archivo braccio_arm.urdf (carpeta urdf) junto con los archivos .stl (carpeta stl) del diseño CAD de los distintos *links* del Tinkerkit Braccio. [16]

```
<?xml version="1.0"?>
<robot name="braccio">
    <link name="base_link">
        <visual>
            <geometry>
                <cylinder length="0.01" radius=".053" />
            </geometry>
            <material name="black" />
            <origin rpy="0 0 0" xyz="0 0 0" />
        </visual>
    </link>
```

En este primer fragmento de código se define un robot llamado “braccio” y el **primer enlace** (“base_link”) del mismo, de un manipulador que consta de 8 enlaces (*links*) y 7 articulaciones (j*oints*) en total (según la descripción URDF). Estas dos descripciones son los requisitos mínimos para poder definir un robot, se describen en sintaxis XML y pueden insertarse en cualquier orden del archivo. Pero es necesario que el resultado sea cinemáticamente consistente

con el árbol cinemático que forman el enlace padre, la articulación intermedia y el enlace hijo.

Es importante recordar que en la representación convencional de Denavit-Hartenberg solamente eran necesarios 4 parámetros, mientras que en la representación URDF podemos necesitar hasta 10 parámetros, lo que introduce cierto grado de confusión y redundancia, pero que queda totalmente justificada en ciertas ocasiones.

Una vez que le hemos puesto un nombre al primer enlace (“base_link”), el siguiente paso es definir el aspecto visual del mismo, donde se especifica la orientación y el origen geométrico de la pieza (xyz= “0 0 0“) respecto a la pieza anterior. Posteriormente se definirá la geometría de la pieza existiendo la posibilidad de que sea un cilindro, una esfera, un prisma rectangular o una forma predefinida en un archivo (mesh: .stl, .dae). Este *link* se ha creado con forma de un cilindro de radio 0.053 m (5.3 cm) y longitud 0.01 m (1 cm). Finalmente, en la parte visual también se podrá definir el color de la pieza (*black* en este caso) y su textura.

```
<link name="braccio_base_link">
        <visual>
            <geometry>
                                                                    <mesh
filename="package://braccio_arduino_ros_rviz/stl/braccio_base.stl"
scale="0.001 0.001 0.001" />
            </geometry>
            <material name="orange" />
            <origin rpy="0 0 3.1416" xyz="0 0.004 0" />
        </visual>
 </link>
```

El **segundo enlace** define su geometría a partir del archivo braccio_base.stl (Figura 24) en escala 1:1000. El directorio de la carpeta tiene que especificarse

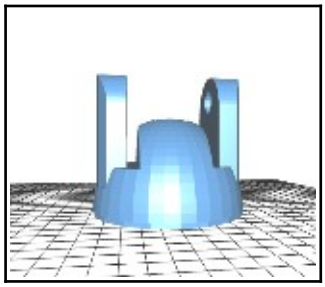


Figura 25. Archivo braccio_base.stl. Fuente: [16]

tras la palabra clave <mesh filename/>. El color de la pieza es naranja ("*orange*"), la pieza está rotada 180° (pi=3.1416) según el eje z y el origen ha sido desplazado 4 milímetros en dirección del eje y. Este tipo de desplazamientos xyz del origen del *link* depende del modelo del robot que tengamos (propiedades geométricas de los archivos .stl) y de las relaciones entre las distintas piezas.

```
<link name="shoulder_link">
        <visual>
            <geometry>
                                                                    <mesh
filename="package://braccio_arduino_ros_rviz/stl/braccio_shoulder.stl"
scale="0.001 0.001 0.001" />
            </geometry>
            <material name="orange" />
            <origin rpy="0 0 0" xyz="-0.0045 0.0055 -0.026" />
        </visual>
    </link>

    <link name="elbow_link">
        <visual>
            <geometry>
                                                                    <mesh
filename="package://braccio_arduino_ros_rviz/stl/braccio_elbow.stl"
scale="0.001 0.001 0.001" />
            </geometry>
            <material name="orange" />
            <origin rpy="0 0 0" xyz="-0.0045 0.005 -0.025" />
        </visual>
    </link>

    <link name="wrist_pitch_link">
        <visual>
            <geometry>
                                                                    <mesh
filename="package://braccio_arduino_ros_rviz/stl/braccio_wrist_pitch.s
tl" scale="0.001 0.001 0.001" />
            </geometry>
            <material name="orange" />
            <origin rpy="0 0 0" xyz="0.003 -0.0004 -0.024" />
        </visual>
    </link>

    <link name="wrist_roll_link">
        <visual>
            <geometry>
                                                                    <mesh
filename="package://braccio_arduino_ros_rviz/stl/braccio_wrist_roll.st
l" scale="0.001 0.001 0.001" />
            </geometry>
            <material name="white" />
            <origin rpy="0 0 0" xyz="0.006 0 0.0" />
        </visual>
    </link>

    <link name="left_gripper_link">
```

```
        <visual>
            <geometry>
                                                                    <mesh
filename="package://braccio_arduino_ros_rviz/stl/braccio_left_gripper.
stl" scale="0.001 0.001 0.001" />
            </geometry>
            <material name="white" />
            <origin rpy="0 1.5708 0" xyz="0 -0.012 0" />
        </visual>
    </link>

    <link name="right_gripper_link">
        <visual>
            <geometry>
                                                                    <mesh
filename="package://braccio_arduino_ros_rviz/stl/braccio_right_gripper
.stl" scale="0.001 0.001 0.001" />
            </geometry>
            <material name="white" />
            <origin rpy="0 1.5708 0" xyz="0 -0.012 0.010" />
        </visual>
    </link>
```

El **resto de enlaces** siguen el mismo procedimiento que el anterior (excepto el origen xyz relativo, la orientación y el color), y no hay mucho más que comentar al respecto.

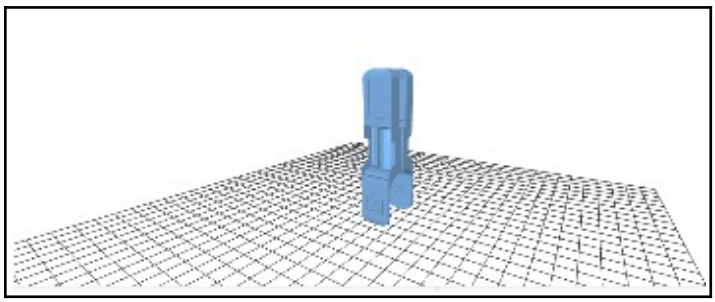


Figura 26. Archivo shoulder_link.stl. Fuente: [16]

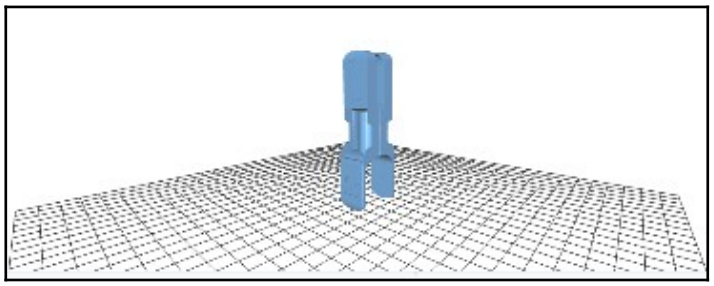


Figura 27. Archivo elbow_link.stl. Fuente: [16]

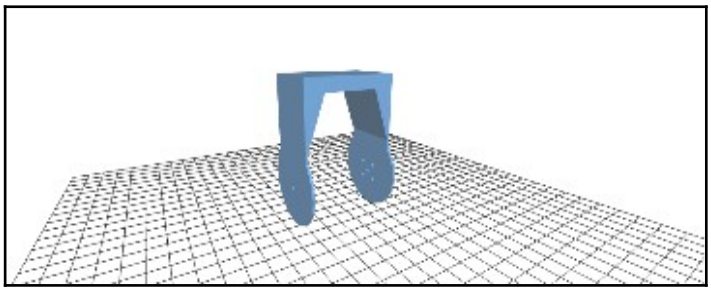


Figura 28. Archivo wrist_pitch_link.stl. Fuente: [16]

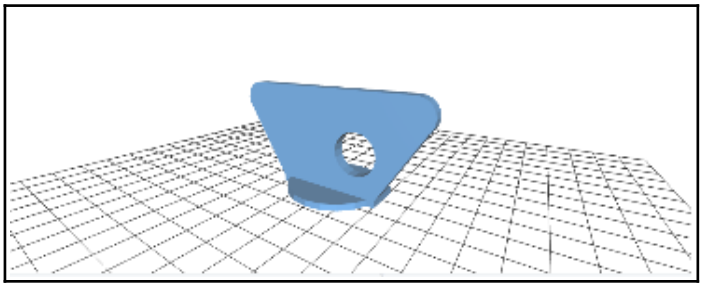


Figura 29. Archivo wrist_roll_link.stl. Fuente: [16]

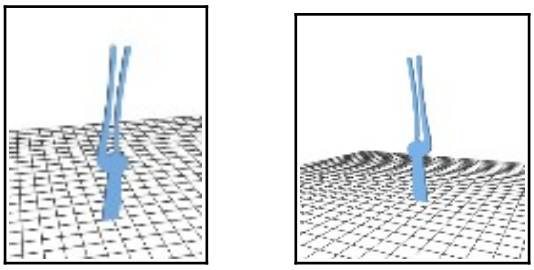


Figura 30. Archivos left_gripper_link.stl y right_gripper_link.stl. Fuente: [16]

Las articulaciones, por su parte, se definen con algunos parámetros distintos. El tipo (*type*) establecerá la clase de movimiento que tendrá esa articulación, que puede ser: *revolute* (giran respecto a un eje con unos límites), *continuous* (giran respecto a un eje sin limites de giro), *prismatic* (se desplaza a lo largo de un eje), *fixed* (fijos, no se mueven), *planar* (se mueven a lo largo de un plano) y *floating* (son articulaciones libres con 6 grados de libertad).

Una vez definido el tipo y el nombre del *joint*, hay que fijar otros parámetros obligatorios: su origen con respecto a la articulación anterior, el enlace padre (*parent link*) y el enlace hijo (*child link*). Seguidamente, si hemos definido algún

movimiento respecto a un eje también habrá que indicar cuál es ese eje de rotación. Y por último se pueden fijar algunos límites, como los límites de giro o desplazamiento alrededor de un eje (*upper* y *lower)*, los límites de velocidad de dicho eje y el esfuerzo que puede soportar el joint.

```
<joint name="base_joint" type="revolute">

       <axis xyz="0 0 1" />
                  <limit  effort="1000.0"  lower="0.0"  upper="3.1416"
velocity="1.0" />
        <origin rpy="0 0 0" xyz="0 0 0" />
        <parent link="base_link" />
        <child link="braccio_base_link" />
    </joint>

    <joint name="shoulder_joint" type="revolute">
        <axis xyz="1 0 0" />
                <limit  effort="1000.0"  lower="0.2618"  upper="2.8798"
velocity="1.0" />
        <origin rpy="-1.5708 0 0" xyz="0 -.002 0.072" />
        <parent link="braccio_base_link" />
        <child link="shoulder_link" />
    </joint>

    <joint name="elbow_joint" type="revolute">
        <axis xyz="1 0 0" />
         <limit effort="1000.0" lower="0" upper="3.1416" velocity="1.0"
/>
        <origin rpy="-1.5708 0 0" xyz="0 0 0.125" />
        <parent link="shoulder_link" />
        <child link="elbow_link" />
    </joint>

    <joint name="wrist_pitch_joint" type="revolute">
        <axis xyz="1 0 0" />
         <limit effort="1000.0" lower="0" upper="3.1416" velocity="1.0"
/>
        <origin rpy="-1.5708 0 0" xyz="0 0 0.125" />
        <parent link="elbow_link" />
        <child link="wrist_pitch_link" />
    </joint>

    <joint name="wrist_roll_joint" type="revolute">
        <axis xyz="0 0 -1" />
                  <limit  effort="1000.0"  lower="0.0"  upper="3.1416"
velocity="1.0" />
        <origin rpy="0 0 1.5708" xyz="0 0.0 0.06" />
        <parent link="wrist_pitch_link" />
        <child link="wrist_roll_link" />
    </joint>

    <joint name="gripper_joint" type="revolute">
        <axis xyz="0 -1 0" />
                <limit  effort="1000.0"  lower="0.1750"  upper="1.2741"
velocity="1.0" />
        <origin rpy="0 -0.2967 0" xyz="0.010 0 0.03" />
        <parent link="wrist_roll_link" />
        <child link="right_gripper_link" />
```

```
    </joint>

    <joint name="sub_gripper_joint" type="revolute">
        <axis xyz="0 1 0" />
        <mimic joint="gripper_joint" />
                <limit  effort="1000.0"  lower="1.2741"  upper="2.3732"
velocity="1.0" />
        <origin rpy="0 3.4383 0" xyz="-0.010 0 0.03" />
        <parent link="wrist_roll_link" />
        <child link="left_gripper_link" />
    </joint>
```

Hay que señalar que los valores del atributo de *effort* están expresados en *newtons* (N) y la velocidad (*velocity*) en m/s. Los valores de los atributos *lower* y *upper*, expresados en radianes, pueden obtenerse de la instrucciones del *Tinkerkit Braccio,* donde se especifica que los servomotores:

-M1 (valores permitidos en grados)=0°-180°, lo que coincide con los valores de los atributos del joint "base_joint": lower="0.0" upper="3.1416" expresados en radianes.

-M2 (valores permitidos en grados)=15°-165°, lo que coincide con los valores de los atributos del joint "shoulder_joint": lower="0.2618" upper="2.8798" expresados en radianes.

-M3 (valores permitidos en grados)=0°-180°, lo que coincide con los valores de los atributos del joint "elbow_joint": lower="0.0" upper="3.1416" expresados en radianes.

-M4 (valores permitidos en grados)=0°-180°, lo que coincide con los valores de los atributos del joint "wrist_pitch_joint": lower="0.0" upper="3.1416" expresados en radianes.

-M5 (valores permitidos en grados)=0°-180°, lo que coincide con los valores de los atributos del joint "wrist_roll_joint": lower="0.0" upper="3.1416" expresados en radianes.

-M6 (valores permitidos en grados)=10°-73°, lo que coincide con los valores de los atributos del joint "gripper_joint": lower="0.1750" upper="1.2741", y del joint "sub_gripper_joint": lower="1.2741" upper="2.3732" expresados en radianes.

El archivo XML braccio_arm.urdf termina con la traducción numérica de los colores naranja, blanco y negro al modelo RGB (***R**ed, **G**reen, **B**lue*).

```
<material name="orange">
        <color rgba="0.57 0.17 0.0 1" />
    </material>
    <material name="white">
        <color rgba="0.8 0.8 0.8 1.0" />
    </material>
    <material name="black">
        <color rgba="0 0 0 0.50" />
    </material>

</robot>
```

Con el comando *check_urdf* se puede comprobar si la sintaxis del fichero es correcta y se presenta un resumen del modelo creado, en el nombre del archivo se debe incluir la extensión .urdf correspondiente.

```
$ check_urdf braccio_arm.urdf

robot name is: braccio
---------- Successfully Parsed XML ---------------
root Link: base_link has 1 child(ren)
    child(1):  braccio_base_link
        child(1):  shoulder_link
            child(1):  elbow_link
                child(1):  wrist_pitch_link
                    child(1):  wrist_roll_link
                        child(1):  right_gripper_link
                        child(2):  left_gripper_link
```

También se puede usar el comando *urdf_to_graphiz* para tener una representación gráfica más detallada creando un archivo .pdf (Figura 31 y 32).

```
$ urdf_to_graphiz braccio_arm.urdf

Created file braccio.gv
Created file braccio.pdf
```

En la Figura 33 vemos que el árbol cinemático del robot se bifurca en dos ramas que comparten un mismo *link* padre (“*wrist_roll_link*”). La rama izquierda se extiende a través del *joint* “*gripper_joint*” y el *link* “*right_gripper_link*”, mientras que la rama derecha lo hace a través del joint “*sub_gripper_joint*” y el *link* “*left_gripper_link*”. Si el *end-effector* (o efector final: último enlace de la cadena cinemática; puede ser una pinza, una herramienta para soldar, etc.) del manipulador fuera la mano de un humanoide (cinco dedos) tendría más sentido describir ese mismo árbol con cinco ramas en lugar de dos. Un archivo URDF puede representar cualquier estructura de árbol, pero nunca una cadena cinemática cerrada (Figura 31, a)).

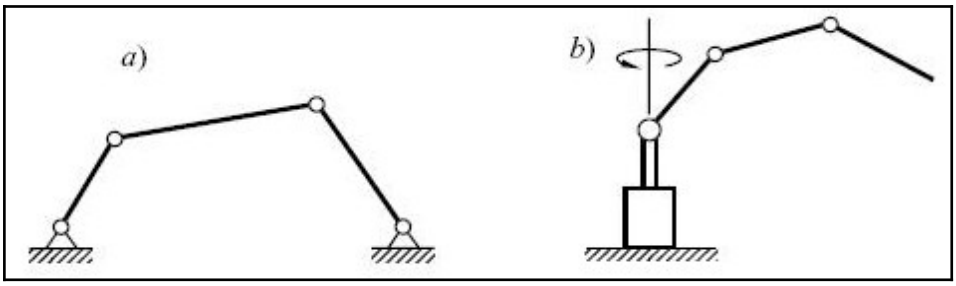


Figura 31. a) Cadena cinemática cerrada y b) abierta. Fuente: [17]

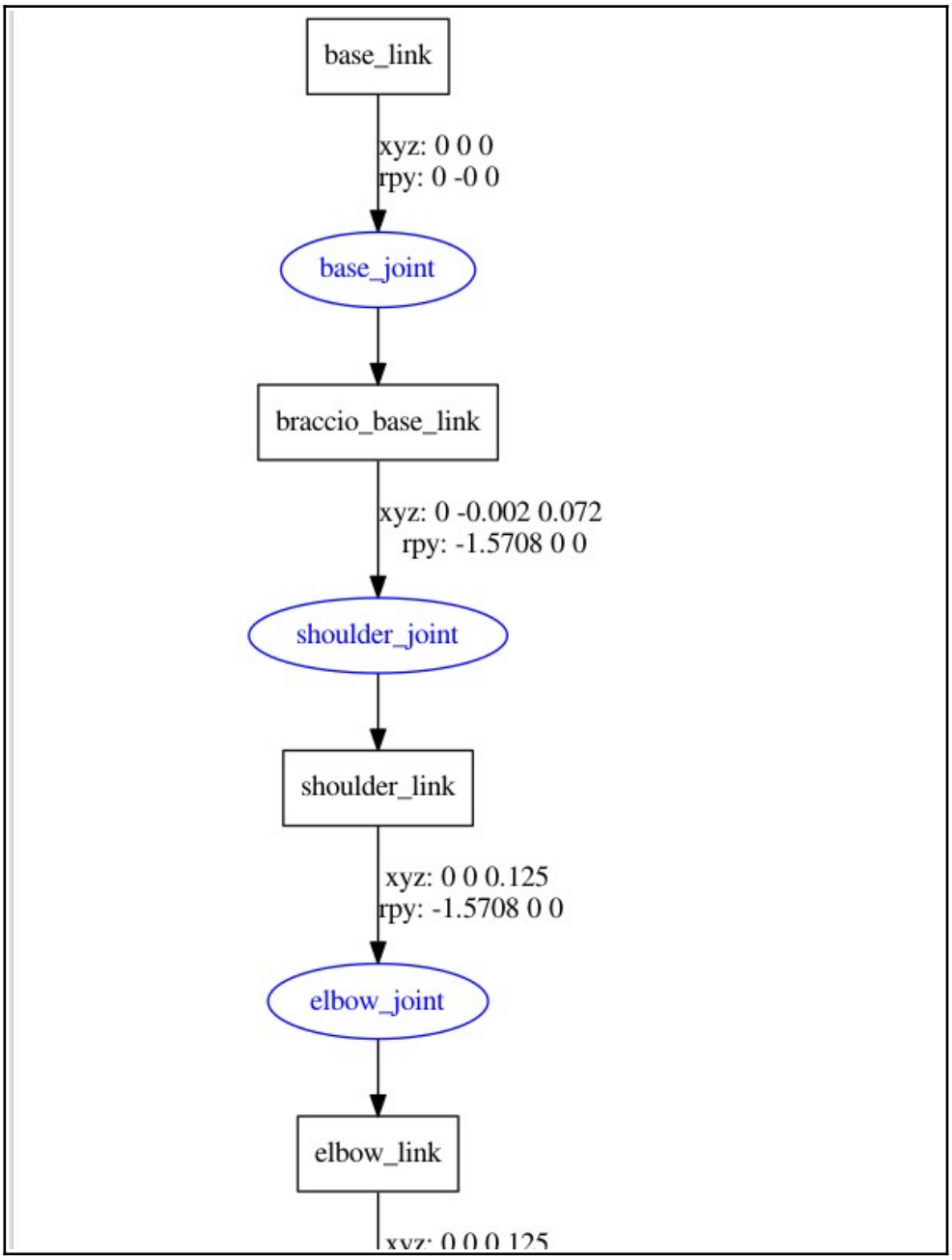


Figura 32. Archivo braccio.pdf. Fuente: [16]

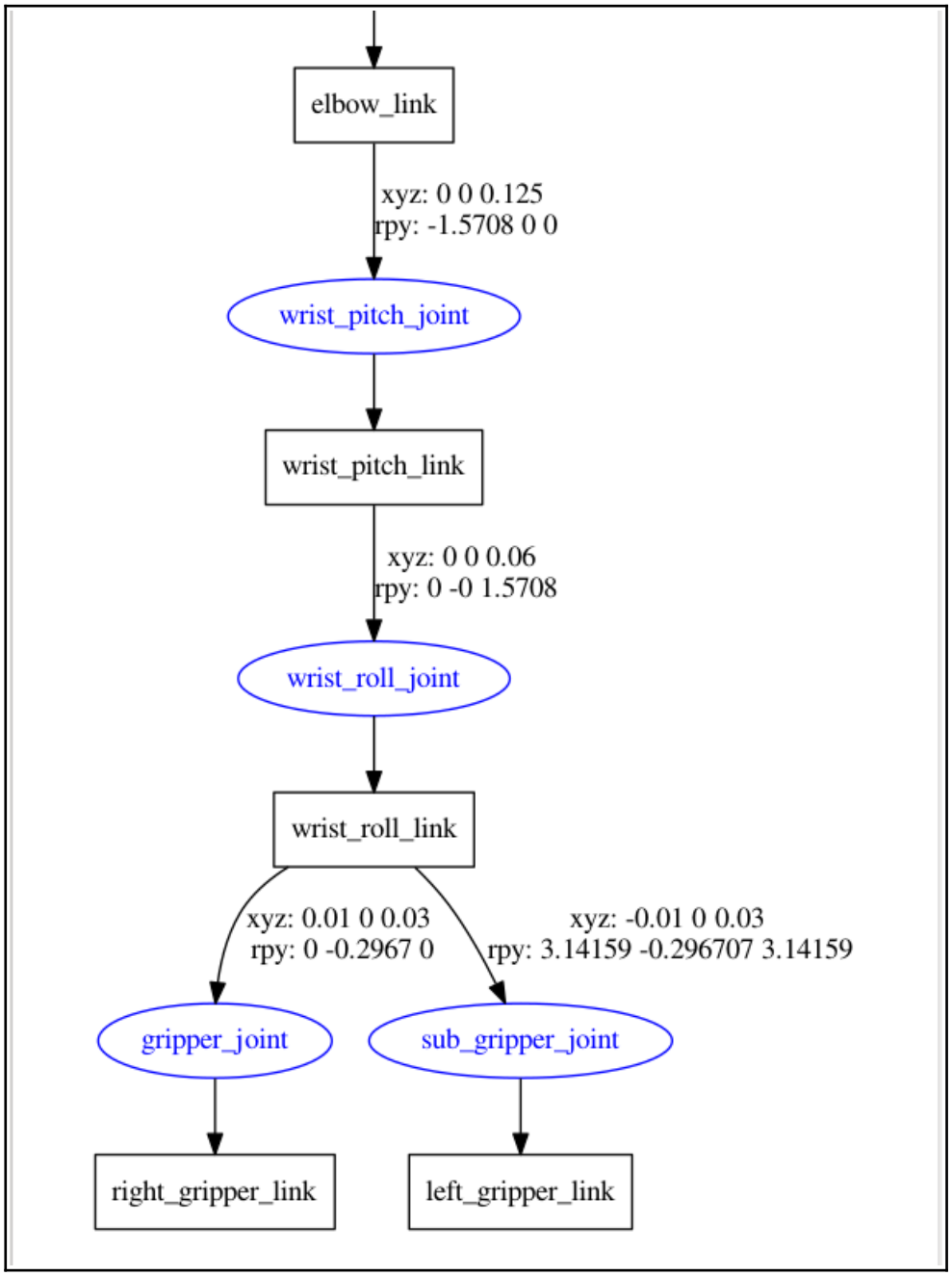


Figura 33. Archivo braccio.pdf. Fuente: [16]

Ahora que tenemos el modelo del robot podemos usar el visualizador 3D *RViz* para ver los movimientos de las articulaciones. Seguidamente pasamos a analizar la carpeta *launch* donde se encuentra el archivo *urdf.launch:*

```
<launch>


      <arg name="model" default="$(find
braccio_arduino_ros_rviz)/urdf/braccio_arm.urdf"/>

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

      <arg name="rvizconfig" default="$(find
braccio_arduino_ros_rviz)/rviz/urdf.rviz" />

    <param name="robot_description" command="$(find xacro)/xacro.py $
(arg model)"/>

    <param name="use_gui" value="gui"/>

    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">

            <param name="use_gui" value="gui"/>
       </node>

    <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />

   <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg
rvizconfig)" required="true" />

</launch>
```

En el código XML mostrado anteriormente se establecen algunos argumentos y parámetros de configuración relacionados con el visualizador Rviz y algunos nodos. El nodo *“joint_state_publisher”,* por ejemplo, que publica los valores de cada articulación (ángulos en radianes) en el formato de mensaje *sensor_msgs/JointState*. Si establecemos el argumento *<arg name="gui" default="true"/>* como verdadero (“*true”*) el nodo *“joint_state_publisher”* muestra una ventana de control deslizante (Figura 34) para que el usuario pueda mover independientemente cada articulación, hacer un movimiento conjunto aleatorio o reiniciar las posiciones. Los valores inferior y superior de los ángulos (lower/upper) se toman de la etiqueta del límite del modelo URDF (braccio_arm.urdf). Al ejecutar el comando *roslaunch braccio_arduino_ros_rviz urdf.launch model:=urdf/braccio_arm.urdf* se inicializan el visualizador Rviz (nodo “rviz“), el nodo ROS Master, el nodo "joint_state_publisher" y el nodo "robot_state_publisher" de forma simultánea y coordinada, obteniéndose el siguiente resultado:

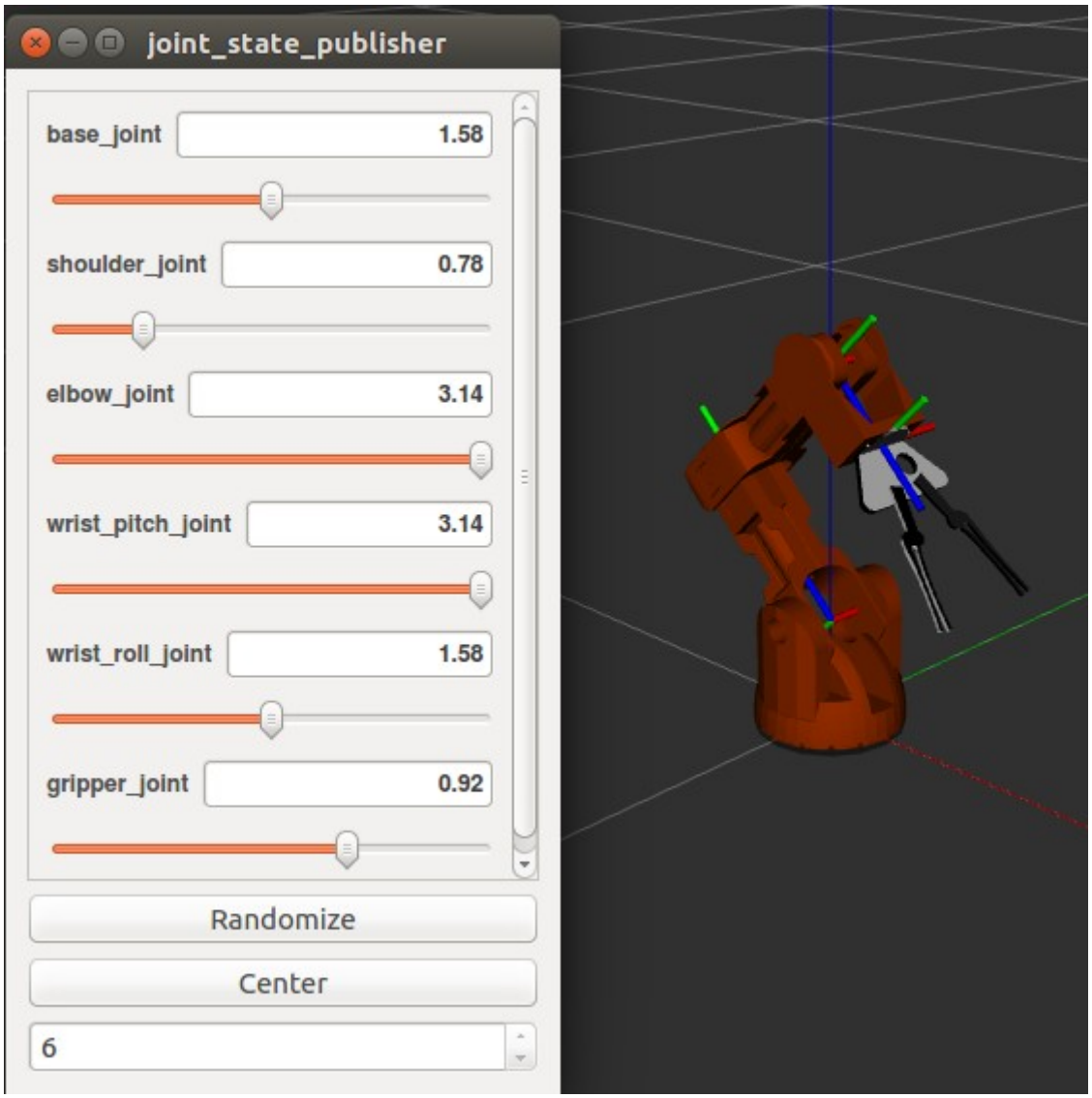


Figura 34. Captura de pantalla recortada de RViz y la GUI del robot. Fuente: elaboración propia

En la Figura 35 puede observarse al robot en la misma posición "safety" estudiada anteriormente, pero a diferencia de lo ocurrido con la *Toolbox Robotics* para *Matlab*, los ángulos introducidos en la GUI *"joint_state_publisher"* coinciden plenamente con los ángulos que las instrucciones del Tinkerkit Braccio especifica para esa pose inicial con los servomotores M1=90°, M2=45°, M3=180°, M4=180° y M5=90°.

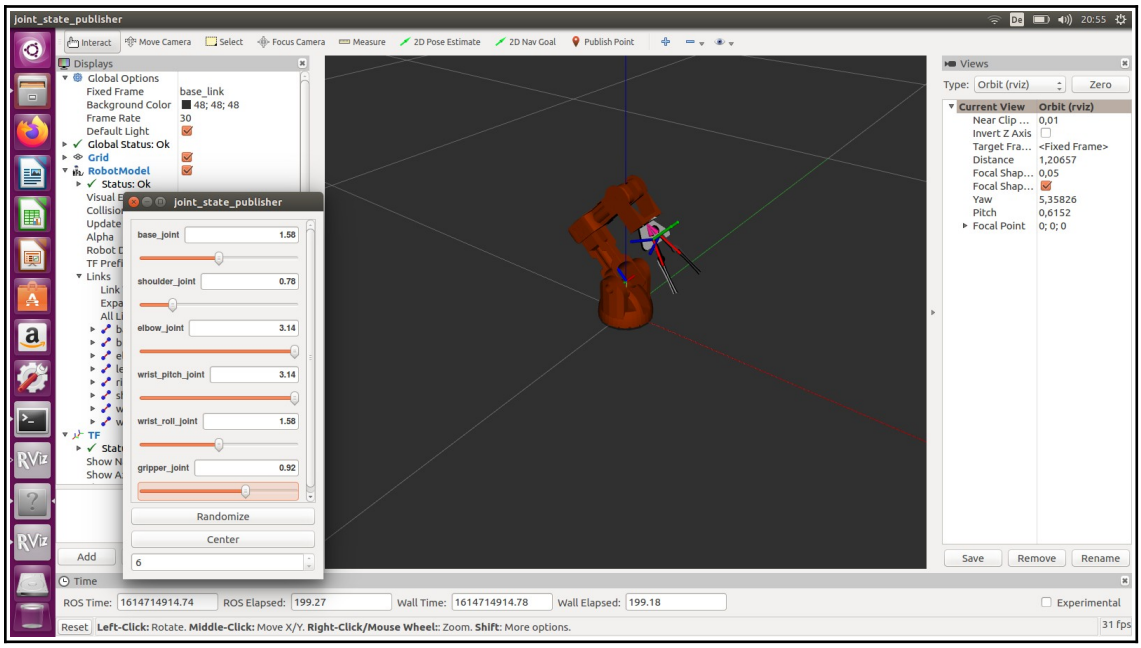


Figura 35. Captura de pantalla completa de RViz y la GUI del robot. Fuente: elaboración propia

El nodo "robot_state_publisher" merece un comentario aparte, puesto que es el encargado de publicar el estado, en conjunto, del modelo del robot a través del tema o tópico (*topic*) /tf. Este tópico se nutre del cálculo computacional procesado por una librería del mismo nombre (*tf library*), que realiza un seguimiento de todos los sistemas de referencia del robot a lo largo del tiempo.

Introduciendo el comando *rostopic info tf* obtenemos información acerca del tópico en cuestión: [18]

```
$ rostopic info tf

Type: tf2_msgs/TFMessage

Publishers:
 * /robot_state_publisher (http://euclides-SATELLITE-C70D-A:40881/)

Subscribers:
 * /rviz (http://euclides-SATELLITE-C70D-A:41587/)
```

Lo que nos indica que el tópico tf lleva mensajes de tipo *tf2_msgs / TFMessage.* Si mostramos su contenido puede apreciarse la estructura del mensaje:

```
$ rosmsg show tf2_msgs/TFMessage

geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w
```

Este mensaje contiene un vector (*array* de longitud variable) de mensajes de tipo *geometry_msgs / TransformStamped. TransformStamped.*

El tipo de datos de transformación ROS no es el mismo que una matriz de transformación homogénea 4x4, pero lleva información equivalente. El tipo de datos de la transformada ROS contiene un vector 3-D (equivalente a la cuarta columna de una transformada 4 × 4) y un cuaternión (una representación alternativa de la orientación). Además, los mensajes de transformación ROS tienen marcas de tiempo (*time stamps*) y nombran explícitamente el *frame* hijo y el *frame* padre como cadenas de texto.

Con el comando rostopic hz tf vemos que el tópico se actualiza (vuelve a publicarse) cada 10 Hz = 0.1 segundos (100 ms) aproximadamente.

```
$ rostopic hz tf
subscribed to [/tf]
average rate: 10.001
      min: 0.100s max: 0.100s std dev: 0.00022s window: 10
average rate: 10.000
      min: 0.097s max: 0.103s std dev: 0.00088s window: 20

...
```

Examinando el componente de transformaciones tf (array de longitud variable) comprobamos que el mensaje incluye múltiples relaciones de transformaciones individuales. Un extracto de la salida del comando *rostopic eco tf* es:

```
$ rostopic echo tf
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1614762839
        nsecs:  86061000
      frame_id: "base_link"
    child_frame_id: "braccio_base_link"
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
...
```

Como hemos comentado anteriormente en ROS las orientaciones se expresan generalmente como unidades de cuaterniones, que es una representación alternativa (más compacta) a las matrices de rotación 3x3. Existe una correspondencia entre los cuaterniones y las matrices de rotación (funciones ROS para realizar tales conversiones), y existen también las operaciones matemáticas correspondientes para las transformaciones de coordenadas con cuaterniones. No obstante, no insistiremos más sobre este interesante aspecto matemático, ya que puede alargar en exceso el contenido del capítulo.

Un último comando que resulta de gran utilidad es *rostopic echo joint_states,* el cual nos proporciona información (tiempo actualizado) sobre el estado de todas las articulaciones expresado en radianes.

```
$ rostopic echo joint_states
header:
  seq: 8423
  stamp:
    secs: 1614763348
    nsecs: 586009979
  frame_id: ''
name:  [base_joint,  shoulder_joint,  elbow_joint,  wrist_pitch_joint,
wrist_roll_joint, gripper_joint,
  sub_gripper_joint]
position: [0.0, 1.5708, 0.0, 0.0, 0.0, 0.72455, 0.72455]
velocity: []
effort: []
---
...
```

Mientras estábamos redactando el TFG pudimos conocer e instalar la nueva versión *Robotics System Toolbox* de Peter Corke. Entre las nuevas funcionalidades que nos ofrece esta herramienta, está la posibilidad de importar un archivo .urdf. Para lo cual, simplemente tendríamos que descargar la carpeta STL (archivos .stl) y el archivo braccio_arm.urdf en el directorio de Matlab en el que estemos trabajando. La serie de comandos y el resultado obtenido es la siguiente:

```
>> robot=importrobot('braccio_arm.urdf');
>> config(2).JointPosition = pi/4;
>> show(robot, config);
>> config(1).JointPosition = pi/2;
>> config(2).JointPosition = pi/4;
>> config(3).JointPosition = pi;
>> config(4).JointPosition = pi;
>> config(5).JointPosition = pi/2;
>> show(robot, config);
```

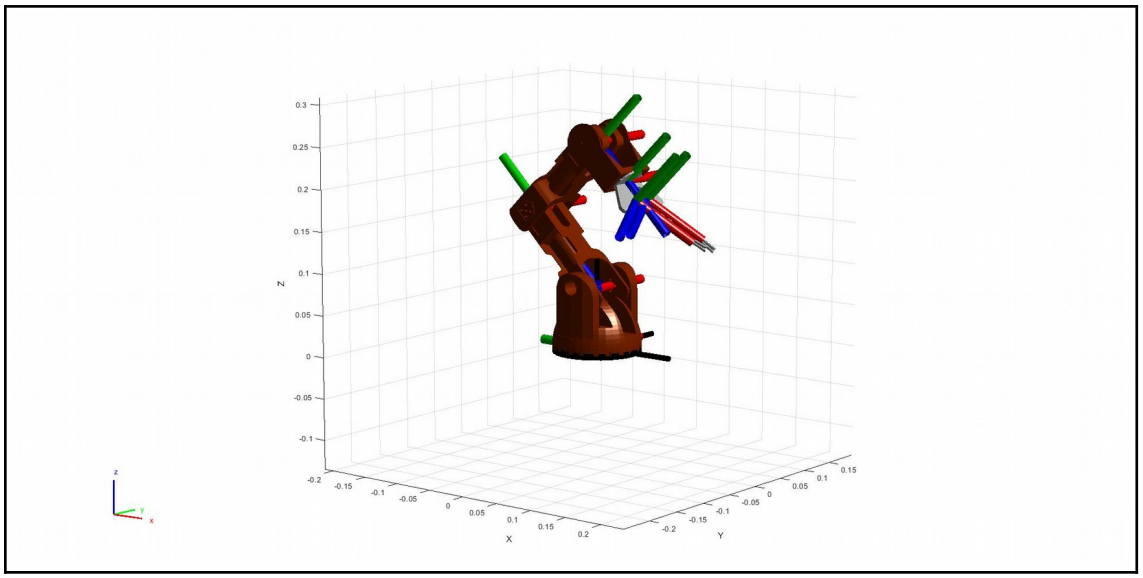


Figura 36. Posición «*safety*» en la *Robotics System Toolbox* . Fuente: elaboración propia

En la Figura 36 también puede observarse al robot en la misma posición "safety" estudiada anteriormente, pero a diferencia de lo ocurrido con la *Toolbox Robotics* para *Matlab*, los ángulos introducidos en la nueva version *Robotics System Toolbox* coinciden plenamente con los ángulos que las instrucciones del Tinkerkit Braccio especifica para esa pose inicial con los servomotores M1=90°, M2=45°, M3=180°, M4=180° y M5=90°.

En este capítulo hemos visto dos formas muy distintas de encarar la cinemática directa del robot. Los parámetros Denavit-Hartenberg, representan una versión un tanto minimalista y eficiente, que con solo 4 parámetros y algo de práctica y pericia en la asignación de los *frames* y *joints* correspondientes, es capaz de describir completamente la cinemática del manipulador.

Por otro lado, los parámetros URDF, junto con los paquetes ROS, librerías, y visualizador 3D Rviz suponen un grado de complejidad importante, pero también se automatiza mucho trabajo inicial por medio de archivos XML, archivos de configuración CMakeLists.txt, etc. Hay que resaltar que con ROS no estamos construyendo un modelo cinemático solamente, estamos construyendo toda una pieza de software para un robot determinado sin tener que partir de cero, en un tiempo récord y con un resultado final impresionante.

6. Cinemática inversa: MoveIt!

En el capítulo anterior hemos estudiado la cinemática directa, donde conocidos ciertos valores de las articulaciones q = (q_1, q_2, … q_n), se calculaba la posición y la orientación del extremo del robot TCP = (x,y,z,a,b,c).

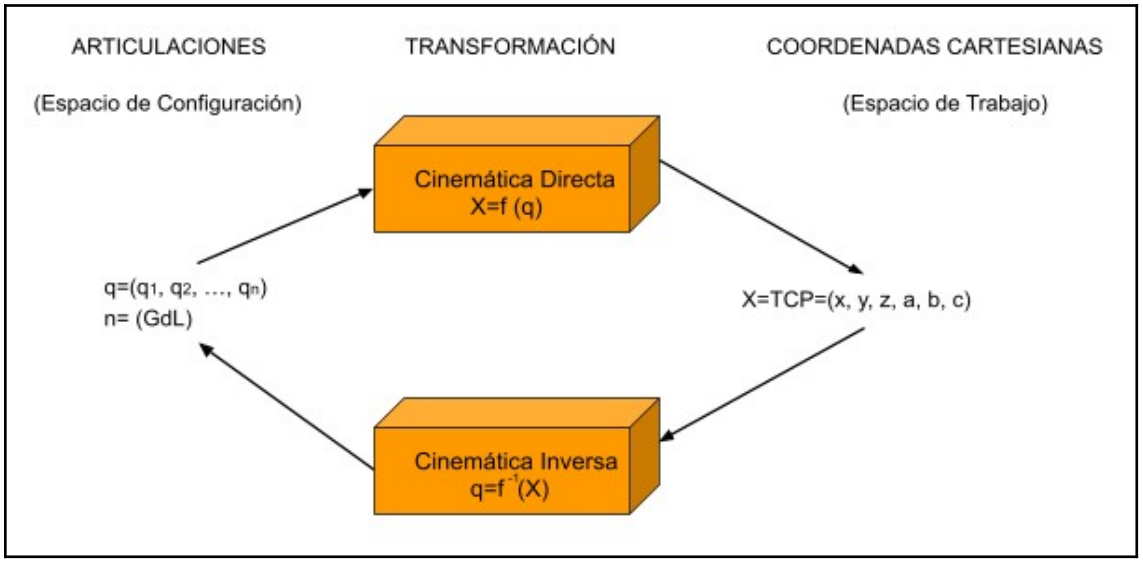


Figura 37. Cinemática Directa e Inversa. Fuente: elaboración propia

Ahora pasamos a estudiar la cinemática inversa (Figura 37), donde conocidos ciertos valores de la posición y la orientación del extremo del robot TCP = (x,y,z,a,b,c), se calculan las coordenadas articulares del robot q = (q_1, q_2, … q_n).

Recordemos que el problema cinemático directo se podía resolver de forma sistemática, independientemente de la configuración del robot, obteniéndose una única solución. En el problema cinemático inverso, por el contrario, la solución (si la hay) no siempre es única, existiendo diferentes combinaciones de articulaciones que posicionan y orientan el extremo del robot del mismo modo. [11]

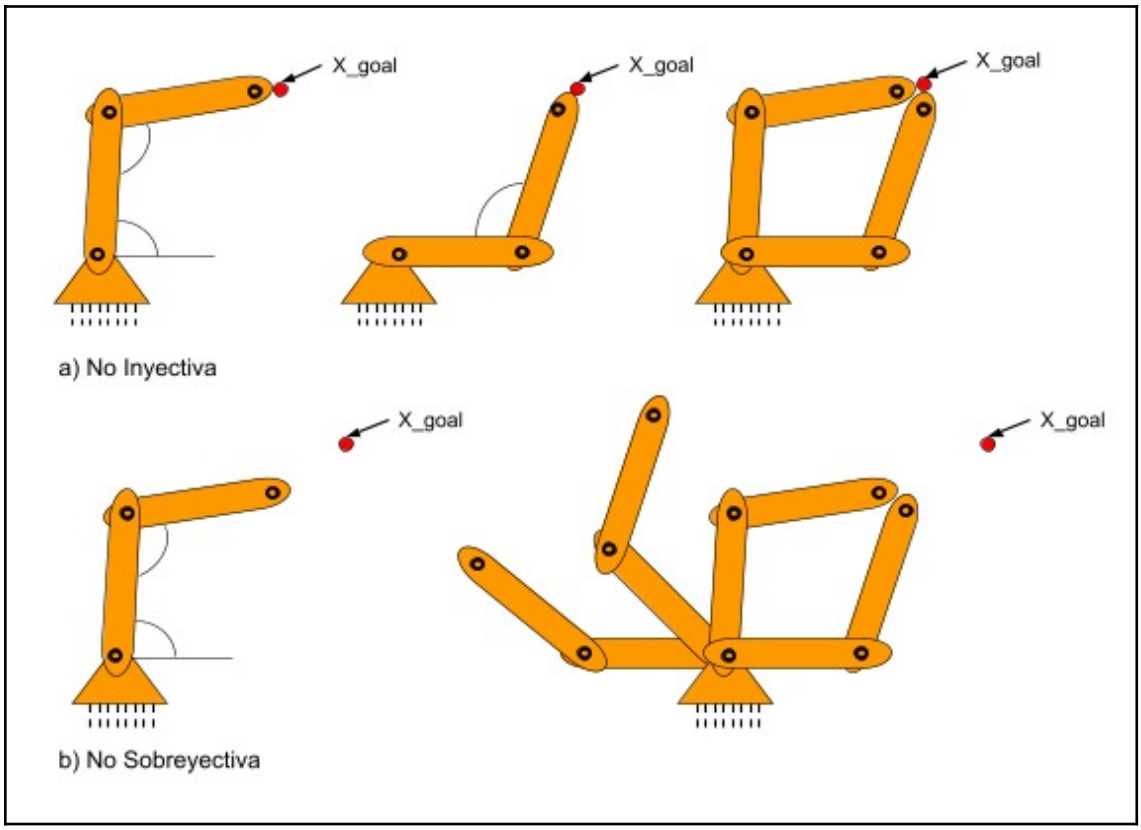


Figura 38. a) Función no inyectiva y b) Función no sobreyectiva. Fuente: elaboración propia

Si definimos la cinemática directa como una función X=f(q), y la cinemática inversa como su función inversa $q=f^{-1}(X)$, podemos encontrar una relación matemática y afirmar que f^{-1} existe; si y solo si, f es biyectiva (inyectiva y sobreyectiva). En general, la función f de cinemática directa no es biyectiva.

En el apartado a) de la Figura 38 se observa como el manipulador puede alcanzar el objetivo (X_goal) mediante dos configuraciones distintas de ángulos articulares. Es decir, que a dos elementos distintos del conjunto inicial (dominio) **no** les corresponden dos elementos distintos en el conjunto final (función no inyectiva). En el apartado b), por el contrario, se observa como el manipulador no puede alcanzar el objetivo (X_goal) mediante ninguna de las configuraciones posibles. Es decir, que existe un elemento del conjunto final que **no** es la imagen de un elemento del conjunto inicial (función no sobreyectiva).

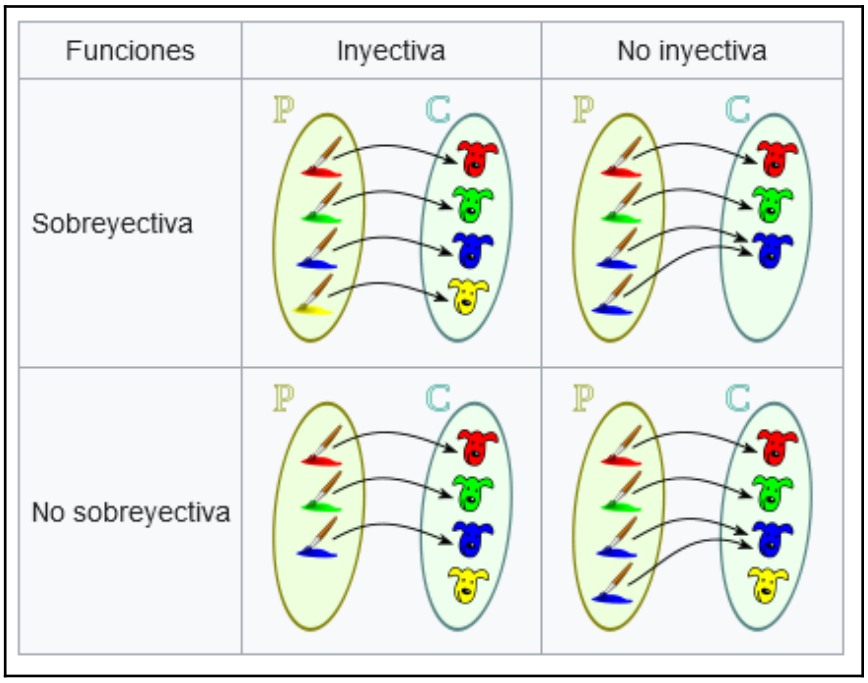


Figura 39. Diagrama de grafos bipartitos. Fuente: [19]

En la Figura 38 se realizaba el esquema de un robot con solo dos grados de libertad, podemos imaginar una situación más compleja con otros robots con un mayor número de grados de libertad: 3, 4, 5, 6 o 7 (para un manipulador normal), o muchos más para un robot humanoide, por ejemplo.

No vamos a seguir insistiendo en la matemática que hay detrás de la cinemática inversa, puesto que tenemos una herramienta muy potente a nuestra disposición que hace todo ese cálculo computacional.

MoveIt! es una librería de herramientas para la planificación y el control del movimiento del robot. *MoveIt!* proporciona el plan que deben seguir las articulaciones para mover un brazo de una posición a otra sin entrar en colisión consigo mismo o con algún objeto externo. Además, *MoveIt!* dispone de un asistente de configuración con una atractiva GUI que facilita mucho la tarea inicial, generando automáticamente todos los archivos y elementos necesarios para la posterior visualización en Rviz.

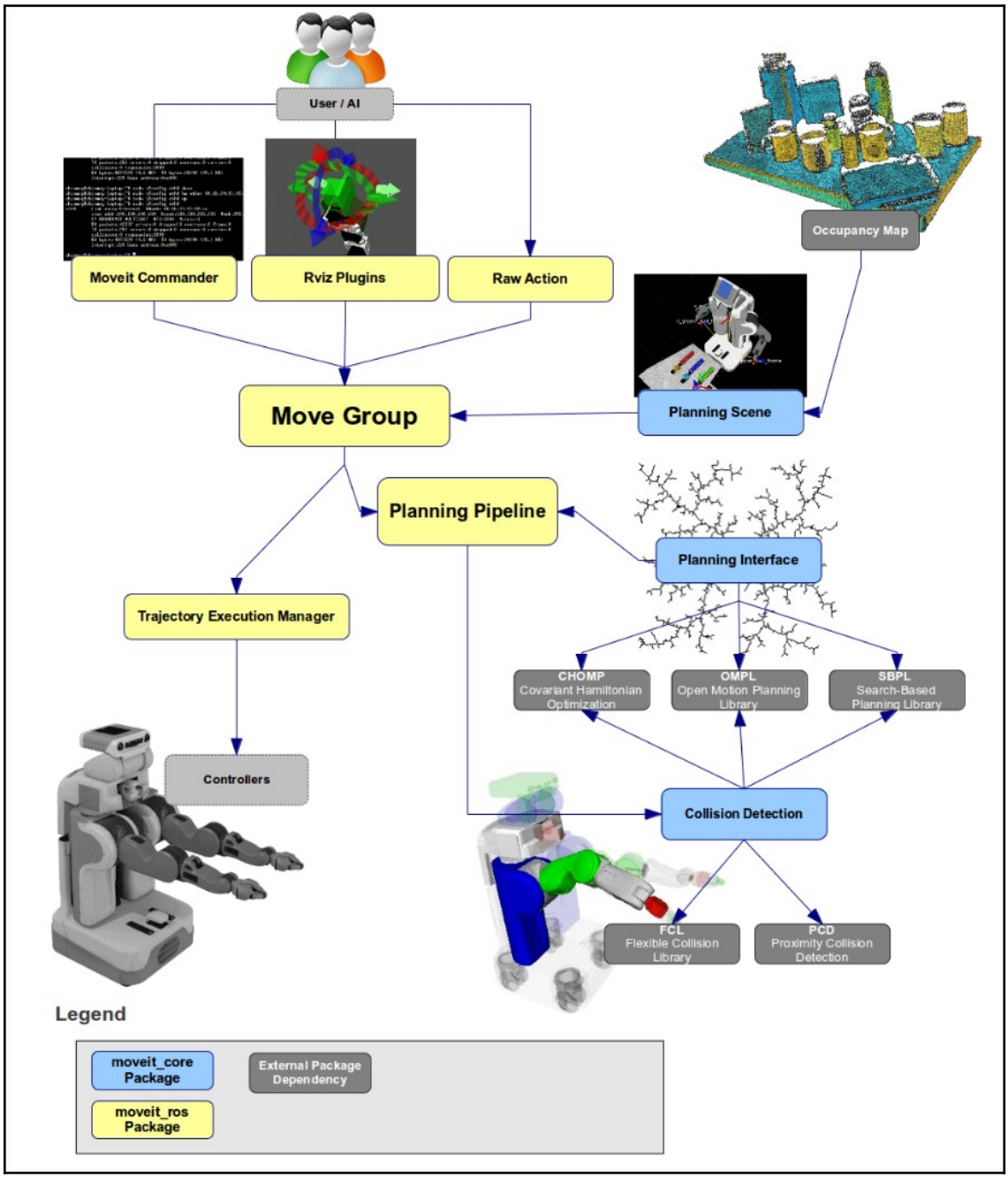


Figura 40. Arquitectura del Sistema (diagrama de alto nivel) . Fuente: [20]

Una vez hayamos instalado *MoveIt! p*odemos usar el *moveit_setup_assistan*t mediante el comando:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

Luego el *moveit_setup_assistant* debe localizar el directorio donde se ha creado el modelo URDF, y cargarlo en la aplicación. [21]

Figura 41. *Moveit_Setup_Assistant*: ventana inicial. Fuente: elaboración propia

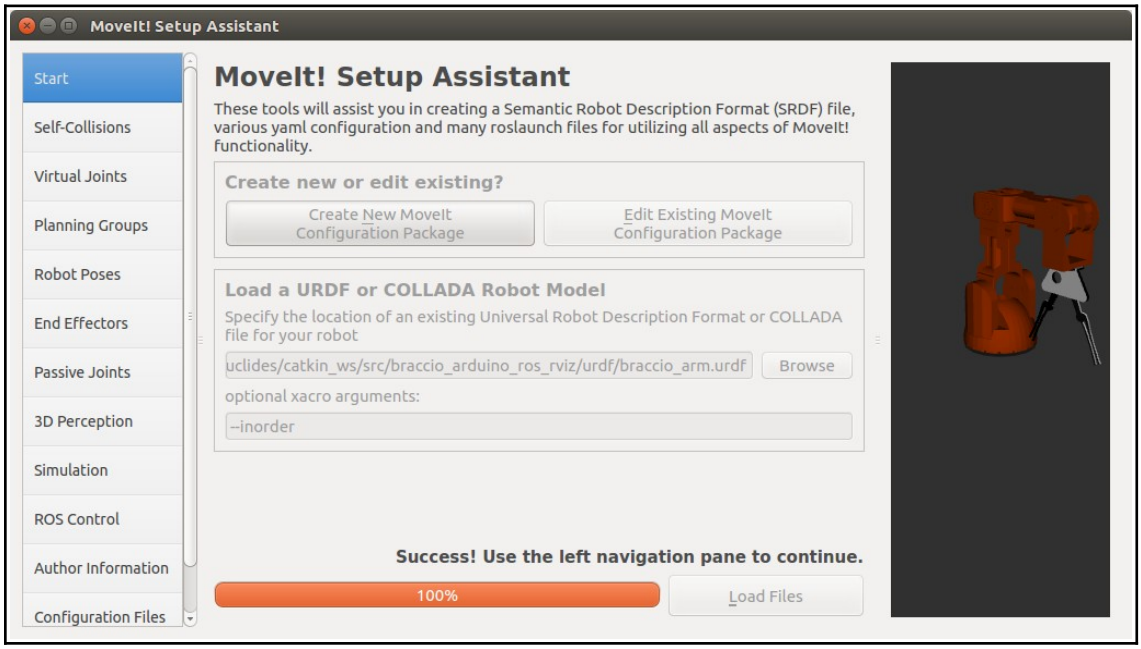


Figura 42. *Moveit_Setup_Assistant*: carga del fichero URDF. Fuente: elaboración propia

El siguiente paso (Figura 43) se denomina la *Self-Collision Checking, que* consiste básicamente, en leer todos los pares de enlaces del modelo URDF examinando los enlaces en el robot que puedan desactivarse de manera segura desde la verificación de colisiones, lo que reduce el tiempo de procesamiento de la planificación del movimiento. Estos pares de eslabones se desactivan cuando siempre están en colisión, cuando nunca están en colisión,

cuando están en colisión en la posición predeterminada del robot, o cuando los eslabones están adyacentes entre sí en la cadena cinemática. La barra deslizante de densidad de muestreo (*Sampling Density*) especifica cuántas posiciones aleatorias del robot se deben verificar para detectar una auto-colisión. Las densidades más altas requieren más tiempo de cálculo, mientras que las densidades más bajas tienen una mayor posibilidad de deshabilitar pares que no deberían deshabilitarse. El valor predeterminado es 10000 controles de colisión, que es el que aparece en la imagen (Figura 43). La comprobación de colisiones se realiza en paralelo para reducir el tiempo de procesamiento.

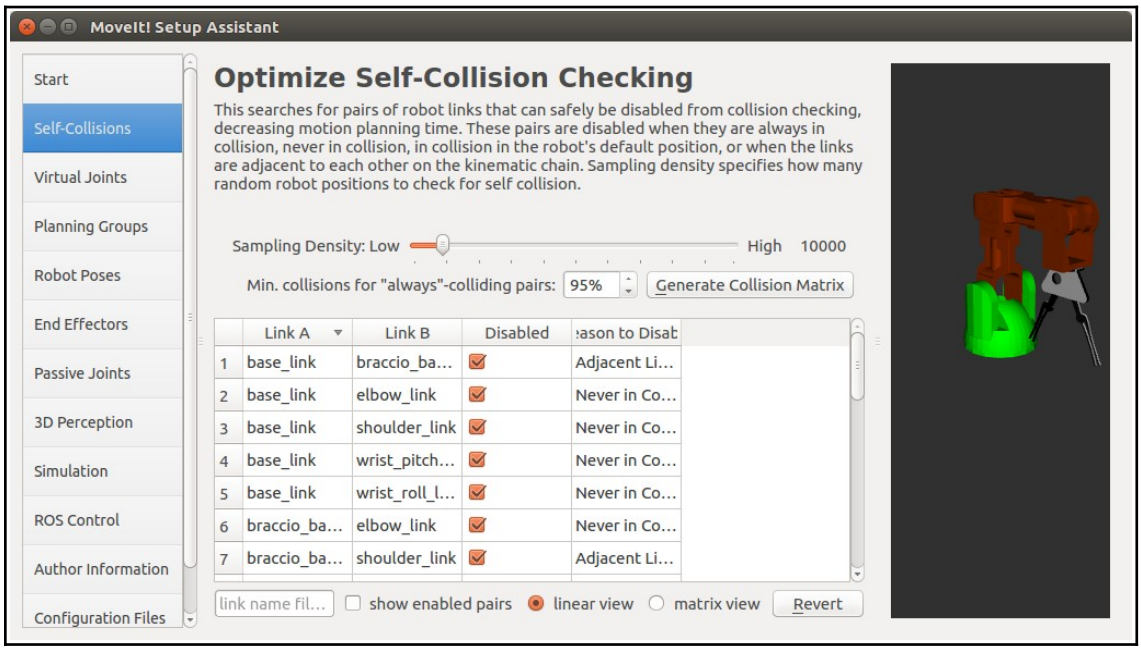


Figura 43. *Moveit_Setup_Assistant: Self-Collision Checking.* Fuente: elaboración propia

Ahora es el turno de añadir una articulación virtual (Figura 44). Las articulaciones virtuales se utilizan principalmente para conectar el robot al mundo. Para el *Tinkertkit Braccio* definiremos solo una articulación virtual adjuntando el enlace *base_link* del mismo al *frame* común *world*. Esta articulación virtual representa el movimiento de la base del robot en un plano (Joint Type: planar).

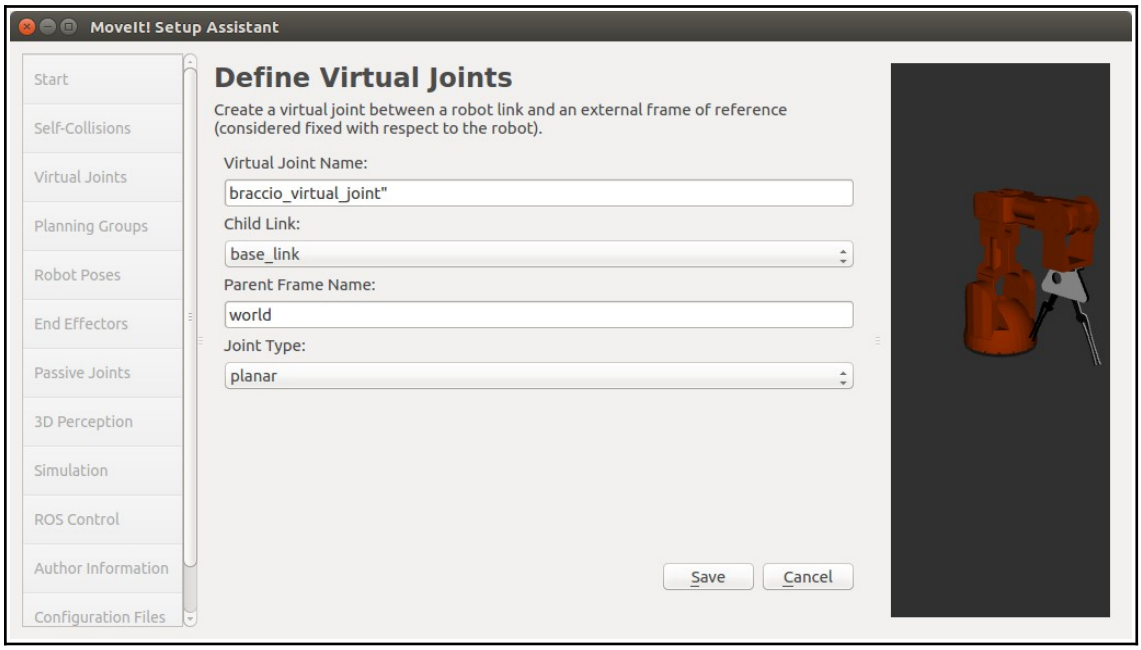


Figura 44. *Moveit_Setup_Assistant: Define Virtual Joints*. Fuente: elaboración propia

Los grupos de planificación (*planning groups*) se utilizan para describir semánticamente diferentes partes de su robot, como definir qué es un brazo o un efector final.

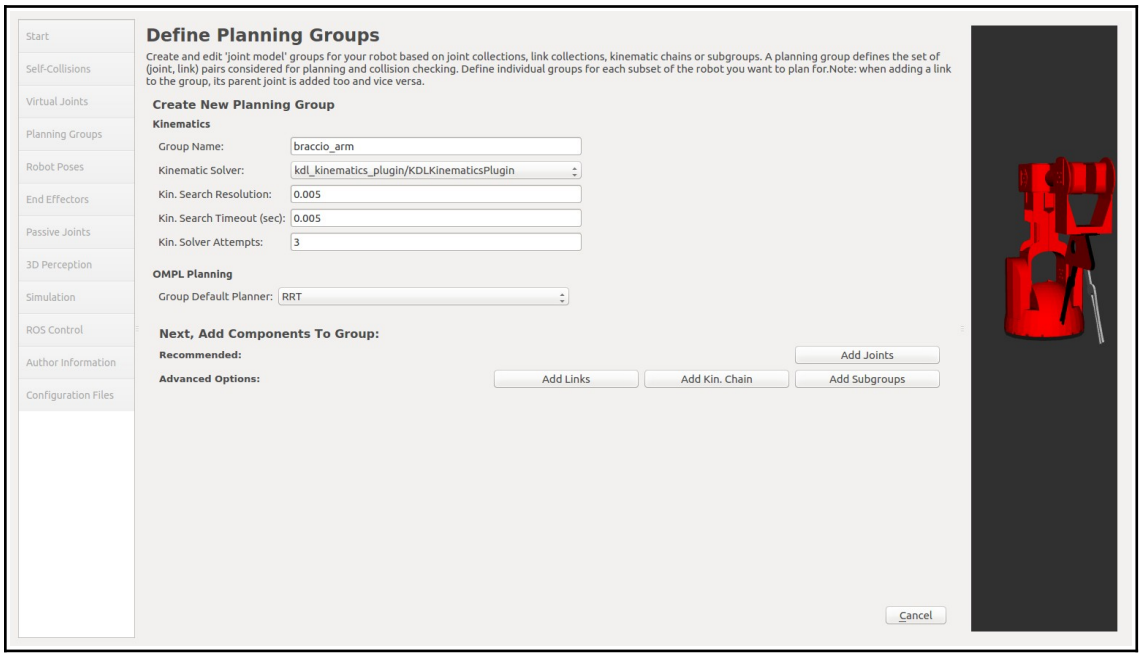


Figura 45. *Moveit_Setup_Assistant: Define Planning Groups*. Fuente: elaboración propia

Este paso es decisivo, puesto que según generemos dichos grupos vamos a tener modelos cinemáticos distintos. En nuestro caso, siguiendo las

recomendaciones generales para los manipuladores ([21]), hemos creado dos grupos distintos. Un primer grupo denominado *braccio_arm* (Figura 45) formado por los *joints* (Figura 46) *base_joint, shoulder_joint, elbow_joint, wrist_pitch_joint* y *wrist_roll_joint.* La opción de cálculo cinemático recomendada es ***kdl_kinematics_plugin/KDLKinematicsPlugin***, mientras que para el planificador OMP marcamos la opción **RRT**.

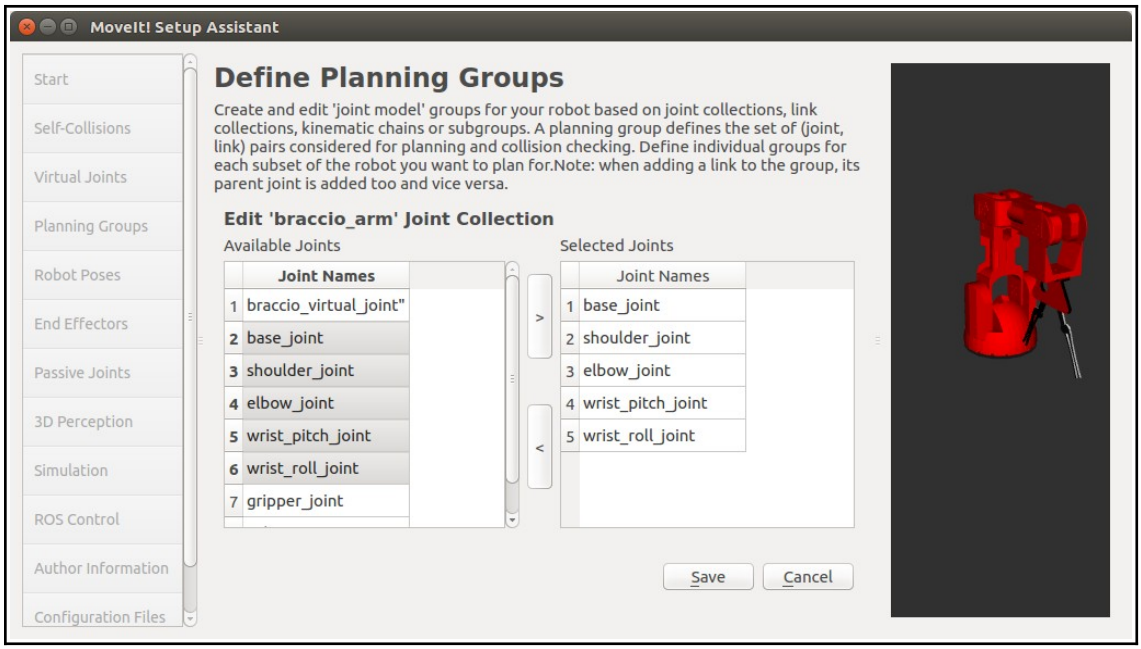


Figura 46. ***Moveit_Setup_Assistant: Define Planning Groups*****. Fuente: elaboración propia**

El segundo grupo se llama *braccio_gripper* (Figura 47), está formado por los *joints gripper_joint y sub_gripper_joint.* Esta vez no es necesario configurar las opciones de cinemática y de planificación de movimientos, puesto que se trata de un extremo del brazo del robot con solamente 2 joints (que se implementan con el mismo actuador de cierre y apertura de la pinza). En la Figura 48 se hace un listado de *joints* de ambos grupos.

El asistente de configuración también permite agregar ciertas poses fijas predeterminadas. Esto ayuda si, por ejemplo, se desea definir una determinada posición del robot como posición inicial o final; o volver a esa posición después de hacer otros movimientos. No obstante, no resulta muy interesante en cuanto a resolver la cinemática inversa (ángulos del espacio de configuración dados).

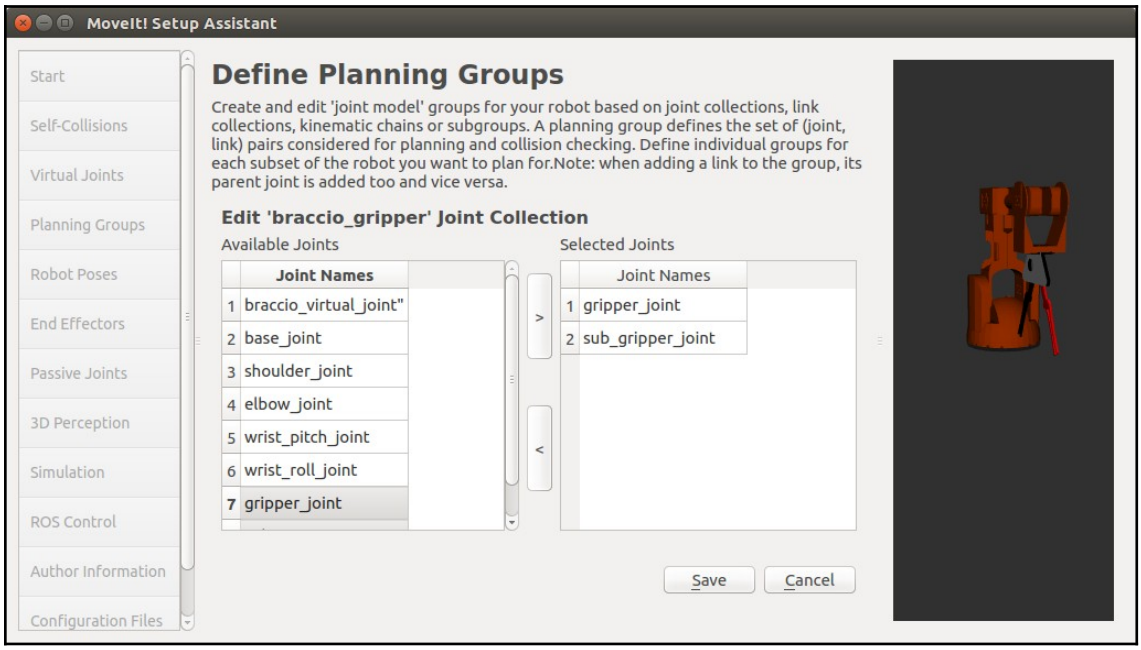


Figura 47. *Moveit_Setup_Assistant: Define Planning Groups*. Fuente: elaboración propia

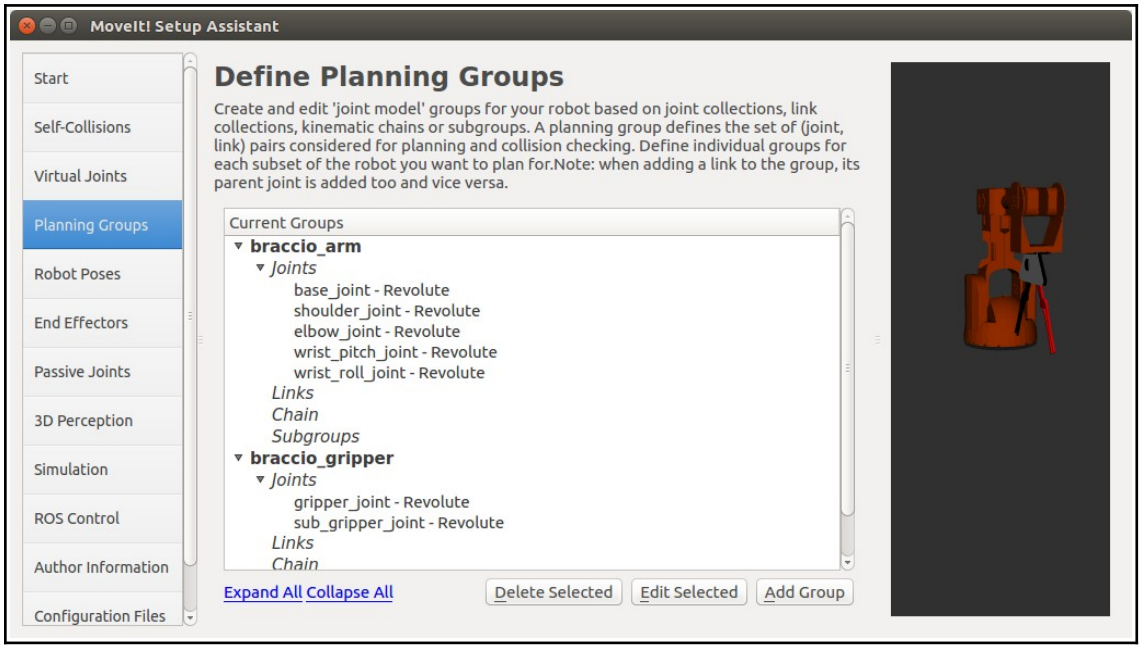


Figura 48. *Moveit_Setup_Assistant: Define Planning Groups*. Fuente: elaboración propia

Para el manipulador Braccio, y siguiendo con la tónica de este proyecto, hemos empezado por definir la pose *braccio_safety* (Figura 49). Los ángulos de los *joints* son los ya conocidos por el lector [90°, 45°, 180°, 180°, 90°] expresados en radianes.

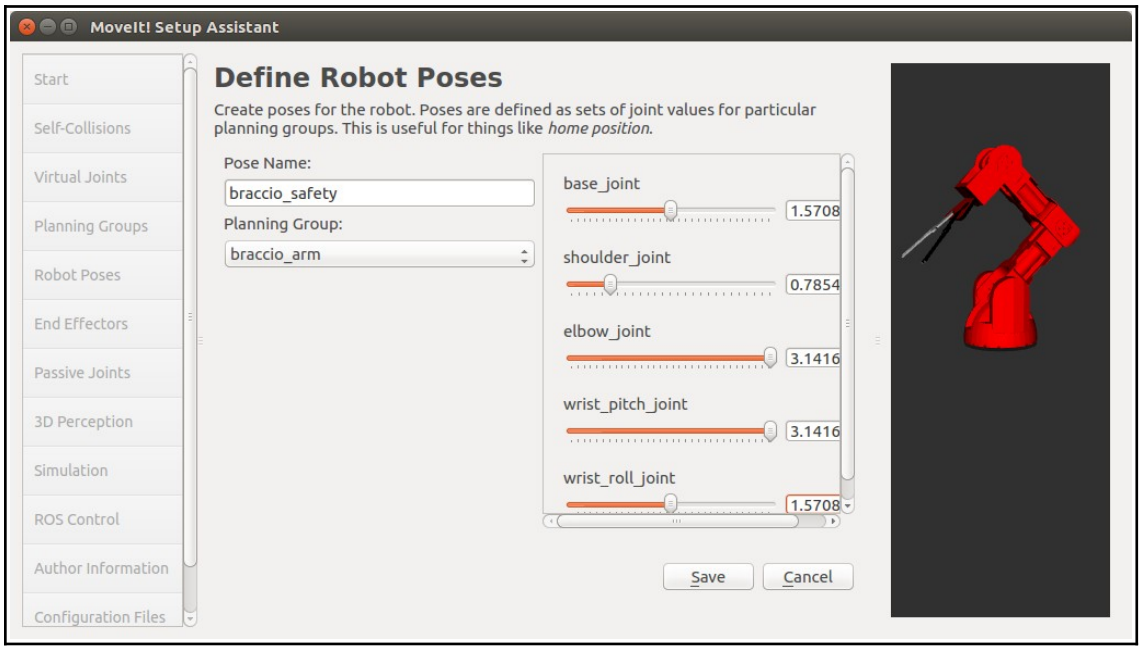


Figura 49. *Moveit_Setup_Assistant: Define Robot Poses: braccio_safety*. Fuente: elaboración propia

La segunda pose elegida es *braccio_up* (Figura 50). Los ángulos de los *joints* son [90°, 90°, 90°, 90°, 90°] expresados en radianes.

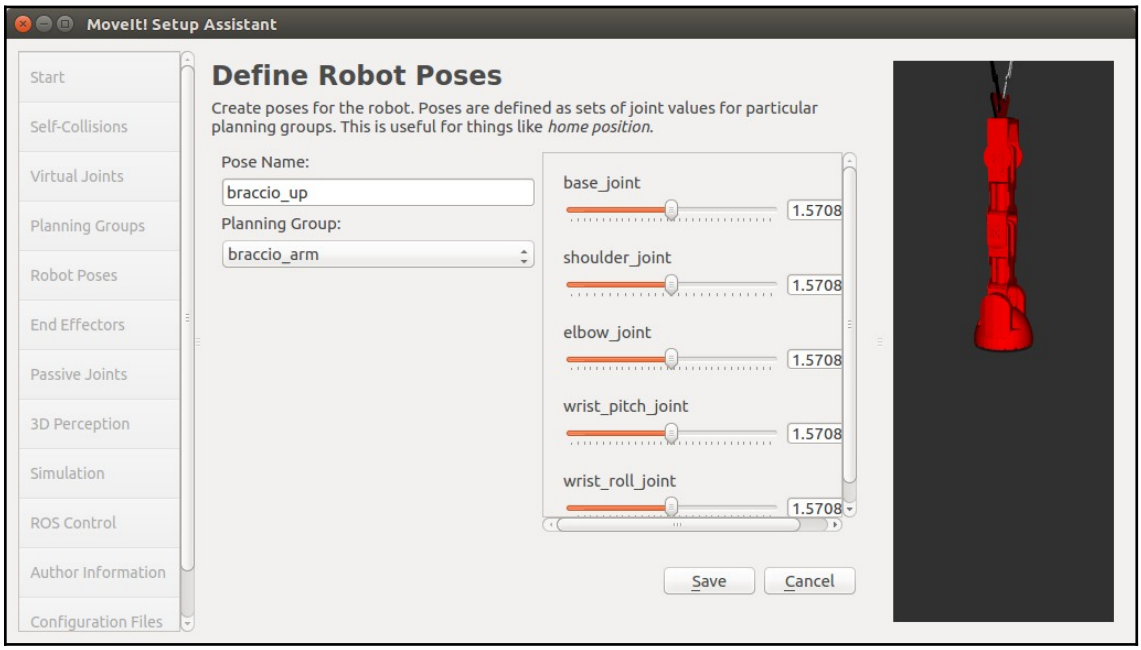


Figura 50. *Moveit_Setup_Assistant: Define Robot Poses: braccio_up*. Fuente: elaboración propia

La tercera y última pose es *braccio_home* (Figura 51). Los ángulos de los *joints* son [90°, 90°, 0°, 0°, 0°] expresados en radianes.

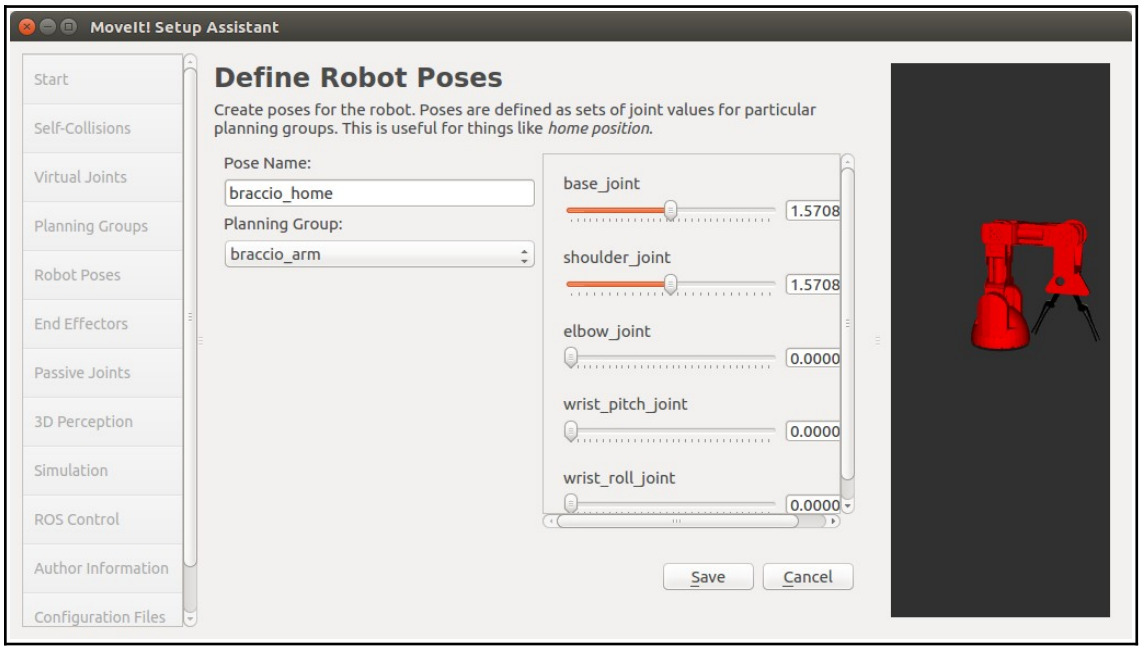


Figura 51. *Moveit_Setup_Assistant: Define Robot Poses: braccio_home*. Fuente: elaboración propia

Seguidamente creamos un grupo para la efector final (Figura 52), en este caso, el efector final es la pinza (grupo de *links* y *joints*). La designación del grupo de efectores finales permite que se realicen algunas operaciones especiales en ellos internamente; como por ejemplo, permitir desplazar la pinza en el simulador y que el brazo se mueva para que la pinza esté en la posición indicada. Es importante señalar, que si no se definiera dicho grupo el simulador; por defecto, tomaría como efector final el último *link* del grupo anterior (con el consecuente error de cálculo posicional).

Los siguientes pasos de configuración de *Passive Joints, 3D Perception, Simulation* y *ROS control* no van a ser necesarios para este proyecto. Así que los ignoramos y pasamos directamente al *Autor information* (Figura 53). Pese a ser un paso obligatorio, se puede rellenar fácilmente con los datos del autor (nombre/email) correspondiente, ya sean reales o ficticios.

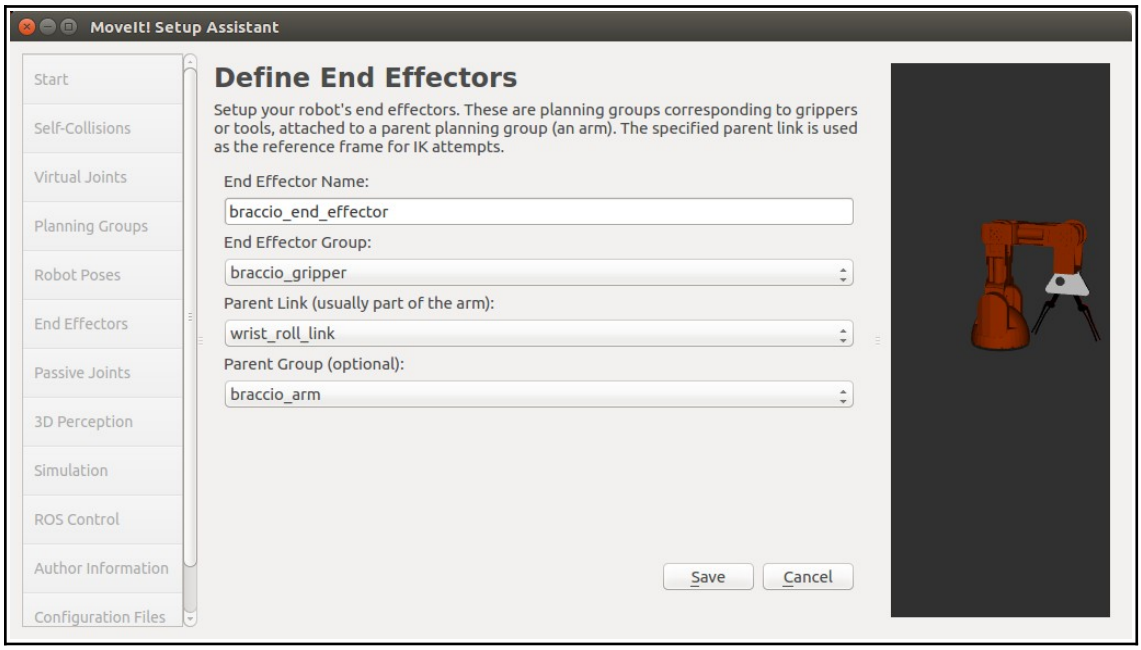


Figura 52 . *Moveit_Setup_Assistant: Define End Effectors*.Fuente: elaboración propia

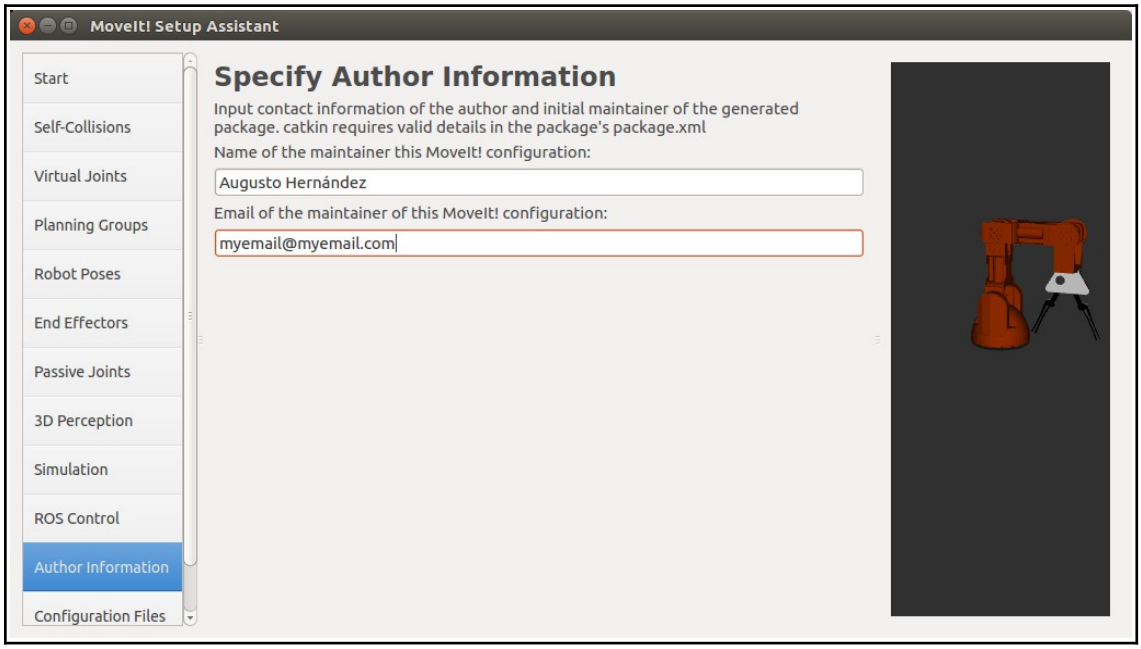


Figura 53 . *Moveit_Setup_Assistant: Specify Author Information*. Fuente: elaboración propia

Finalmente, llegamos al paso en el que se generan todos los archivos necesarios para el correcto funcionamiento de *MoveIt!* (Figura 54) y la utilización del visualizador RViz. De los archivos generados automáticamente, hay uno muy interesante llamado *visual.srdf* (puede consultarse su contenido

en el anexo) dentro de la carpeta *config,* donde aparece la información más relevante que acabamos de configurar.

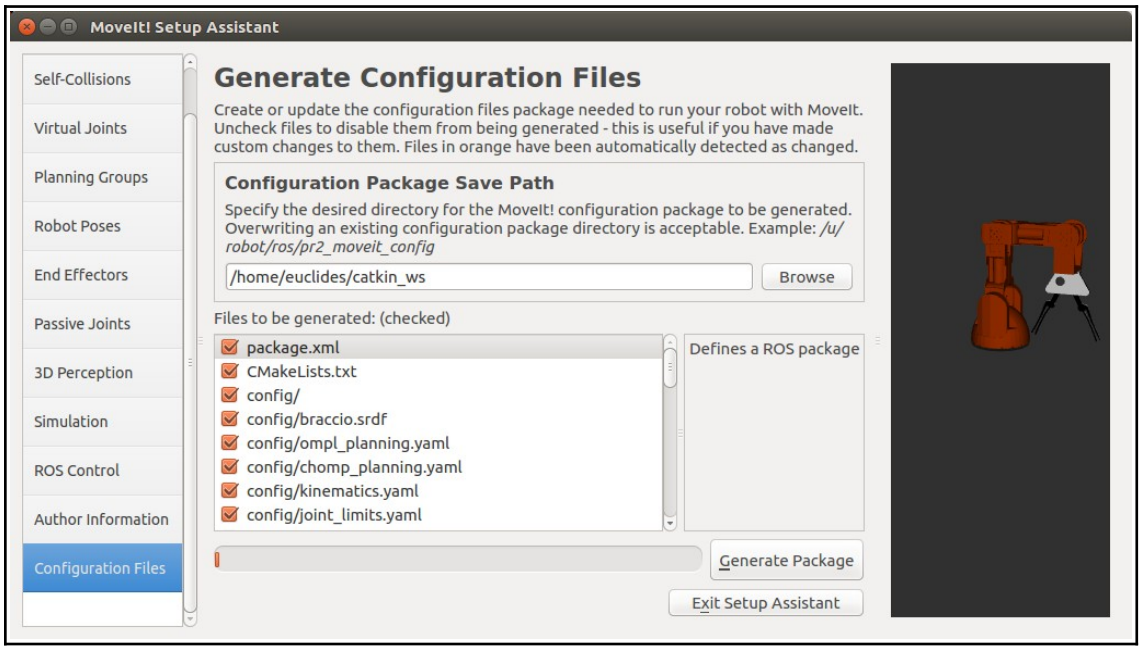


Figura 54 . *Moveit_Setup_Assistant: Generate Configuration Files* Fuente: elaboración propia

7. Prueba de funcionamiento y algunos movimientos

Llegados a este punto del TFG ya hemos visto casi toda la arquitectura de software esbozada en el apartado 2 del capítulo 3 (Figura 55).

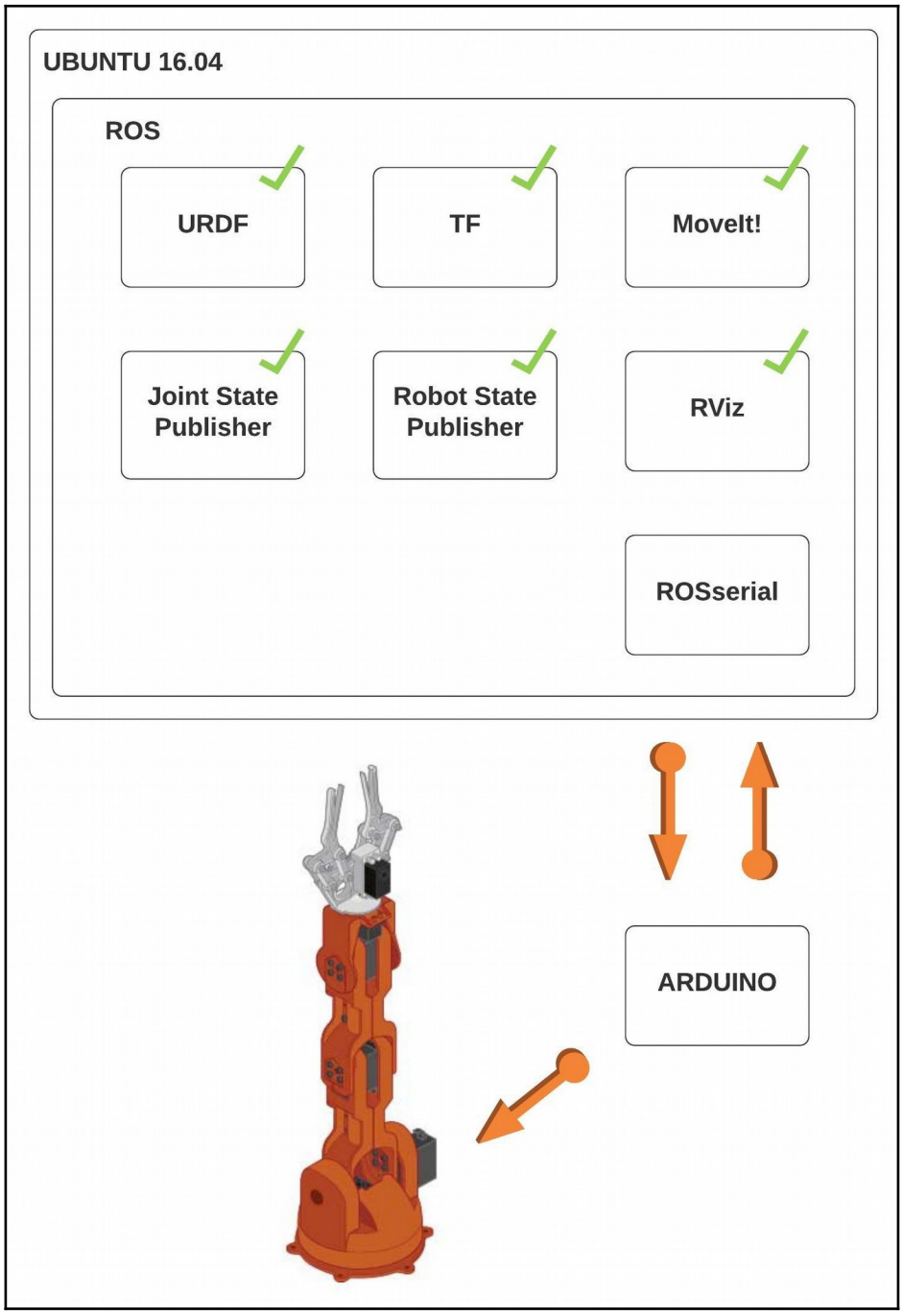


Figura 55. Diagrama de la arquitectura de software. Fuente: elaboración propia

ROSserial es un elemento que sirve de puente entre la arquitectura de *hardware* y *software* del robot manipulador. Este paquete de ROS permite que las plataformas basadas en microcontroladores se comuniquen con un PC normal y proporciona el protocolo para establecer dicha comunicación, utilizando un enfoque de arquitectura cliente-servidor. No obstante, es importante señalar, que a partir de ROS2 /DDS (*Data Distribution Service*) resulta conveniente el uso de otro estándar serial XRCE-DDS (definido y mantenido por el *Object Management Group*) que aporta comunicaciones más dinámicas, fiables y seguras. Esta nueva librería (Micro XRCE-DDS) desarrollada por la empresa española *eProsima*; ya mencionada anteriormente, posibilita que dispositivos con recursos limitados como los microcontroladores puedan comunicarse, de forma muy eficiente, en un entorno DDS [22].

Podemos instalar los paquetes rosserial en Ubuntu usando los siguientes comandos [23]:

1. Instalar los binarios del paquete rosserial usando apt-get:

$ sudo apt-get install ros-kinetic-rosserial ros-kinetic-rosserial-arduino ros-kinetic-rosserial-server

2. Crear la carpeta ros_lib que el entorno de compilación de Arduino necesita para permitir que los programas del mismo interactúen con ROS.

$ cd <sketchbook>/libraries

$ rm -rf ros_lib

$ rosrun rosserial_arduino make_libraries.py

Si el *script* anterior (make_libraries.py) se instala correctamente, se deberá generar una carpeta llamada ros_lib dentro de la carpeta de Ejemplos. Reinicie la IDE de Arduino y verifique que dicha carpeta se encuentra en el lugar correspondiente.

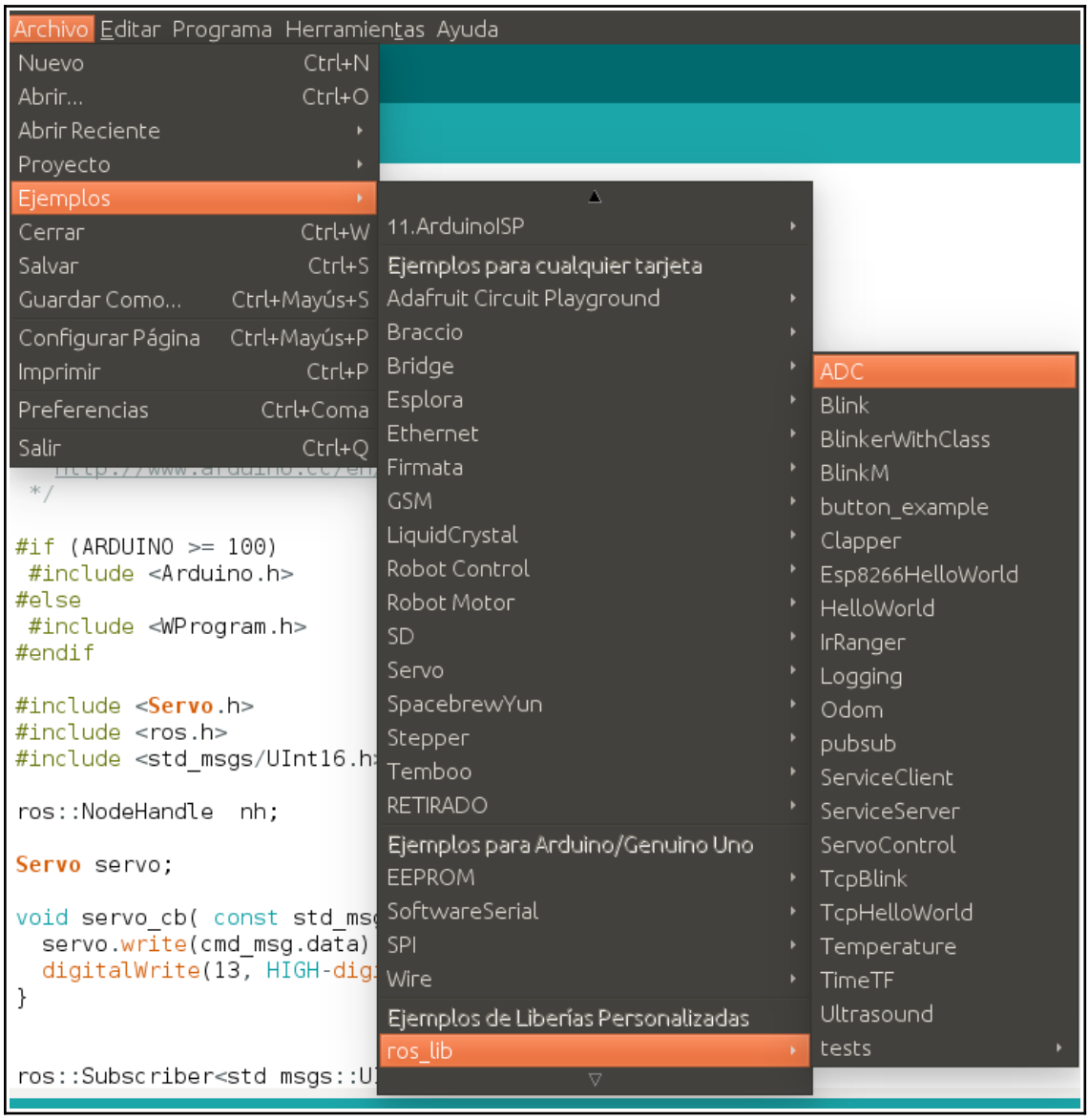


Figura 56. Ejemplos->ros_lib. Fuente: elaboración propia

Ahora ya estamos preparados para analizar el código del *sketch* **braccio_ros.ino** disponible en el repositorio de referencia:

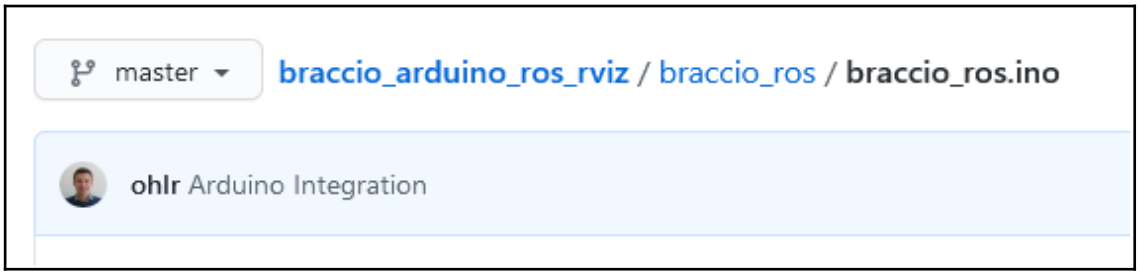


Figura 57. /braccio_ros.ino. Fuente: [16]

```
#include <ros.h>
#include <Arduino.h>
#include <BraccioLibRos.h>
#include <std_msgs/MultiArrayLayout.h>
#include <std_msgs/MultiArrayDimension.h>
#include <std_msgs/UInt8MultiArray.h>
#include <Servo.h>

ros::NodeHandle  nh;

Servo base;
Servo shoulder;
Servo elbow;
Servo wrist_ver;
Servo wrist_rot;
Servo gripper;

unsigned int _baseAngle = 90;
unsigned int _shoulderAngle = 90;
unsigned int _elbowAngle = 90;
unsigned int _wrist_verAngle = 90;
unsigned int _wrist_rotAngle = 90;
unsigned int _gripperAngle =73; //closed

void BraccioMove( const std_msgs::UInt8MultiArray& angleArray){
  _baseAngle = (unsigned int)angleArray.data[0];
  _shoulderAngle = (unsigned int)angleArray.data[1];
  _elbowAngle = (unsigned int) angleArray.data[2];
  _wrist_verAngle = (unsigned int)angleArray.data[3];
  _wrist_rotAngle = (unsigned int)angleArray.data[4];
  _gripperAngle = (unsigned int)angleArray.data[5];
}

ros::Subscriber<std_msgs::UInt8MultiArray> sub("joint_array",
&BraccioMove);

void setup()
{
  Braccio.begin();
  nh.initNode();
  nh.subscribe(sub);
}

void loop()
{


Braccio.ServoMovement(20,_baseAngle,_shoulderAngle,_elbowAngle,_wrist_
verAngle,_wrist_rotAngle,_gripperAngle);
  nh.spinOnce();
  delay(1);
}
```

Empezamos comentando los #include que se utilizan para incluir algunas bibliotecas externas relacionadas con ROS y Arduino:

```
#include <Arduino.h>
```

Es una librería de migración de versiones que incluye a su vez una serie de librerías relacionadas con el hardware.

```
#include <Servo.h>
```

Es una librería que se utiliza para controlar los servomotores.

```
#include <ros.h>
```

Es la librería de ROS para Arduino.

```
#include <std_msgs/MultiArrayLayout.h>
#include <std_msgs/MultiArrayDimension.h>
#include <std_msgs/UInt8MultiArray.h>
```

Son librerías de ROS que se usan para publicar arrays multidimensionales entre dos nodos que se ejecutan en el mismo *roscore.* Conviene recordar que la comunicación entre los nodos publisher/subscriber se establece mediante mensajes y tópicos de la siguiente forma:

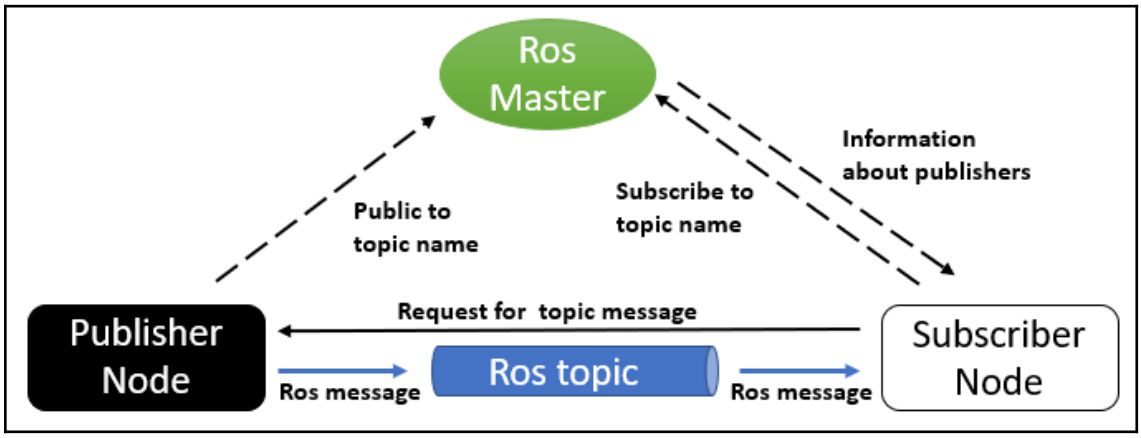


Figura 58. Componentes del sistema ROS. Fuente: [24]

El tipo del mensaje del array multidimensional es unsigned 8-bit int, puesto que el contenido del mismo serán ángulos, expresados en grados, que no superan los 180° del límite impuesto a los servomotores.

```
ros::NodeHandle  nh;
```

Instanciamos un objeto llamado nh de la clase NodeHandle. Esta clase implementa nodos de tipo subscribers, publishers, etc. en el lenguaje de programación C++. No obstante, su uso en un sketch de Arduino difiere un poco del uso en programa C++ compilado. En **braccio_ros.ino** es inializado con el método nh.initNode(); mientras que en el nodo **parse_and_publish.cpp** será inicializado con ros::init(argc, argv, "parse_and_publish");

```
Servo base;
Servo shoulder;
Servo elbow;
Servo wrist_ver;
Servo wrist_rot;
Servo gripper ; //closed
```

Instanciamos los objetos de la clase Servo necesarios para cada servomotor. La implementación del hardware se realiza de forma transparente al usario mediante la librería BraccioLibRos de Arduino.

```
unsigned int _baseAngle = 90;
unsigned int _shoulderAngle = 90;
unsigned int _elbowAngle = 90;
unsigned int _wrist_verAngle = 90;
unsigned int _wrist_rotAngle = 90;
unsigned int _gripperAngle =73;
```

```
void BraccioMove( const std_msgs::UInt8MultiArray& angleArray){
  _baseAngle = (unsigned int)angleArray.data[0];
  _shoulderAngle = (unsigned int)angleArray.data[1];
  _elbowAngle = (unsigned int) angleArray.data[2];
  _wrist_verAngle = (unsigned int)angleArray.data[3];
  _wrist_rotAngle = (unsigned int)angleArray.data[4];
  _gripperAngle = (unsigned int)angleArray.data[5];
}
```

Declaramos e inicializamos las variables de la pose inicial del robot, que se podrá ir actualizando según vayan llegando nuevos mensajes a la función `BraccioMove` (*callback*). Hay que matizar que los ángulos en el entorno Arduino se especifican en grados, mientras que en el entorno ROS están expresados en radianes. Y por lo tanto, necesitaremos crear un segundo nodo intermedio (**parse_and_publish.cpp**) que se encargue de convertir unas unidades a otras y publicarlas.

```
ros::Subscriber<std_msgs::UInt8MultiArray> sub("joint_array",
&BraccioMove);
```

Subscripción al tema (topic) “joint_array“ mediante el ROS Master. ROS llamará a la función `BraccioMove` cada vez que llegue un nuevo mensaje sobre el tópico “joint_array“. Este método sub(); del *sketch* Arduino también es algo distinta del método gemelo subscribe() que veremos posteriormente en el nodo **parse_and_publish.cpp**.

```
void setup()
{
  Braccio.begin();
  nh.initNode();
  nh.subscribe(sub);
}
```

Con Braccio.begin(); se inicializa el robot Braccio y se ajusta la posición inicial. Modificando esta función se puede configurar la posición inicial de todos los servomotores. Según la versión de placa shield que se disponga (en nuestro hardware tenemos la versión V4), es recomendable pasar el valor int 0 (SOFT_START_DEFAULT) como parámetro (`Braccio.begin(0);`)para que el brazo robótico no se ponga en funcionamiento de forma brusca.

La inicialización del objeto ROS nh y su correspondiente suscripción vuelve a diferir un poco en el código Arduino, si bien el objetivo es el mismo.

```
void loop()
{


Braccio.ServoMovement(20,_baseAngle,_shoulderAngle,_elbowAngle,_wrist_
verAngle,_wrist_rotAngle,_gripperAngle);
  nh.spinOnce();
  delay(1);
}
```

Finalmente llegamos al loop donde se van a pasar los ángulos a los servomotores y ejecutarse los movimientos deseados. El primer parámetro que se pasa es un retardo de 20 milisegundos que se producirá entre los movimientos de cada servo, el resto de parámetros son los ángulos en grados que ya conocemos.

`nh.spinOnce()`permite que se reciban todas las llamadas a la función *callback* `BraccioMove()`. Con el retardo de 1 milisegundo termina el bucle del microcontrolador Arduino.

Como hemos podido comprobar, el código de la parte de Arduino se corresponde a un nodo suscriptor (*subscriber*). No obstante, como ya hemos mencionado anteriormente, también necesitamos implementar un nodo intermedio que haga las conversiones de unidades de radianes a grados y las publique en el tópico "joint_array". Además también tendrá que suscribirse al tópico "joint_states", para poder recibir los datos (ángulos expresados en radianes) que ese otro nodo **/joint_state_publisher** está publicando desde la GUI o ventana de control deslizante (Figura 34), pero también de forma interna.

A continuación, hacemos un diagrama específico del sistema de comunicación ROS utilizado para una mejor comprensión del mismo:

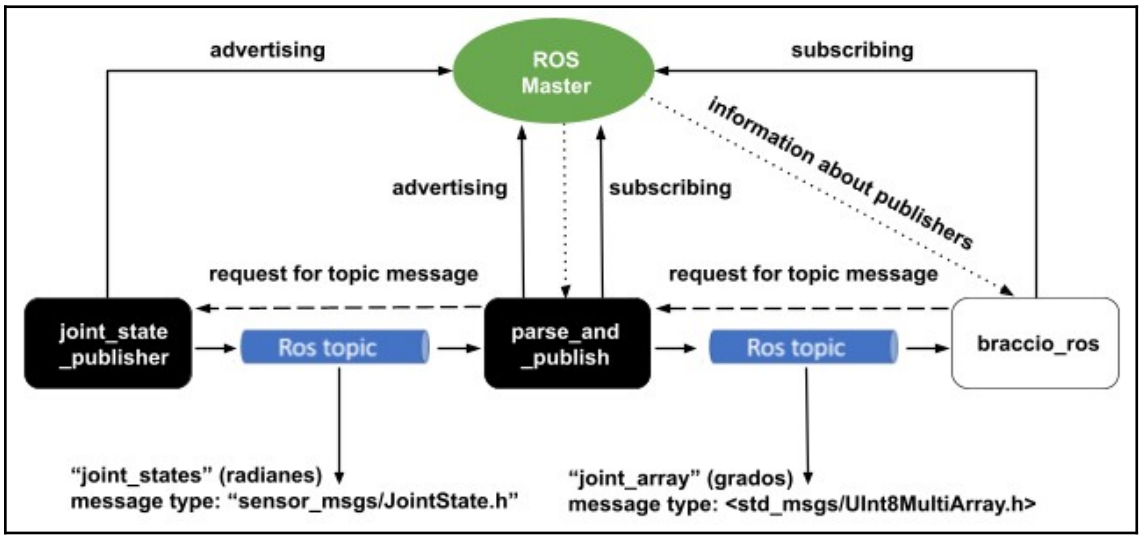


Figura 59. Componentes específicos del sistema ROS. Fuente: elaboración propia

El siguiente código que vamos a analizar es **parse_and_publish.cpp**, también disponible en el repositorio de referencia:

Figura 60. /parse_and_publish.cpp. Fuente: [16]

```
#include "ros/ros.h"
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/UInt8MultiArray.h"
#include "sensor_msgs/JointState.h"

#define PI 3.1416

uint _DataArray[6];

void chatterCallback(const sensor_msgs::JointState::ConstPtr& msg)
{
  int i=0;
  for(i=0; i<6; i++)
  {
    //ROS_INFO("I heard: [%d]", uint((msg->position[i])/PI*180));
    _DataArray[i]= uint((msg->position[i])/PI*180);
  }
}

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

ros::init(argc, argv, "parse_and_publish");
ros::NodeHandle n;
ros::Subscriber sub = n.subscribe("joint_states", 6, chatterCallback);

ros::Publisher pub =
n.advertise<std_msgs::UInt8MultiArray>("joint_array", 6);

ros::Rate loop_rate(10);

while (ros::ok())
  {
    std_msgs::UInt8MultiArray array;

    //Clear array

    array.data.clear();

    //for loop, pushing data in the size of the array
    for (int i = 0; i < 6; i++)
    {
     array.data.push_back(_DataArray[i]);
    }
    pub.publish(array);

    ros::spinOnce();

    loop_rate.sleep();
  }

return 0;

}
```

En C++ los archivos de encabezado (*header files*) se incluyen entre comillas:

```
#include "ros/ros.h"
```

El *header* `/ros.h` tiene todos los archivos necesarios para implementar las funcionalidades de ROS. No podemos crear un nodo ROS sin incluir el mismo.

```
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/UInt8MultiArray.h"
```

Estos tres `#include` son los mismos que hemos visto en el sketch de Arduino. Si queremos usar un tipo de mensaje específico en un nodo, tenemos que incluir el archivo de encabezado del mensaje. ROS tiene una serie de mensajes con tipos predefinidos, y el usuario también puede crear tipos de mensajes nuevos. Hay un paquete de mensajes incorporado en ROS llamado `std_msgs`, que tiene una definición de mensaje de tipos de datos estándar como int, float, string, UInt8MultiArray, etc.

```
#define PI 3.1416

uint _DataArray[6];
```

Se declara y define la constante matemática PI necesaria para realizar la conversión de radianes a grados. También se declara un array de tipo entero sin signo para ir almacenando los ángulos de los *joints* en radianes.

```
void chatterCallback(const sensor_msgs::JointState::ConstPtr& msg)
{
  int i=0;
  for(i=0; i<6; i++)
  {
    //ROS_INFO("I heard: [%d]", uint((msg->position[i])/PI*180));
    _DataArray[i]= uint((msg->position[i])/PI*180);
   }
 }
```

Cuando nos suscribimos a un tema (*topic*) ROS y llega un mensaje sobre dicho tema, se activa la función devolución de llamada *callback*. (`chatterCallback()`. Además, es precisamente en esta función, donde se hace la operación `radianes/PI*180` para poder pasar de radianes a grados, y guardar todo en el array anteriormente declarado. Si se desactivan las // del

comentario //ROS_INFO, podremos ver por pantalla los ángulos en grados que se van guardando en el array.

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

ros::init(argc, argv, "parse_and_publish");
```

Inicializar el nodo es un paso obligatorio para cualquier nodo ROS. Después de la función int main (), tenemos que incluir `ros :: init ()` que se encarga de inicializar apropiadamente el mismo. Podemos pasar los argumentos de la línea de comandos argc, argv a la función init () y el nombre del nodo `"parse_and_publish"`. Este será el nombre del nodo que podremos recuperar, cuando hagamos un listado de los nodos activos por la *shell,* con el comando $rosnode list.

```
ros::NodeHandle n;
```

Después de inicializar el nodo, tenemos que crear una instancia del objeto NodeHandle que arranque el nodo ROS y otras operaciones, como publicar/suscribir un tópico.

```
ros::Subscriber sub = n.subscribe("joint_states", 6, chatterCallback);
```

Al contrario que sucedía con el *Subscriber* de la parte Arduino, en la implementación C++ compilada, cuando nos suscribimos a un tema, no es necesario que mencionemos el tipo de mensaje, pero sí debemos mencionar el nombre del tema y la función de devolución de llamada (*callback*). Esta función es definida por el usuario y se ejecuta una vez que llega un mensaje ROS sobre el tema de suscripción. Dentro de la devolución de llamada, podemos modificar el mensaje ROS, imprimirlo o tomar una decisión basada en los datos del mensaje. El segundo parámetro de la función `subscribe ()` es el tamaño de la cola de mensajes (6). Si los mensajes llegan más rápido de lo que se procesan, esta será la cantidad de mensajes que se almacenarán en el búfer antes de comenzar a desechar los más antiguos.

```
ros::Publisher pub =
n.advertise<std_msgs::UInt8MultiArray>("joint_array", 6);

ros::Rate loop_rate(10);
```

Por otro lado, al crear el objeto Publsher, sí que resulta necesario mencionar el tipo del mensaje. El tamaño de la cola de mensajes también es 6. El `loop_rate(10)` se usa para dar un período de tiempo específico a una tarea, o introducir un retardo expresado en Hz (10 Hz=1/10 s= 100 ms). Una vez declarado e inicializado, habrá que llamar a la función `sleep()` para que tenga efecto.

```
while (ros::ok())
  {
    std_msgs::UInt8MultiArray array;

    //Clear array

    array.data.clear();
```

Dentro del bucle `while` se declara el array multivariable y se vacía de datos;si tiene alguna información previa sobre ángulos en grados.

```
 //for loop, pushing data in the size of the array
    for (int i = 0; i < 6; i++)
    {
     array.data.push_back(_DataArray[i]);
    }
```

El *loop* interno *for* va recorriendo el array de tipo `uint` e inicializando el array `UInt8MultiArray` multidimensional, necesario para poder publicar los datos en serie.

```
 pub.publish(array);

 ros::spinOnce();

 loop_rate.sleep();
}
```

Se publica el array completado en el bucle anterior. Después de iniciar la suscripción o la publicación hay que llamar a una función para poder procesar dicha solicitud. En un nodo C++ se llama a la función `ros :: spinOnce ()` después de publicar un tema, o llamar a la función `ros :: spin ()` si solo está suscribiendo un tema. Si como en este caso, se están haciendo ambas cosas, se llama a la función `ros :: spinOnce ()`. [25]

Finalmente, con el método `loop_rate.sleep()`se activa el *loop_rate* declarado fuera del *while*. A partir de aquí el nodo podrá ser abortado con la combinación de teclas CTRL + C.

Una vez que ya se ha visto el ecosistema ROS, estudiado la arquitectura de software, analizado el sistema de comunicación entre nodos y el código que implementa esos componentes. El último paso es realizar algunas pruebas de funcionamiento con el robot real. Para ello vamos a seguir las instrucciones que aparecen en la página web de Github [16]:

Run

Terminal 1:

```
$roscore
```

Launches Roscore, that handles communictation between all ROS nodes.

Terminal 2:

```
$source devel/setup.bash
$cd src/braccio_arduino_ros_rviz
$roslaunch braccio_arduino_ros_rviz urdf.launch
model:=urdf/braccio_arm.urdf
```

Starts the GUI and publishes angles in Radian.

Terminal 3:

```
$source devel/setup.bash
$cd src/braccio_arduino_ros_rviz
$rosrun braccio_arduino_ros_rviz parse_and_publish
```

Converts the joint angles to degrees and reduces the message size

Terminal 4:

```
$rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Starts the communication between Arduino and Pc. Change "/dev/ttyACM0" to the port the Port of your Arduino. You find this Information in the Arduino IDE, ArduinoIDE>Tools>Port.

Terminal 5 (optional/Debugging):

```
$rostopic echo joint_array
```

view what is published to the Arduino

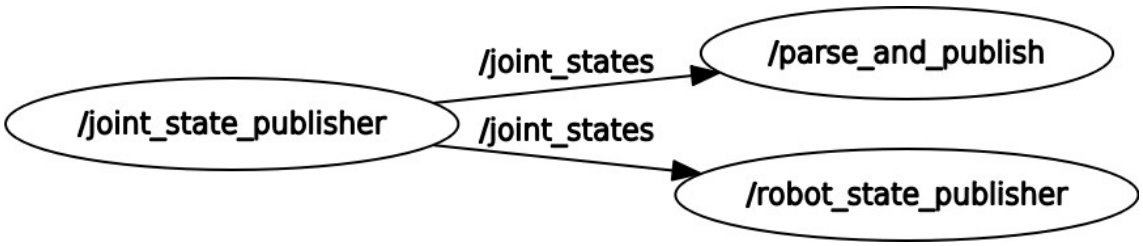
```
$rostopic echo joint_states
```

view what is published by the joint_state_publisher (GUI)

```
$rqt_graph
```

overview about the topics and nodes

Should look similar like this:



Al ejecutar la serie de comandos anteriores en las terminales o *shell* correspondientes, se inició el visualizador RVIZ y se desplegó la GUI (ventana deslizante) desde la que se puede controlar la cinemática directa del robot Braccio. En el siguiente enlace de *youtube* se puede ver un vídeo con el resultado obtenido:

https://www.youtube.com/watch?v=DEiugQy9lIM

No obstante, la cinemática inversa no está implementada en el repositorio que hemos tomado como referencia para este TFG. Como ya vimos en el capítulo 6, esa parte se puede configurar íntegramente con la herramienta *Moveit!.* La serie de instrucciones a seguir son casi las mismas que acabamos de ver en la página anterior. Simplemente tenemos que sustituir los comandos de la **Terminal 2** por:

```
$source devel/setup.bash
$cd src/braccio_arduino_ros_rviz
$roslaunch ros_braccio_moveit demo.launch
```

En realidad, la **Terminal 1** no sería necesaria, puesto que el comando *roslaunch* ya se encarga, por defecto, de arrancar el ROS Master. En el siguiente enlace de *youtube* se puede ver un vídeo con el resultado obtenido, la pose seleccionada *braccio_safety*, fue configurada en el capítulo 6 (Figura 48):

https://www.youtube.com/watch?v=XPNOdXIY3JQ

8. Conclusiones

Como se ha podido experimentar a lo largo de todo este proyecto, ROS es un entorno ideal para el desarrollo de robots. El ecosistema que lo acompaña también es digno de destacar, URDF, Rviz y *MoveIt*! son herramientas muy útiles que facilitan y automatizan gran parte del trabajo inicial.

Señalar que seguimos viviendo un segundo curso de pandemia COVID-19 que ha mermado la continuidad del TFG. No obstante, la extensión mínima y objetivos del mismo han sido alcanzados. Finalmente no ha sido necesario implementar nuevo código C++ y usar la librería de álgebra Eigen, puesto que se ha generado el código de forma automática con *Moveit!* (consultar Anexos).

En este proyecto hemos trabajado intensamente la cinemática directa e inversa del robot manipulador. El siguiente paso lógico podría consistir en la localización y agarre de la pelota de tenis con ayuda de algún tipo de sensor de proximidad ToF (*Time of Flight*) o cámara, y un algoritmo Deep Learning de tiempo real, como por ejemplo, YOLO_v3 (*You Only Look Once*).

9. Bibliografía

[1]. Curva de aprendizaje. [En línea]. Disponible en:
https://es.wikipedia.org/wiki/Curva_de_aprendizaje
[Accedido: 1-febrero-2021]

[2]. ROS Kinetic Kame. [En línea]. Disponible en:
http://wiki.ros.org/kinetic
[Accedido: 8-febrero-2021]

[3]. Tinkerkit Braccio Robot. [En línea]. Disponible en:
https://store.arduino.cc/tinkerkit-braccio-robot
[Accedido: 10-febrero-2021]

[4]. Arduino Uno. [En línea]. Disponible en:
https://es.wikipedia.org/wiki/Arduino_Uno
[Accedido: 12-febrero-2021]

[5]. ROS Concepts. [En línea]. Disponible en:
http://wiki.ros.org/ROS/Concepts
[Accedido: 15-febrero-2021]

[6]. ROS CommandLine. [En línea]. Disponible en:
http://wiki.ros.org/ROS/CommandLineTools#roslaunch
[Accedido: 17-febrero-2021]

[7]. *Paul, Richard (1981) Robot Manipulators: Mathematics, Programming, and Control : the Computer Control of Robot Manipulators. MIT Press, Cambridge, Massachusetts*

[8]. Hubert, Harald Andreas Uwe (2012) *Selbstkalibrierung der Hand-Kamera-Kinematik eines anthropomorphen Roboters. pp. 11, 12.* Rheinische Friedrich-Wilhelms-Universität Bon.

[9]. Arquitectura Micro-ROS. [En línea]. Disponible en:
https://micro.ros.org//docs/overview/features/
[Accedido: 18-febrero-2021]

[10]. Craig, John J. (2006) *Robótica.* pp. 69. 3° Edición. PEARSON EDUCACIÓN, México.

[11]. Barrientos, Antonio et al. (2012) *Fundamentos de Robótica.* pp. 170,. 2° Edición. McGraw-Hill, Madrid, España.

[12]. Peter Corke. [En línea]. Disponible en:
https://petercorke.com/toolboxes/robotics-toolbox/
[Accedido: 01-marzo-2021]

[13]. K.M. Lynch & F.C. Park (2017) *Modern Robotics – Mechanics, Planning & Control*, Cambridge University Press.

[14]. *Aircraft principal axes*. [En línea]. Disponible en:
https://en.wikipedia.org/wiki/Aircraft_principal_axes
[Accedido: 01-marzo-2021]

[15]. Lentin, Joseph (2015) *Mastering ROS for robotics Programming*, Packt Publishing.

[16]. braccio_arduino_ros_rviz. [En línea]. Disponible en:
https://github.com/ohlr/braccio_arduino_ros_rviz
[Accedido: 02-marzo-2021]

[17]. Cadena cinemática cerrada y abierta. [En línea]. Disponible en:
http://www.sitenordeste.com/mecanica/maquinas_mecanismos.htm
[Accedido: 03-marzo-2021]

[18]. Newman, Wyatt (2017) *A Systematic Approach to Learning Robot Programming with ROS*, Chapman and Hall/CRC.

[19]. Función biyectiva. [En línea]. Disponible en:
https://es.wikipedia.org/wiki/Funci%C3%B3n_biyectiva
[Accedido: 05-marzo-2021]

[20]. MoveIt!. Concepts [En línea]. Disponible en:
https://moveit.ros.org/documentation/concepts/
[Accedido: 11-marzo-2021]

[21]. MoveIt!. Setup_Assistant [En línea]. Disponible en:
http://moveit2_tutorials.picknik.ai/doc/setup_assistant/
setup_assistant_tutorial.html
[Accedido: 11-marzo-2021]

[22]. eProsima Micro XRCE-DDS [En línea]. Disponible en:
https://github.com/eProsima/Micro-XRCE-DDS
[Accedido: 10-abril-2021]

[23]. rosserial_arduino [En línea]. Disponible en:
http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup
[Accedido: 11-abril-2021]

[24]. Hands-On Introduction to Robot Operating System (ROS) [En línea]. Disponible en:
https://trojrobert.medium.com/hands-on-introdution-to-robot-operating-system-ros-4914386e4a45
[Accedido: 01-mayo-2021]

[25]. Lentin, Joseph (2018) *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*, Packt Publishing.

10. Anexos

Contenido del archivo de configuración generado automáticamente: **visual.srdf**

```
<?xml version="1.0" ?>
<!--This does not replace URDF, and is not an extension of URDF.
    This is a format for representing semantic information about the robot structure.
    A URDF file must exist for this robot as well, where the joints and the links that are referenced are defined
-->
<robot name="braccio">
    <!--GROUPS: Representation of a set of joints and links. This can be useful for specifying DOF to plan for, defining arms, end effectors, etc-->
    <!--LINKS: When a link is specified, the parent joint of that link (if it exists) is automatically included-->
    <!--JOINTS: When a joint is specified, the child link of that joint (which will always exist) is automatically included-->
    <!--CHAINS: When a chain is specified, all the links along the chain (including endpoints) are included in the group. Additionally, all the joints that are parents to included links are also included. This means that joints along the chain and the parent joint of the base link are included in the group-->
    <!--SUBGROUPS: Groups can also be formed by referencing to already defined group names-->
    <group name="braccio_arm">
        <joint name="base_joint" />
        <joint name="shoulder_joint" />
        <joint name="elbow_joint" />
        <joint name="wrist_pitch_joint" />
        <joint name="wrist_roll_joint" />
    </group>
    <group name="braccio_gripper">
        <joint name="gripper_joint" />
        <joint name="sub_gripper_joint" />
    </group>
    <!--GROUP STATES: Purpose: Define a named state for a particular group, in terms of joint values. This is useful to define states like 'folded arms'-->
    <group_state name="braccio_safety" group="braccio_arm">
        <joint name="base_joint" value="1.5708" />
        <joint name="elbow_joint" value="3.1416" />
        <joint name="shoulder_joint" value="0.7854" />
        <joint name="wrist_pitch_joint" value="3.1416" />
        <joint name="wrist_roll_joint" value="1.5708" />
    </group_state>
    <group_state name="braccio_up" group="braccio_arm">
        <joint name="base_joint" value="1.5708" />
        <joint name="elbow_joint" value="1.5708" />
        <joint name="shoulder_joint" value="1.5708" />
        <joint name="wrist_pitch_joint" value="1.5708" />
        <joint name="wrist_roll_joint" value="1.5708" />
```

```
</group_state>
<group_state name="braccio_home" group="braccio_arm">
  <joint name="base_joint" value="1.5708" />
  <joint name="elbow_joint" value="0" />
  <joint name="shoulder_joint" value="1.5708" />
  <joint name="wrist_pitch_joint" value="0" />
  <joint name="wrist_roll_joint" value="0" />
</group_state>

  <!--END EFFECTOR: Purpose: Represent information about an end effector.-->

<end_effector name="braccio_end_effector" parent_link="wrist_roll_link" group="braccio_gripper" parent_group="braccio_arm" />

  <!--VIRTUAL JOINT: Purpose: this element defines a virtual joint between a robot link and an external frame of reference (considered fixed with respect to the robot)-->
        <virtual_joint name="braccio_virtual_joint" type="planar" parent_frame="world" child_link="base_link" />

  <!--DISABLE COLLISIONS: By default it is assumed that any link of the robot could potentially come into collision with any other link in the robot. This tag disables collision checking between a specified pair of links. -->
        <disable_collisions link1="base_link" link2="braccio_base_link" reason="Adjacent" />
  <disable_collisions link1="base_link" link2="elbow_link" reason="Never" />
  <disable_collisions link1="base_link" link2="shoulder_link" reason="Never" />
        <disable_collisions link1="base_link" link2="wrist_pitch_link" reason="Never" />
  <disable_collisions link1="base_link" link2="wrist_roll_link" reason="Never" />
        <disable_collisions link1="braccio_base_link" link2="elbow_link" reason="Never" />
        <disable_collisions link1="braccio_base_link" link2="shoulder_link" reason="Adjacent" />
        <disable_collisions link1="braccio_base_link" link2="wrist_pitch_link" reason="Never" />
        <disable_collisions link1="braccio_base_link" link2="wrist_roll_link" reason="Never" />
        <disable_collisions link1="elbow_link" link2="left_gripper_link" reason="Never" />
        <disable_collisions link1="elbow_link" link2="right_gripper_link" reason="Never" />
        <disable_collisions link1="elbow_link" link2="shoulder_link" reason="Adjacent" />
        <disable_collisions link1="elbow_link" link2="wrist_pitch_link" reason="Adjacent" />
  <disable_collisions link1="elbow_link" link2="wrist_roll_link" reason="Never" />
        <disable_collisions link1="left_gripper_link" link2="right_gripper_link"
```

```
reason="Never" />
    <disable_collisions link1="left_gripper_link" link2="shoulder_link"
reason="Never" />
    <disable_collisions link1="left_gripper_link" link2="wrist_pitch_link"
reason="Never" />
    <disable_collisions link1="left_gripper_link" link2="wrist_roll_link"
reason="Adjacent" />
    <disable_collisions link1="right_gripper_link" link2="shoulder_link"
reason="Never" />
    <disable_collisions link1="right_gripper_link" link2="wrist_pitch_link"
reason="Never" />
    <disable_collisions link1="right_gripper_link" link2="wrist_roll_link"
reason="Adjacent" />
    <disable_collisions link1="shoulder_link" link2="wrist_pitch_link"
reason="Never" />
    <disable_collisions link1="shoulder_link" link2="wrist_roll_link"
reason="Never" />
    <disable_collisions link1="wrist_pitch_link" link2="wrist_roll_link"
reason="Adjacent" />
</robot>
```