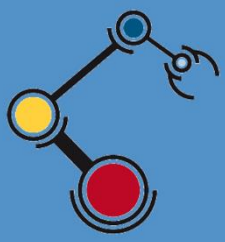
Escuela Politécnica Superior

Simulación de un robot Mitshubishi-PA10 sobre ROS



Máster Universitario en Automática y Robótica

Trabajo Fin de Máster

Autor:
Martín Martín Meirás

Tutor/es:
Gabriel Jesús García Gómez

Julio 2017

Universitat d'Alacant
Universidad de Alicante

Simulación de un robot Mitsubishi-PA10 sobre ROS

Martín Martín Meirás

Junio 2017

Dedicado a mi compañera
por todo su apoyo en mis decisiones
y a mi familia por estar cuando la necesito.

Agradecimientos

Agradezco a mi tutor Gabriel Jesús García Gómez y al grupo de Automática, Robótica y Visión Artificial de la Universidad de Alicante por los materiales prestados así como por las instalaciones del laboratorio.

Resumen

El contenido de este documento es el del Trabajo Fin de Máster en Automática y Robótica Industrial de la Universidad de Alicante que tiene como objetivo detallar los pasos seguidos para la integración del robot industrial Mitsubishi-PA10 sistema operativo ROS, así como el proceso seguido para la creación de un servidor propio compatible con el protocolo de comunicación de ROS, para comunicar dicho sistema con el controlador real del robot.

Por otro lado también se explica en dicho documento cómo hacer uso de los paquetes y comandos para poner en marcha el robot y controlarlo desde ROS.

No es propósito de este proyecto realizar ningún uso práctico con las herramientas que otorga ROS, sin embargo sí garantizar que los paquetes creados funcionan correctamente y se podrá hacer uso de ellos para próximos proyectos o investigaciones.

Así mismo también se explica el protocolo que utiliza ROS para integrar y comunicar cualquier robot en su entorno y con sus herramientas. Por otro lado se explica el estado en el que se dejan las herramientas y las compatibilidades con los paquetes más usados en investigación con ROS.

Índice general

Agradecimientos **III**

Resumen **V**

Índice de figuras **IX**

Lista de Acrónimos **XI**

1. Introducción **1**

1.1. Motivación del proyecto 2

2. Directorios de descripción del robot **3**

2.1. Archivos de lanzamiento 4

2.2. Archivos de configuración de los eslabones 5

3. Creación del archivo de descripción **7**

4. Herramienta de planificación moveit **9**

4.1. Creación del paquete de planificación moveit 9

4.1.1. Comprobación de la configuración 11

4.2. Gestión de controladores 11

4.3. Archivo Moveit planning execution launch 12

5. Simulación del robot Mitsubishi-PA10 **13**

6. Diseño del servidor del Mitsubishi-PA10 **15**

6.1. Flujo de ejecución y separación de tareas 16

6.2. Seguimiento de la trayectoria. 18

6.3. Configuración de la ejecución de la trayectoria. 19

7. Elementos del sistema **21**

7.1. Puesta en marcha del sistema 22

7.1.1. Ejemplo de trayectoria 23

8. Resultados y Conclusiones. **27**

A. Archivo de descripción URDF. **29**

B. Archivo de lanzamiento del asistente moveit. **35**

Bibliografía **37**

Índice de figuras

2.1. Organización de directorios del robot Mitsubishi-PA10 3

2.2. Organización de directorios de los archivos inicializadores del Mitsubishi-PA10 5

3.1. Visualización del archivo de descripción del robot Mitsubishi-PA10 . . . 8

4.1. Formato de controladores usados por los paquetes de ROS-Industrial . 11

4.2. Interfaz de control gráfica de las posiciones objetivo y planificaciones con moveit 12

7.1. Nodos que intervienen en la simulación con ROS 21

7.2. Primeros comandos en ROS para ejecutar los paquetes del robot. . . . 23

7.3. Resultado en consola de la puesta en marcha. 24

7.4. Resultado de simulación del robot en Rviz. 24

7.5. Planificación de la trayectoria. 25

Lista de Acrónimos

ROS Robot Operating System

URDF Unified Robot Description Format

RViz ROS Visualization or Robot Visualizer

XML Extensible Markup Language

ISA Industrial Standard Architecure

TCP Transmision Control Protocol

IP Internet Protocol

UDP User Datagram Protocol

Capítulo 1

Introducción

Con el transcurso del tiempo las tecnologías cambian y evolucionan. La robótica ha cobrado desde hace años un papel cada vez más importante en la industria y en la investigación, y como consecuencia de ello han surgido herramientas cada vez más potentes. Todas ellas como necesidad para el desarrollo de nuevos sistemas o elementos robóticos y adaptadas a las necesidades de todo tipo de sensores, actuadores y tipos de datos que forman parte de las aplicaciones robóticas de hoy en día. Una de ellas y que cada vez se ha ido extendiendo más, tanto en la industria, como en investigación, es el sistema operativo ROS. Tanto es así que marcas conocidas tales como Universal Robots, Kuka o Fanuc, desarrollan los paquetes de sus robots industriales en la parte industrial de dicha plataforma [1].

ROS es un entorno sobre Linux donde se puede diseñar, programar e integrar sistemas, sensores o robots de distintos fabricantes, lenguajes de programación o naturaleza de uso. Como otros sistemas operativos proporciona servicios estándar tales como: abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidades de uso común, paso de mensajes entre procesos y mantenimiento de paquetes.

Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores entre otros. Por lo tanto junta en un mismo entorno información de clases distintas y puede trabajar con ella al mismo tiempo que está disponible para la toma de decisiones (automáticas o no) y para todo el hardware que se esté comunicando con ROS.

ROS tiene dos partes básicas donde radica su potencia. La primera de ellas es el sistema operativo en sí el cual con la descripción que se ha dado, se pueden desarrollar multitud de sistemas. Sin embargo sus características no son su único ni su más potente atractivo, la herramienta más potente es ros-pkg.

ROS-pkg es una suite donde se encuentran una gran cantidad de paquetes aportados por la extensa comunidad de usuarios que implementan funcionalidades para su libre uso por el resto de usuarios de ROS. Paquetes con finalidades tales como la localización y mapeo simultáneo de objetos o la planificación, simulación y control de robots, que es el tópico que nos atañe en este trabajo.

Por todo ello, lo que se pretende con este proyecto es permitir a los usuarios del brazo robótico Mitsubihi-PA10 utilizarlo en el mundo de la investigación para tareas mucho más avanzadas que las que su propio software otorga, ya que dicho robot no cuenta con soporte para este sistema operativo debido a que es más antiguo que el propio entorno de ROS.

Cabe destacar que han sido de ayuda para entender mejor dada la falta de documentación a parte de las propias paginas oficiales de ROS, a veces incompletas o poco detalladas consultar y comprobar consultas ya realizadas por la comunidad de ROS [2]. Por este motivo la comunidad que ROS tiene detrás es una de sus grandes ventajas.

1.1. Motivación del proyecto

Dada la potencia del software anteriormente descrito y la gran comunidad que existe detrás del mismo se establece como finalidad de este trabajo fin de Máster la integración del robot Mitsubishi-PA10 en el sistema operativo robótico ROS.

Dentro de dicha integración se han englobado: el estudio de los estándares de ROS para que sea aceptado por la comunidad, la creación de paquetes de simulación en el formato ámpliamente extendido en la robótica como es URDF, la creación de paquetes para la herramienta moveit de planificación de trayectorias, así como crear un servidor propio para entender las órdenes comandadas desde ROS y que puedan ser llevadas a cabo por el controlador del robot en la vida real y no sólo en el entorno de simulación.

En primer lugar se definió una serie de archivos para el modelado e integración del robot necesarios para trabajar en ROS. Seguidamente se procedió a la creación de dichos archivos, principalmente los relacionados con la visualización de los eslabones y la creación de las articulaciones.

En segundo lugar, fueron creados y adecuados para nuestro caso los archivos correspondientes a la extendida y potente herramienta de planificación de ROS llamada ***moveit***. Esta tarea conllevaba no solo la conversión de la descripción en URDF sino también la configuración de un controlador para las tareas internas de ROS.

A continuación se trató la comunicación e interpretación de los mensajes entre ROS y el controlador. Para ello se tuvo que estudiar el protocolo utilizado por los paquetes de ROS- Industrial para la comunicación con plataformas que funcionen con hardware y software distintos, y por otro lado diseñar el servidor que funcionase en el hardware del PA10, capaz de interpretar no sólo órdenes y movimientos enviados por ROS, sino también de enviar periódicamente señales o mensajes de estado del controlador y de las articulaciones del robot real.

Capítulo 2

Directorios de descripción del robot

Con el fin de que el trabajo aquí realizado pueda ser de utilidad y uso para la amplia comunidad de ROS se han desarrollado todos los paquetes de manera que cumplan los requerimientos y estándares de ROS.

Dichos estándares fueron desarrollados con el objetivo de utilizar una misma nomenclatura en el entorno de ROS de manera que la utilización de cualquier robot resulte sencilla para el usuario y de esta manera una vez se comprenda la estructura de los paquetes y lo que contienen cada uno se pueda proceder a interactuar con ellos y crear aplicaciones propias dentro del desarrollo de cualquier sistema industrial o investigación académica.

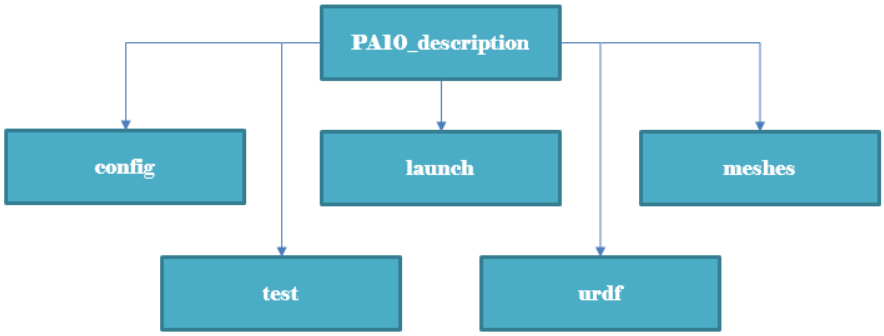


Figura 2.1: Organización de directorios del robot Mitsubishi-PA10

Dentro de ROS existen dos tipos principales de archivos, comunes a cualquier tipo de robot. Los archivos de lanzamiento o "launch files": encargados de poner en marcha dentro del sistema los nodos necesarios para visualizar e interactuar con el modelo del robot y su controlador. Y por otro lado los archivos de descripción o "xacro macro"que nos permiten simular las distintas variantes del robot proporcionándonos su modelo en formato URDF(Universal Robot Description File).

La estructura principal del directorio del paquete de apoyo o de descripción de cualquier robot es la que se puede observar en la Figura 2.1. Dicha oranización se ha extraido de las directrices que proporciona la propia Organización de ROS, para desarrolladores y que se puede consultar en [1].

La carpeta ***config*** contendrá archivos básicos como nombres de las articulaciones (joint names file) configuraciones de entornos preconfigurados en Rviz(ROS visualization tool), controladores propios creados para la simulación que actúen como el real, y otra serie de archivos más complejos dependiendo de las herramientas o paquetes que usemos en el diseño de nuestra aplicación en ROS.

Los archivos contenidos en la carpeta ***meshes*** son aquellos donde se encuentran los modelos 3D de cada uno de los eslabones del robot. Dichos archivos se dividen a su vez en modelos visuales y de colisiones.

Los de colisiones son archivos menos pesados y menos precisos, esto es debido a que son los usados para calcular las posibles colisiones entre eslabones y con el entorno, por lo tanto para agilizar la simulación y disminuir la carga computacional se requiere que contengan menos puntos que definan la estructura.

La tercera carpeta es la carpeta ***test*** en la cual se encuentran los archivos con movimientos simples o predeterminados que comprueban el estado de la conexión entre ROS y el controlador y localizan posibles errores en la configuración.

Finalmente la cuarta y quinta carpeta son en las que se determina la compatibilidad entre ROS y nuestro propio robot, en este caso el Mitsubishi-PA10.

2.1. Archivos de lanzamiento

Por otro lado la carpeta ***launch*** es aquella que arranca los paquetes de ROS o programas creados por nosotros mismos, para llevar a cabo diferentes tareas. Esta carpeta debe contener al menos 4 tipos de archivos:

- El archivo ***load_pa10.launch*** se encarga de cargar en el parámetro del sistema denominado ***robot_description***, que no es más que un tipo de variable estandarizada que requieren y al que pueden acceder los distintos paquetes dentro de ROS.

- El archivo ***test_pa10.launch*** por otro lado se encarga de cargar el archivo de descripción en formato URDF en el visualizador Rviz, de manera que el usuario pueda interactuar con el modelo del robot y verificar sus movimientos y limitaciones. Por otro lado también lanza los nodos ***joint_state_publisher*** y ***robot_state_publisher*** los cuales son responsables de publicar los movimientos o estados de las articulaciones del robot en cuestión. Este archivo no necesita pues de ningún tipo de elemento físico o de hardware para llevarlo a cabo, ya que su misión es la de representar los posibles movimientos y configuraciones del robot.

- Por otro lado el archivo ***robot_state_visualize_pa10.launch*** permite al usuario visualizar el estado actual real o simulado del robot en concreto. Este archivo por otro lado si que necesita de un hardware real o simulado además de al menos un parámetro, la dirección IP donde el controlador del robot deberá estar corriendo un servidor capaz de comunicarse con los nodos de ROS que lanza este archivo. En nuestro caso es un servidor propio que hemos desarrollado para que interprete el protocolo estándar de ROS llamado *simple_message* y que procederemos a explicar en próximos capítulos.

- Por último el caso del archivo ***robot_interface_streaming_pa10.launch*** es un poco más complicado. Este archivo es donde entra en escena el diseño del cliente y de la aplicación en ROS que se debe comunicar con el controlador del robot real.

Sin embargo ROS proporciona una serie de archivos estándar para poderse comunicar con un servidor que cumpla unos requisitos mínimos que será de los que partamos en el diseño del servidor del Mitsubishi-PA10. Por lo tanto y dado que el propósito de este trabajo no es diseñar una aplicación en ROS sino crear los archivos y la configuración necesarios para poder ser utilizados en próximas investigaciones y por la comunidad de ROS, no se ha creado ningún cliente específico más allá del proporcionado por los paquetes de ROS-Industrial.

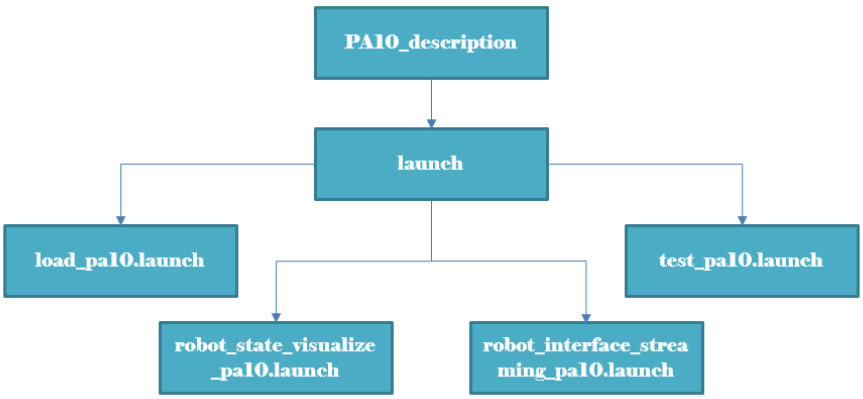


Figura 2.2: Organización de directorios de los archivos inicializadores del Mitsubishi-PA10

2.2. Archivos de configuración de los eslabones

Otro directorio importante a la hora de crear los archivos compatibles de un robot dentro de ROS, es el que contiene la descripción de las articulaciones y eslabones del propio robot.

Este directorio es importante, no sólo porque contiene la descripción principal del robot a partir de la cuales calcular movimientos y posiciones objetivo, tipo de anclaje con el mundo, etc. Sino también para permitir a futuros usuarios crear variantes del mismo con diferentes terminaciones, versiones distintas de la que se dispone en este caso, u otro tipo de configuraciones con las que pueda contar el robot.

Por lo general este directorio cuenta al menos con 3 tipos de archivos: ***"pa10-macro.xacro"***, ***"pa10.xacro"***, ***"pa10.urdf"***.

El primer archivo de todos denominado como ***"pa10-macro.xacro"*** y con extensión *".xacro"* es el archivo principal que define el robot específico con el que se está trabajando.

No constituye un archivo URDF en sí mismo sino que contiene la definición de los parámetros que pueden cambiar de entre las diferentes versiones disponibles.

Por otro lado el segundo archivo con extensión .xacro es el archivo ***"pa10.xacro"*** cuya misión es la de instanciar un modelo de nuestro robot en una escena vacía. Por lo tanto la principal diferencia con el archivo anterior radica en que no sólo se describe al robot sino también al entorno en el que se encuentra y las características tridimensionales de la aplicación en concreto que estemos desarrollando.

Esta instancia de nuestro robot será con la que el software interactúe tanto en simulación como para los cálculos de movimientos en el hardware real, como en la vida real. Este archivo no está pensado para ser usado en otro tipo de archivos xacro en los que se modifique la escena o en los que se incluyan otros robots, es el archivo anterior el idóneo del que partir e incluir en otras futuras escenas para esas tareas.

Este archivo será el que cargará por defecto el archivo *"load_pa10.launch"*.

El archivo ***"pa10.urdf"*** con extensión *".urdf"* es el archivo que contiene la descripción completa del robot a partir del archivo *"pa10.xacro"* y debe ser usado por aquellos paquetes de ROS que no usan o son compatibles con la extension *".xacro"*.Por la definición del propio formato URDF no puede incluir otros archivos con descripciones de otros sensores, componentes o actuadores para ser cargados en un visualizador, para ese propósito es el archivo indicado arriba.

Debido a que el principal este proyecto se centra en poner en marcha el sistema completo de simulación del robot Mitsubishi-PA10 de 7 grados de libertad y el servidor en el controlador real, el desarrollo de estos archivos carece de relevancia, pero sí lo será para futuros trabajos donde se utilicen varios modelos del mismo brazo robótico, una escena distinta o para el uso de estas librerías por los usuarios de la comunidad de ROS, por ese motivo han sido explicados y tenidos en cuenta.

Capítulo 3

Creación del archivo de descripción

El formato URDF es un formato universal usado para describir un robot. La descripción de un robot conlleva indicar cuáles son los eslabones de los que está compuesto, por qué medios se conectan unos a otros, qué tipo de articulaciones(prismáticas, cilíndricas...) y las limitaciones de giro de sus articulaciones además de otras muchas características.

Dicho formato incluye la posibilidad de añadir mallas de puntos que describen la tipología y apariencia 3D de los eslabones que componen el robot, es decir, su apariencia visual o en su defecto usar elementos geométricos básicos y poder representarlo por cualquier visualizador.

A pesar de lo extendido que está este formato, cuenta con algunas limitaciones, como por ejemplo el hecho de que se puede incluir solamente un robot, por archivo de descripción.

Esto es una gran limitación en la industria ya que existen numerosos elementos de agarre que se utilizan de forma alternada en la industria con un mismo brazo robótico y que poseen su propio, completo y específico archivo URDF. Por no mencionar los casos en los que para una aplicación se utilizan más de un robot que interactúan entre si o con el entorno.

Para estos casos existe otro formato denominado xacro, el cual se encarga de generar otro URDF más completo una vez incorporados los datos de los URDF que queremos incluir. Por lo tanto este formato sirve para incluir descripciones de otros robots en dicho formato y de esta forma crear escenas y modelos más completos, o versiones distintas evitando copiar y pegar código para cada versión y tener un archivo URDF distinto para cada robot personalizado.

Ambos formatos constan de un archivo escrito en un lenguaje de etiquetas con palabras clave en formato XML, como el del Apéndice A.

En principio para comprobar los sentidos de giro y ubicación de los eslabones de nuestro robot el archivo URDF será sencillo. El Mitsubishi-Pa10 que vamos a modelar cuenta con 7 eslabones y 7 articulaciones. Por lo que deberemos crear con la etiqueta *link*, cada uno de los eslabones indicando su posición y orientación con respecto al sistema de referencia fijo, además de un link añadido denominado base_link que carecerá de dimensiones físicas y se utilizará para indicar la conexión con el sistema de referencia fijo, en nuestro caso un anclaje al mismo.

Una vez contemos con los eslabones descritos tendremos que describir las articulaciones con la etiqueta *joint*, indicar sus ejes de giro o desplazamiento, la tipología de articulación (prismática o de giro), los eslabones que están conectados con cada articulación, etc.

Una vez se defina nuestro URDF podremos visualizarlo con el siguiente comando y comprobar si todo está definido correctamente o si se tienen que cambiar sentidos de giro, enlaces de las articulaciones o posiciones de los eslabones. El resultado se muestra en la Figura 3.1

roslaunch urdf_tutorial display.launch model:=<urdf_file>gui:=True

Mediante esta visualización se puede comprobar el sentido de giro de las articulaciones, las orientaciones entre cada una de ellas y los límites de giro. Con la creación de este archivo le hemos proporcionado a ROS un parámetro accesible por todos y cada uno de los nodos y herramientas de robótica incluidos en ROS y que se identifica con la etiqueta *robot_description*.

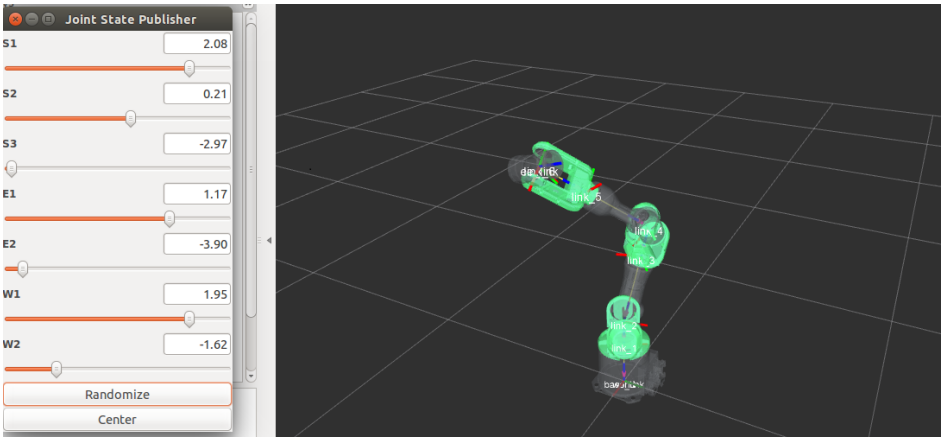


Figura 3.1: Visualización del archivo de descripción del robot Mitsubishi-PA10

Capítulo 4

Herramienta de planificación moveit

Moveit es una poderosa herramienta dentro de ROS para aplicaciones de manipulación móvil de objetos, planificación de trayectorias, control y navegación. Consiste en un conjunto de paquetes y librerías que configuran una plataforma de uso fácil para el desarrollo de aplicaciones avanzadas en el mundo de la robótica.

Es capaz de permitirnos desarrollar aplicaciones de coordinación entre varios grupos robóticos de control para coordinar los movimientos y además de planificar las trayectorias a seguir evitando obstaculos que se describan en la escena. Se puede encontrar más información acerca de su uso y capacidad en su pagina oficial que se indica en [3].

Es la plataforma más extendida de software libre, por ello crear los paquetes de configuración de cualquier robot en esta plataforma es una gran recomendación para el mantenimiento en el tiempo y la potencia de uso y desarrollo de los paquetes del mismo.

4.1. Creación del paquete de planificación moveit

Como se ha visto en este documento, para cada paquete o herramienta de ROS que queramos integrar con un robot, debemos crear los archivos de configuración que la herramienta requiere.

En el caso de moveit la configuración es bastante sencilla. Basta con tener el archivo URDF que describe el robot en cuestión, instalar las librerías de moveit desde su página oficial y lanzar el asistente que proporcionan sus propios desarrolladores. Mediante el siguiente comando desde la consola de un terminal en Linux se puede iniciar dicho asistente:

roslaunch moveit_setup_assistant setup_assistant.launch

Es importante notar que al partir tan solo del archivo URDF de descripción del robot, en el momento en el que se produzca algún cambio en el mismo, se deberá lanzar de nuevo el asistente, para que de esta manera asegurarnos de que los cambios realizados sobre la descripción no afectan a los archivos que necesita y genera el planificador moveit.

Por otro lado y en beneficio del formato, se tiene la opción de añadir archivos con la extensión que anteriormente comentábamos, para de esta manera incluir otros grupos móviles o manipuladores finales a nuestro entorno de simulación. De igual forma se podrán añadir varios brazos en un mismo entorno como se pretende hacer en futuros proyectos del grupo de investigación.

Tras esto y siguiendo el asistente gráfico se puede proceder a crear:

- Primero: una matriz de autocolisión. Comprobando las colisiones que aparecen detectadas por moveit, se podrá determinar si dichas colisiones se deben a elementos en contacto o a errores en la creación del archivo de descripción.

- Segundo: articulaciones virtuales. Moveit al menos requiere una articulación virtual que es, ni más ni menos, que el anclaje del eslabón base *("base_link")* con el mundo o la escena *("world")*.

- Tercero: los llamados grupos planificadores. Este termino se refiere a cada subconjunto del robot que es capaz de moverse independientemente del resto. Para el caso de varios brazos robóticos cada uno de ellos es un grupo planificador ya que para alcanzar un punto determinado se calcula la solución de cada uno de los grupos de articulaciones de cada brazo para llegar a ese punto por separado, no existe una transmisión cinemática entre cada subconjunto de articulaciones.

Así mismo en este apartado se puede definir una terminación para cada grupo con el propósito de realizar movimientos y aplicaciones de agarre de objetos o también llamado (*"grasping"*).

- Cuarto: añadir posiciones del robot. Este apartado es opcional dentro del asistente sin embargo sirve para definir ciertas posiciones de las articulaciones del robot como por ejemplo la posición *home*, *escape*, *security*...

- Quinto: añadir una herramienta final. Simplemente se deben rellenar las opciones en concordancia con los nombres otorgados a las articulaciones y eslabones en el archivo URDF.

- Sexto: generar los archivos. En este apartado se deberá especificar el lugar donde se guardaran los archivos de configuración.

Cabe destacar que dado que el principal objetivo a alcanzar por este trabajo es simular e integrar el robot con los paquetes de ROS-Industrial, no se ha añadido ninguna herramienta en la punta del mismo.

No obstante se indican los distintos pasos a realizar en la configuración, en caso de desear incluir dicha terminación:

- Por un lado tener la definición de la herramienta en el archivo URDF. Posteriormente en el apartado tercero de la configuración con moveit, habría que añadir un grupo planificador de tipo herramienta para la misma darle un nombre e indicar las articulaciones que lo componen.

- Y en segundo lugar se deberá añadir en el apartado quinto una herramienta, añadiendo un nombre, el nombre del grupo planificador creado anteriormente y la última articulación del brazo robótico con la que se enlaza dicha herramienta.

4.1.1. Comprobación de la configuración

Para comprobar que los paquetes de configuración se han creado correctamente basta con hacer uso del siguiente comando en la terminal:

roslaunch pa10_moveit_config/launch/demo.launch

Mediante este comando se lanzan los paquetes de moveit como si se lanzara el nodo *robot_state_publisher* cuya misión es la de permitir mover las articulaciones dentro de sus límites para comprobar que la representación, y descripción del robot es correcta.

Siguiendo los pasos descritos en el apartado 1.1 de [4] conseguimos guardar la configuración visual en Rviz en un archivo que posteriormente utilizará moveit.

4.2. Gestión de controladores

Una vez acabada la fase de configuración, moveit tiene suficiente información para realizar planificación de trayectorias y comprobación de colisiones, pero no para comunicarse con un robot físico.

Para ello debemos crear o modificar los siguientes archivos:

- Archivo ***controllers.yaml***. Este archivo tiene que contar con un formato similar al de la Figura 4.1, y estará ubicado en la carpeta config del paquete de moveit que se haya generado(<robotName>_moveit_config/config/controllers.yaml).

```
controller_list:
  - name: ""
    action_ns: joint_trajectory_action
    type: FollowJointTrajectory
    joints: [joint_1, joint_2, joint_3, joint_4, joint_5, joint_6]
```

Figura 4.1: Formato de controladores usados por los paquetes de ROS-Industrial

El nombre determinará parte de la ruta del topic al que enviará la trayectoria deseada *(name/ns)*, y el nombre del servidor de acción que estará escuchando a los nodos de moveit, para enviar comandos de creación o cancelación de movimientos.

Los nombres de las articulaciones que aparecen en la figura deberán coincidir con los especificados en el archivo descriptivo URDF.

4.3. Archivo Moveit planning execution launch

Además del archivo que define el controlador que se quiera utilizar, se han de elaborar los archivos que nos permitan poner en marcha toda la simulación del robot o en su defecto el robot real. El principal archivo que desempeña esta función es pa10_moveit_execution_planning.launch.

Lanzando este archivo dispondremos de toda la potencia de planificación de moveit tanto en simulación como en el robot real.

Este archivo se caracteriza por lanzar todos los paquetes, nodos, parámetros y servicios necesarios para planificar, calcular y comunicar movimientos del robot. La función de cada uno de los paquetes de ese entramado de archivos ya ha sido explicada en cada apartado de este trabajo así como en la explicación del marco general de la creación de los paquetes de un robot en ROS.

Por lo tanto será mediante el siguiente comando como se empezará la simulación del Mitsubishi-Pa10 y de cualquier otro robot en ROS:

roslaunch pa10_moveit_config pa10_moit_execution_planning.launch sim:=<true or false>robot_ip:= <ip del robot>.

Tras esta orden se debe abrir el visualizador Rviz con el plugin para comandar mediante una interfaz simple coordenadas objetivo para el robot como se puede observar en la Figura 4.2

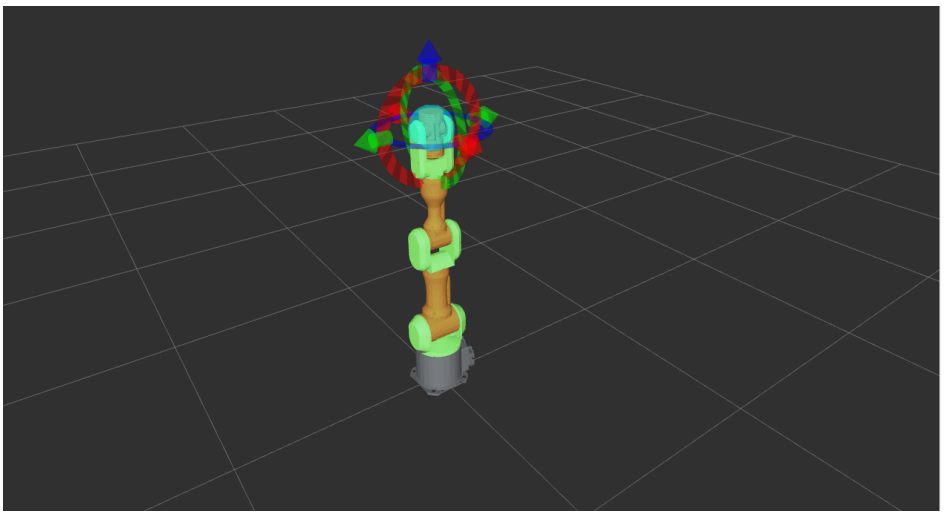


Figura 4.2: Interfaz de control gráfica de las posiciones objetivo y planificaciones con moveit

Capítulo 5

Simulación del robot Mitsubishi-PA10

Llegado a este punto se tienen desarrollados la descripción del robot y los archivos de configuración de la herramienta planificadora Moveit. Con todo ello el simulador del robot en ROS está listo para transmitir de manera correcta los movimientos y la cinemática al controlador real. No obstante carece de un elemento de comunicación con el controlador físico y real.

Es en este proceso de la integración con ROS en el que tenemos dos opciones de desarrollo: usar los paquetes que ROS-Industrial proporciona o por otro lado crear librerias propias que sigan el protocolo denominado como simple_message que se definen en [5].

En el caso de este proyecto se ha decidido optar por usar los paquetes que ROS proporciona para tal cometido, dado que forman parte del conjunto de los paquetes de ROS-Industrial donde existe un entorno completamente integrado, donde posteriormente se pretende poner a disposición del resto de usuarios los paquetes de este robot. Por ello si se usan elementos ya conocidos por la comunidad será más fácil que se extienda se uso. Además no es propósito de este trabajo generar nuevas formas de integración con ROS, sino que tiene como objetivo la creación de dichos paquetes listos para su uso como herramienta para desarrollos más prioritarios.

Para el tema a tratar en este capítulo los paquetes que realizan la tarea son *industrial_robot_client* e *industrial_robot_simulator*. Los cuales suscribiéndose a los topics que se publican en ROS relacionados con el robot e indicándoles como parámetro de entrada las dirección IP de destino del controlador físico del brazo robótico, crean la comunicación por sockets TCP/IP necesaria para enviar la información al mismo.

Industrial_robot_client es un nodo cuya labor es mandar paquetes mediante comunicación por sockets Ethernet. Los detalles de dicha comunicación se tratan en profundidad en el siguiente capítulo, no obstante cabe destacar que existen dos modalidades de comunicación.

Aquella en la que el robot inicia la comunicación y transmite en tiempo real los movimientos al controlador, o en su lugar una transmisión en la que el controlador se descarga por completo la trayectoria antes de ejecutarla. Dicha trayectoria se especifica en coordenadas articulares y así es como se transmite.

La principal ventaja de este último es que se indica cuando inicia y acaba un movimiento, y sin embargo en el modelo de transmisión en tiempo real se supone una trayectoria infinita en la que una vez que inicia el movimiento no se detiene hasta que se cierra la comunicación.

Este último modo, debido a las características y modos de control de nuestro robot supone un problema. Como se verá en el capítulo siguiente, el modo de control elegido por limitaciones de comunicación y cumplimiento de tiempos en la ejecución de las trayectorias, será el control en velocidad y debido a ello, el hecho de desconocer el momento en que finaliza una trayectoria supone un problema para comandar una velocidad nula y detener el robot en el último punto de cada trayectoria.

Los parámetros necesarios para poner en marcha estos nodos son la dirección IP donde se encontrará activo un servidor que transforme y comunique esos movimientos al controlador real del Mitsubishi-PA10 y una variable booleana denominada *sim* que indica si deseamos simular el robot o por el contrario realizar movimientos con el controlador real.

Por el contrario el paquete industrial robot simulator, simula la recepción y generación de los mensajes por parte del controlador real. Este paquete no implica una simulación de movimientos del robot sino una simulación del módulo de comunicaciones entre ROS y el controlador real, generando los mensajes que debería generar el servidor del controlador real.

El lanzamiento de estos nodos de comunicación se realiza incluyendo en el archivo de lanzamiento de los archivos de moveit, el archivo de lanzamiento genérico llamado *robot_interface_download* dentro del paquete *industrial_robot_client* y lanzar de nuevo el mismo comando que se indicó en el capítulo anterior. Este archivo se puede consultar en el Apéndice B.

Capítulo 6

Diseño del servidor del Mitsubishi-PA10

Una vez puesta a punto la simulación por completo en el entorno de ROS. Se puede ir un paso más allá y no solo utilizar el entorno para calcular movimientos y simular en una escena previamente creada, sino también llevar esos movimientos a cabo comunicando el controlador del robot físico y el robot en ROS.

Para transmitir los datos, el conjunto de los paquetes de ROS envía los movimientos por TCP-IP, por lo que para poder recibirlos en el lado del robot se deberá realizar una aplicación del tipo cliente-servidor que reciba dichos mensajes, envíe las respuestas que se requieran por el mismo canal y realice los movimientos oportunos comunicándose con el controlador del robot.

Los movimientos van codificados en un protocolo que se denomina en los repositorios y documentación oficial ***simple_message***. El protocolo es bastante sencillo y se explica en profundidad en el siguiente apartado, no obstante cabe decir que se trata de una trama de bytes con distinto significado dependiendo del tipo de mensaje que se esté enviando, como por ejemplo una orden de movimiento una petición del estado del robot o de las coordenadas actuales del brazo robótico.

El desarrollo del servidor se concibió partiendo de tres posibles vías de desarrollo. La primera de ellas era desarrollar un nodo propio dentro de ROS que transformase los mensajes a un formato completamente distinto del *simple_message* y los transmitiera a un servidor ya existente en la computadora que se comunica con el controlador del robot.

A pesar de las ventajas de esta vía, como por ejemplo que ya se partía de un servidor previamente realizado, se observaron una serie de limitaciones que ralentizarían el desarrollo. Dicho servidor surgió con la motivación de controlar el robot a distancia pero a la vez con las librerías propias que proporcionaba el fabricante, no poseía ninguna funcionalidad para interpretar el protocolo de ROS.

Otra de las limitaciones era la posibilidad de que surgieran incompatibilidades debidas a las necesidades de ROS, tales como mensajes de estado del robot y de las articulaciones, requerimientos que no poseía el servidor previamente existente y por lo tanto esas funcionalidades tendrían que ser desarrolladas. Y por otro lado el desarrollo en ROS de la integración de dichas respuestas con múltiples paquetes que son transparentes para nosotros mediante el uso de las librerías ya existentes: *industrial_robot_client* e *industrial_robot_simulator*.

La segunda posibilidad que se planteaba era la de partir del servidor ya existente e implementar el protocolo usado por ROS y además las funcionalidades existentes. Sin embargo esta tarea conllevaba la comprensión, análisis, implementación e integración de las funcionalidades de ambas partes, tarea que también nos ralentizaría.

En tercer lugar también se contempló la idea de desarrollar las librerías propias del robot en el sistema operativo Linux y de esa manera tan solo realizar un archivo que tradujese las órdenes del topic relevante, en este caso *joint_path_command*. No obstante dicha tarea conlleva entender muy bien la comunicación con la tarjeta de la computadora que se comunica con el controlador y además se necesita poseer el código fuente de las librerías del robot. De manera que también ralentizaría mucho la tarea.

Por último se observó que muchas de esas complicaciones desaparecían si el servidor se llevaba a cabo de la tercera forma: desde cero. Conociendo por completo el diseño y el flujo de ejecución del programa se puede diseñar e implementar las funcionalidades de forma más rápida y precisa.

Por lo tanto ha sido esta vía la que se ha decidido tomar, dejando otras posibles como futuros proyectos.

6.1. Flujo de ejecución y separación de tareas

Como en toda aplicación de tipo cliente-servidor, el servidor debe ejecutar dos flujos de programa paralelos. En uno de ellos el servidor debe quedarse a la espera de recibir la información y el otro debe enviar los mensajes que se solicitan al cliente.

Cualquiera de los mensajes del protocolo definido por ROS tiene en común tres cosas: un primer carácter de 4 bytes el cual indica la longitud en bytes del conjunto del mensaje; una cabecera en la que se indican 3 parámetros,el tipo de mensaje, el tipo de comunicación, y el tipo de respuesta a dicho mensaje y la tercera que lleva el propio mensaje.

El primer parámetro de dicha cabecera indica el tipo de dato que se envía y que está definido y usado dentro de ROS. Cada tipo de dato que soporta el protocolo está enumerado en el archivo de cabecera denominado *simple_message.h* y que se encuentra en el repositorio de Github destinado a este protocolo [5].

Para el caso de Moveit el tipo de mensaje que contiene las órdenes de movimiento que genera es llamado *joint_trajectory_point* el cual contiene información de las coordenadas articulares del destino en un vector de hasta 10 componentes del tipo *joint_data*, en el caso de aquellas componentes que fueran no necesarias se les asignaría un valor nulo. Por otra parte dicho mensaje también incluye como dato el tiempo en el que se requiere que se haya completado dicho movimiento.

El segundo parámetro de la cabecera indica el tipo de comunicación de entre 4 opciones: mensaje no valido("0"), mensaje de tipo topic("1") que es un tipo de mensaje periódico para que sea tratado como un topic de ROS, mensaje de tipo petición de servicio("2"), y mensaje de tipo respuesta a un servicio("3").

Por último el tercer parámetro de la cabecera determina el tipo de respuesta, que puede ser de tres tipos; no válido ("0"), exitoso("1"), o fallido("2"). Será no valido cuando el tipo de comunicación sea invalido, topic o de petición de servicio.

El conjunto de mensajes necesarios para comunicar ROS y nuestro servidor esta compuesto por tres y todos ellos están definidos en el protocolo *simple_message*.

- El primero de ellos es el de *joint_states*, este mensaje será emitido por el servidor en el hilo de transmisión y como una comunicación de tipo topic. La información que transmite es la posición de las coordenadas articulares del robot en cada instante de tiempo (recomendable de entre 10 y 50 Hz).

- El segundo mensaje que se generará será el llamado *robot_status*. En este mensaje se transmitirá el estado del controlador en general, si el robot se encuentra en movimiento, si está disponible para moverse, si el pulsador de emergencia está activado o si los motores del robot están alimentados. Este segundo mensaje también será generado y transmitido en el mismo hilo de ejecución que el anterior.

Ambos mensajes se reciben por el puerto 11002 de la computadora y se han de enviar al mismo puerto de la máquina sobre la que esté funcionando ROS. Son de tipo de comunicación topic, por lo que no requieren de ninguna respuesta.

- Por último el mensaje que indica las órdenes de movimiento es *joint_path_command*. Este servicio de ROS se compone de una trayectoria de puntos, generados por moveit, para que el robot no solo alcance el punto final objetivo sino que lo haga por el camino indicado por el planificador. Este mensaje se recogerá en el hilo receptor del servidor y será almacenado en variables globales de las que el hilo transmisor hará uso.

Se ha de recibir por el puerto 11001 por protocolo UDP. Además el parámetro de comunicación de dicho mensaje es de tipo petición de servicio, por lo que una vez completado el movimiento habrá que generar una respuesta por el mismo puerto indicando que la comunicación es de respuesta a un servicio y con el tipo de respuesta exitoso, o en caso de completar el movimiento, o fallido en el caso contrario.

El tipo de dato que transporta es el generado por Moveit *joint_trajectory_point* que cuenta con las coordenadas articulares objetivo de cada punto de la trayectoria y el tiempo en el que se tienen que completar.

Como se ha indicado antes será el hilo transmisor el que se encargará, además de transmitir las respuestas necesarias al robot, de realizar las órdenes de movimiento al controlador. Esto es así debido a que dicho hilo trabaja a más alta frecuencia que el de recepción y por limitaciones de diseño del robot las órdenes de velocidad tienen que realizarse a una frecuencia mayor de 300 ms.

6.2. Seguimiento de la trayectoria.

Como se ha indicado en el apartado anterior el mensaje que porta la orden de movimiento además incluye el tiempo en el que se tiene que completar la maniobra. Para unas primeras pruebas se intentó controlar el robot mediante posición a partir de los comandos propios de la librería del robot.

No obstante saltaba un error por parte del controlador de ROS el cual al ver que el robot completaba la trayectoria, pero tardaba demasiado en la ejecución, abortaba la trayectoria. Por ese motivo se procedió a controlar el robot en velocidad, no obstante el mensaje que se obtenía directamente de ROS no contenía las velocidades articulares por lo que se decidieron calcular de manera lineal entre la posición actual y la posición deseada.

Una vez obtenidas dichas velocidades articulares el objetivo estaba más cerca. Sin embargo debido a motivos de programación del robot, a la hora de controlarlo en velocidad, se le deben de enviar órdenes con una frecuencia de al menos 300 ms.

Este requerimiento creaba una limitación para ejecutar las órdenes en el propio hilo de recepción de los mensajes. Este motivo sumado a que pudieran haber colisiones a la hora de acceder a los recursos de comunicación del robot por parte de ambos hilos, hizo que se buscara una alternativa.

La alternativa alcanzada fue actualizar desde el hilo de transmisión las velocidades en variables globales accesibles desde los dos hilos y limitar las órdenes de acceso a la tarjeta de la máquina que se comunica con el controlador al hilo de recepción. De esta manera ya que éste trabaja a una frecuencia mayor se logra el objetivo de comandar velocidades a la frecuencia mínima deseada y se evitan problemas de acceso a memoria.

No obstante hay dos momentos en los que la ejecución ha de ignorar los valores incluidos en el mensaje que son: cuando haya una orden de parada, ya que todos los valores son cero y eso llevaría a velocidades infinitas, y cuando se recibe el último punto de la trayectoria donde las velocidades a comandar son cero. Por motivos de seguridad también se actualizan a velocidad cero las variables cuando se pulsa el accionador de emergencia.

Tras dichos ajustes la velocidad de seguimiento de la trayectoria seguía en algunos casos siendo insuficiente. Por este motivo se tomo en cuenta el tiempo de transmisión de la información por UDP y el procesamiento de los datos.

Dichas mediciones se llevaron a cabo con el programa Wireshark, resultando en tiempos de hasta 50 ms de retraso debido a la comunicación por UDP. Para compensar dicho tiempo de transimisión se decidieron menguar el tiempo de duración de cada movimiento al siguiente punto de la trayectoria en un 10 %.

Estos cambios resultaron ya que no se volvieron a detectar abortes de trayectoria en las siguientes tandas de pruebas. Y no fue la única mejora ya que se había detectado que en aquellas trayectorias que no resultaban abortadas, el error en la posición final era de entre 0.05 radianes correspondientes al tiempo que tardaba el sistema en recibir y procesar la orden de ROS, ya que era tiempo en el que se mantenía una velocidad distinta de 0.

Tras los cambios aplicados a los tiempos entre movimientos los resultados fueron los deseados con un error de 0.007 radianes.

6.3. Configuración de la ejecución de la trayectoria.

Dado que el modo final de control del robot ha sido en modo velocidad, conocer la manera de configurar la velocidad de control puede ser de gran utilidad.

Si bien con el entorno que se ha desarrollado no se entra a actuar sobre el topic de ROS en concreto que permitiría un control más preciso del robot, existe un archivo de configuración donde se establecen las velocidades máximas de las articulaciones, de donde moveit toma dichas restricciones para realizar sus cálculos.

Dicho archivo se denomina *joints_limits.yamls*, en ese archivo se establece si existen o no límites de velocidad y aceleración para cada articulación y además establece dichos valores en unidades del sistema internacional.

De esta manera sin necesidad de entrar dentro de la complejidad de los cálculos de moveit un usuario puede familiarizarse con el entorno gráfico de la herramienta y realizar movimientos en tiempo real y a la vez tener un control sobre la velocidad de los mismos.

Capítulo 7

Elementos del sistema

Llegado este punto del documento con todos los componentes explicados, parece importante hacer una recapitulación de todos los elementos que conforman el sistema y con las principales tareas que desempeñan para el buen funcionamiento del mismo.

Una visión global de los elementos que intervienen para la simulación y planificación de los movimientos del brazo robótico Mitsubishi-PA10 sobre ROS, se puede observar en la Figura 7.1.

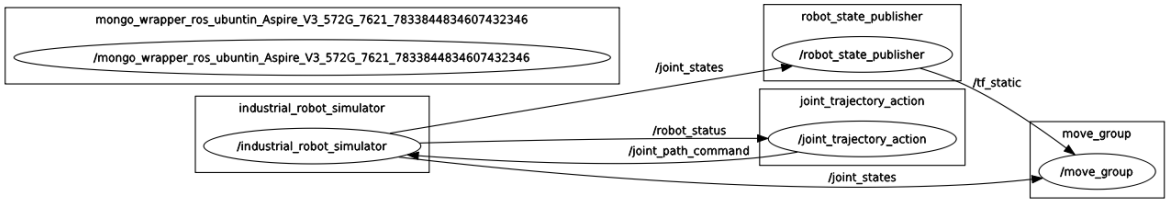


Figura 7.1: Nodos que intervienen en la simulación con ROS

En dicha figura se puede observar que existen cuatro nodos principales: *move_group*, *robot_state_publisher*, *joint_trajectory_action* e *industrial_robot_simulator*. Cada uno de ellos desempeña un papel importante en la simulación del Mitsubishi-PA10 en el entorno de ROS.

- **Move_group**: es el nodo encargado de generar los mensajes internos para el cálculo y planificación de trayectorias de moveit. Además también genera las órdenes de movimiento que son transmitidas a través del servicio de ROS *joint_path_command* de forma indirecta.

- **Robot_state_publisher**: este nodo se encarga de transformar, usando la descripción del robot, los datos de la actual posición de cada articulación del robot a datos tf que son comprendidos por moveit. Dichos datos son los que provienen de nuestro servidor en el controlador real o en su defecto del nodo *industrial_robot_simulator* a traves del topic *joint_states*.

- **Joint_trajectory_action**: el caso particular de este nodo es algo más complejo. Es el nexo de unión entre moveit y el controlador (real o simulado) del robot en cuestión. Se puede decir que traduce las planificaciones de moveit a ordenes del tipo *joint_path_command*, en función del estado del controlador, cuya información se encuentra en el topic *robot_status*.

Se trata de un tipo de servidor interno de ROS proveniente de la librería *actionlib*, el cual responde a las peticiones de moveit como si de un cliente se tratara.

- Por último el nodo **industrial_robot_simulator** se encarga de generar los mensajes de comunicación del robot según el protocolo *simple_message* como si de un controlador real se tratara. Para el caso en el que no se esté simulando el robot sino mandando órdenes al robot real, este nodo se dividirá en dos: *motion_streaming_interface* o *motion_download_interface* y el nodo *robot_state*.

El primero de ellos se encarga de enviar las órdenes que transporta el topic denominado *joint_path_command* y de iniciar la comunicación UDP. Mientras que el segundo se encarga de recibir periódicamente los mensajes de estado del controlador y de la posición de las articulaciones y suministrarlas al resto de nodos implicados en la simulación.

Cabe aclarar que existe un tercer topic que suministra este nodo, es el topic *feedback_states*. Dicho nodo está formado por datos actuales de posición, velocidad y aceleración de cada una de las articulaciones. Siendo difícil que robots con la antigüedad como la del que atañe a este proyecto posean sensores para monitorizar dichas variables, es un tipo de mensaje no usado por ROS en la actualidad.

En la práctica lo que ocurre es que con el mensaje recibido de *joint_states* genera el mensaje autocompletando con ceros el resto de variables. De manera que no es un mensaje que se tenga que generar en el servidor del Mitsubishi-PA10.

7.1. Puesta en marcha del sistema

Una vez conocidos todos los elementos que conforman el sistema y cómo se comunican entre ellos, es importante realizar una primera aproximación sobre la puesta en marcha del sistema.

Las herramientas con las que tenemos que contar son las siguientes: un PC con sistema operativo Linux, un PC con sistema operativo Windows compatible con el hardware necesario para manejar el robot por medio de su controlador y una red local inalámbrica.

Cuando se habla del hardware necesario para manejar el robot, se quiere decir, la tarjeta compatible con el controlador del brazo robótico Mitsubishi-PA10 y la versión de Windows compatible con dicha tarjeta, en nuestro caso Windows XP.

En cuanto a la versión de Linux, se ha utilizado la versión 14.04 por razones de compatibilidad con la versión de ROS instalada, Indigo.

Para instalar los paquetes de ROS basta con seguir los tutoriales que ofrece la propia organización y que se pueden encontrar en [6].

A parte de los paquetes principales de la distribución Indigo que estemos utilizando, será necesario instalar los paquetes de ROS-Industrial, y los paquetes del propio repositorio del robot que se han creado y que se encuentran en [7].

Para descargar los paquetes necesarios de ROS o de otro origen y que se encuentren en la plataforma github, con el comando *git clone* se podrán descargar y utilizar en el workspace donde se esté trabajando en ROS.

Con solo escribir *git clone* y la dirección web donde se encuentre el repositorio se podrán descargar los archivos y trabajar con ellos.

7.1.1. Ejemplo de trayectoria

Una vez se cuenta con todas las herramientas instaladas poner en marcha el simulador es sencillo. Basta con abrir dos terminales en el ordenador con Linux y lanzar los siguientes comandos:

- *source <ROS workspace >/devel/setup.bash*. Este comando sirve para cargar la configuración del workspace de ROS y que el ordenador sepa donde buscar los paquetes de ROS con los que se esta trabajando.

- *roscore*. Este comando sirve para cargar el nodo maestro de ROS y poder trabajar con sus herramientas.

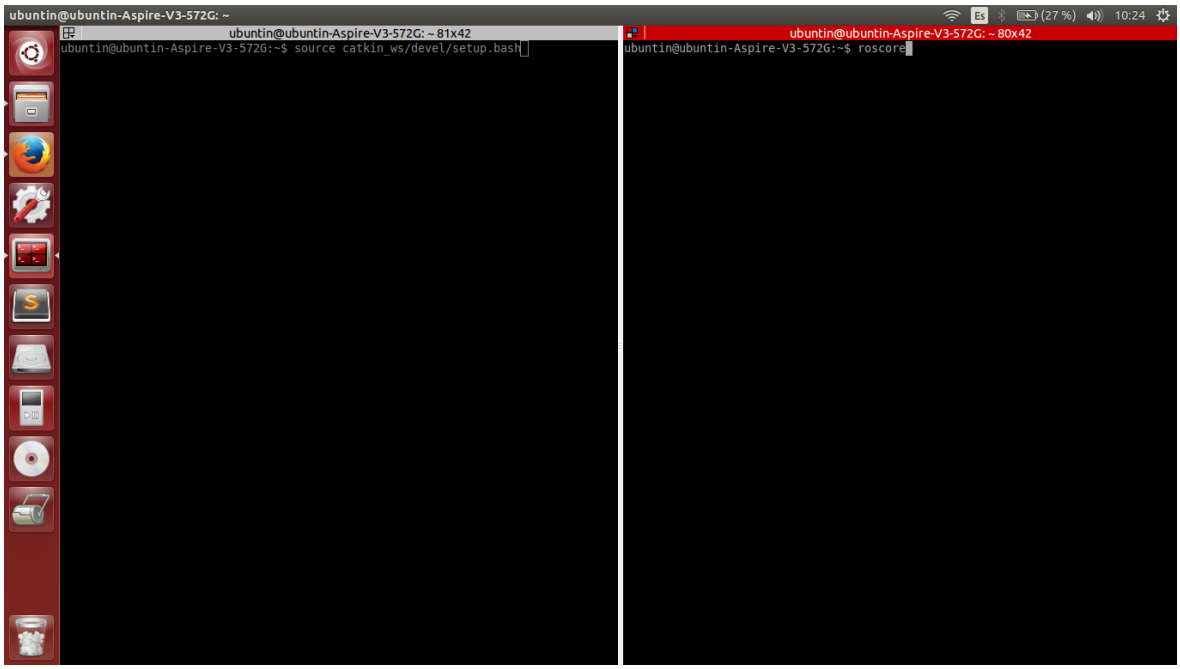


Figura 7.2: Primeros comandos en ROS para ejecutar los paquetes del robot.

Tras lanzar estos dos comandos como se observa en la Figura 7.2, se deberá lanzar el archivo launch que carga todos los nodos y herramientas de la simulación. Dicho comando es *roslaunch pa10_moveit_config pa10_moveit_execution_planning.launch robot_ip:=<PC Windows ip>sim:= <true or false >*

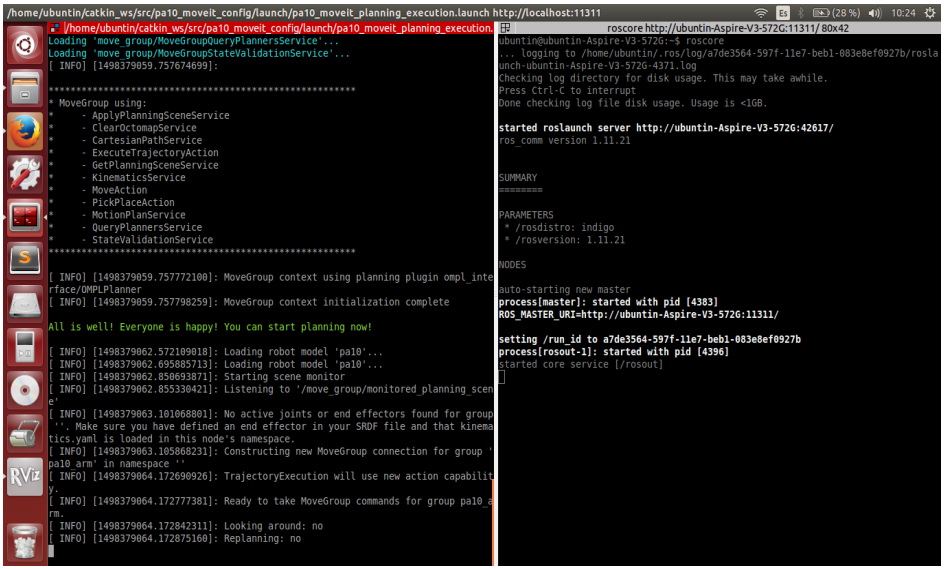


Figura 7.3: Resultado en consola de la puesta en marcha.

Este comando indica que se va a lanzar un archivo tipo *launch*, el paquete donde se encuentra ese archivo *pa10_moveit_config*, el nombre del archivo *pa10_moveit_execution_planning.launch* y los parámetros que solicita ese archivo, la dirección ip del pc con Windows donde se encuentra el controlador real del robot y un booleano donde se indica si deseamos simular la cinemática o realizarla en el robot real.

El resultado se muestra en la Figura 7.3 y 7.4. En la herramienta Rviz se visualiza el modelo simulado del robot.

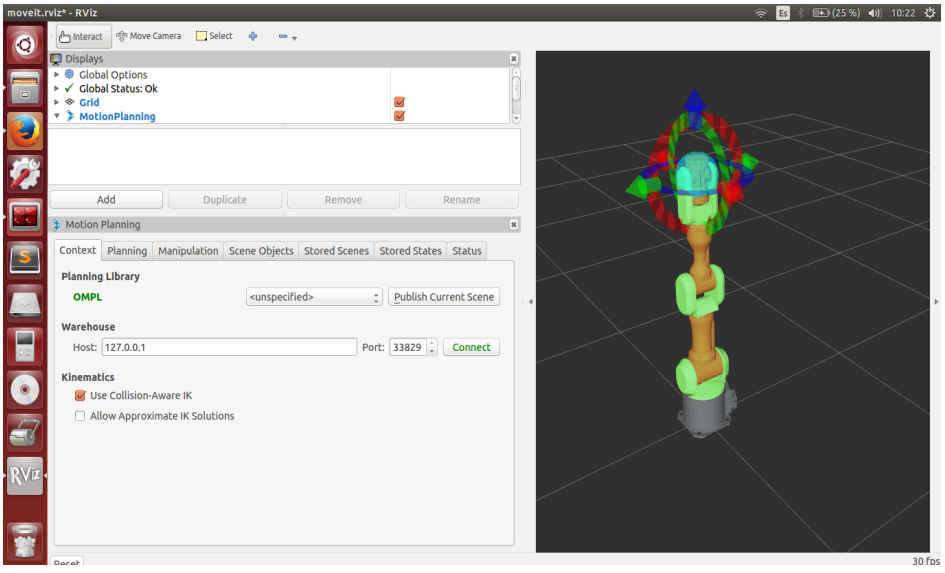


Figura 7.4: Resultado de simulación del robot en Rviz.

Para realizar una primera trayectoria se deben seguir los siguiente pasos. Dentro de la ventana de Motion Planning en Rviz hacer clic en la pestaña de Planning.

Tras ello, se debe mover con el asistente visual el robot a una posición distinta como la mostrada en la Figura 7.5.

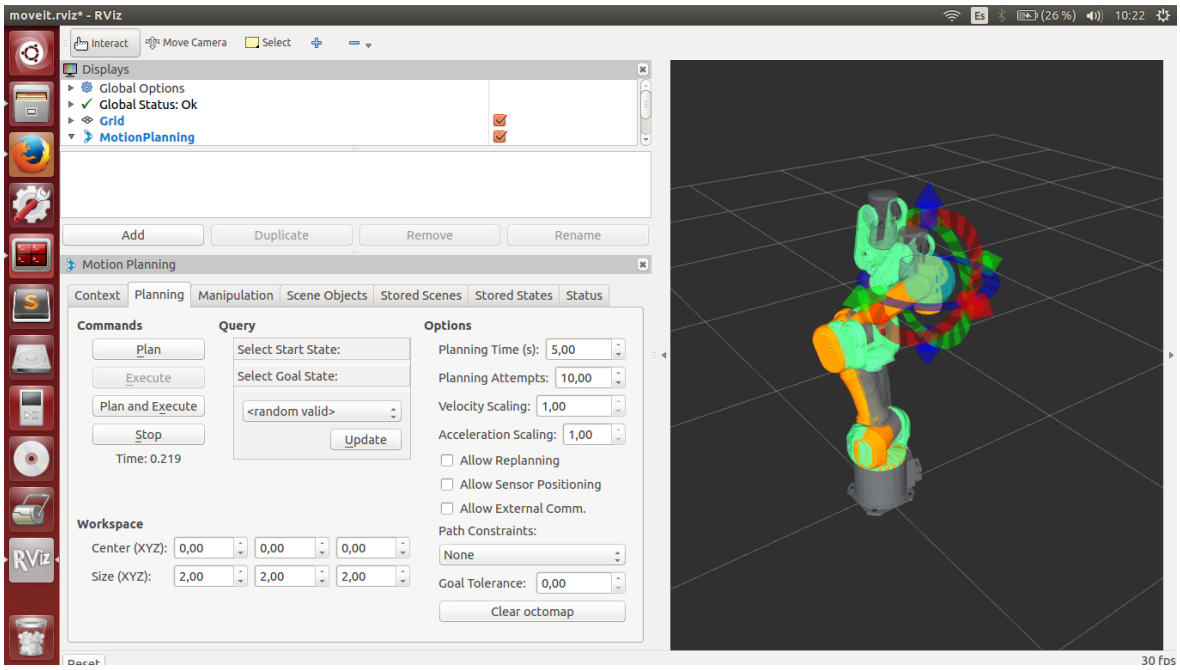


Figura 7.5: Planificación de la trayectoria.

Tras ello y pulsando en el botón *Plan* se podrá observar en la terminal si la trayectoria es realizable y en su caso una segunda representación en movimiento dentro del asistente visual de la trayectoria que seguirá el robot.

Si tras ello se pulsa el botón *Execute* se ejecutará la trayectoria. En el caso de encontrarnos en simulación este proceso habrá finalizado y podremos ver la posición en cada momento durante el movimiento en el asistente.

Por el contrario si estamos moviendo el robot real, en el momento de pulsar *Execute* el robot comenzará a moverse, siempre y cuando esté encendido el controlador y esté en funcionamiento la aplicación del servidor desarrollada.

Por lo tanto se ruega siempre que se esté cerca del pulsador de parada de emergencia por motivos de seguridad.

Como se puede observar la única diferencia entre el modo de simulación y el robot real es el argumento booleano *sim* en el comando *launch*, además de por otra parte estar funcionando el servidor dentro del PC de Windows. Para lo cual solo se ha de hacer doble clic en el ejecutable del mismo, o en su defecto descargarse el proyecto completo en Visual Studio 2010 y crear dicho ejecutable.

Capítulo 8

Resultados y Conclusiones.

La finalidad principal de este proyecto era obtener los paquetes y archivos necesarios para poder simular el brazo robótico Mitsubishi-PA10 dentro del entorno de ROS. Dicha meta como se ha demostrado en los capítulos anteriores a este, se ha logrado.

En este capítulo se pretende analizar en mayor profundidad con qué herramientas a parte de la de simulación de la cinemática RViz se ha conseguido integrar el brazo robótico y de qué manera. Tras ello se van a analizar posibles mejoras o futuros trabajos que se puedan derivar de éste.

En primer lugar se ha conseguido una descripción del robot y de la relación de cada uno de sus eslabones en un formato universal y ampliamente extendido como es URDF. Este formato tiene mucha relevancia ya que no sólo se puede utilizar dentro de ROS sino con otras herramientas de simulación como por ejemplo V-Rep, y otras muchas.

También es importante y de utilidad este formato ya que a partir de él se pueden obtener otros, mucho más potentes y que incluyen otros robots, y otras descripciones en el mismo entorno de simulación o escena.

En segundo lugar se han obtenido los archivos de configuración del robot para la herramienta planificadora moveit. Desde su creación moveit ha ido creciendo y tiene mucho apoyo dentro de la comunidad de desarrolladores de ROS. Por lo tanto el desarrollo de estos paquetes resulta de gran utilidad por su cesión al resto de la comunidad y por la potencia de la propia herramienta.

El siguiente logro de este trabajo ha sido implementar el protocolo definido en ROS conocido como *simple_message* de manera que no sólo se ha conseguido un entorno de simulación donde comprobar los movimientos del robot en el entorno de ROS sino llevarlos a cabo en la realidad mediante un servidor.

Dicho servidor está desarrollado para el sistema operativo Windows, en el lenguaje C++ haciendo uso de la librería que proporciona el propio fabricante a la hora de mandar órdenes de movimiento al controlador real.

El desarrollo del servidor en este entorno se realizó debido a un motivo principal y de peso. Dicho motivo es que la comunicación entre la tarjeta ISA que comunica el controlador con la computadora se realiza mediante la librería proporcionada para Windows, sin embargo este es uno de los posibles trabajos futuros, desarrollar un servidor en Linux completamente integrado dentro del entorno de ROS.

Esto conseguiría disminuir el retraso o error producido en la ejecución de la trayectoría que se debe a que el tiempo invertido en la comunicación es variable dependiendo de las tareas de los sistemas operativos de ambas máquinas, aquella con ROS en Linux y la que ejecuta el servidor en Windows.

Con estos principales objetivos cumplidos se ha conseguido un entorno de gran potencia y utilidad donde simular y calcular trayectorias y posteriormente llevarlas a cabo.

No obstante existen futuros trabajos o proyectos que pueden derivar de el que se ha realizado. Uno de ellos podría ser la inclusión del brazo robótico Mitsubishi-PA10 dentro de la herramienta de simulación dinámica Gazebo.

Para el caso de trayectorias a altas velocidades o con grandes cargas que puedan generar grandes inercias es una herramienta poderosa que permite simular escenas donde los esfuerzos y fuerzas que se puedan producir sobre las articulaciones son de gran importancia.

Por otro lado mejorar el servidor existente añadiendo una interfaz gráfica e incorporando más mensajes definidos en el protocolo *simple_message* a parte de los que ya se encuentran incluidos como son el definido como *joint_position* y *joint_trajectory_point*.

Por último indicar que todos los archivos desarrollados como resultado de este Trabajo Fin de Máster están recogidos en 2 repositorios, uno dedicado para los archivos del robot relativos a ROS y el segundo para los relacionados con el servidor que comunica ROS con el controlador. La dirección de ambos se indica en [7] y [8].

Apéndice A

Archivo de descripción URDF.

En este apéndice se adjunta el código en formato URDF del robot como ejemplo de archivo descriptivo del cual partir para futuras mejoras a este proyecto.

```
<?xml version="1.0" ?>

<robot name="pa10" xmlns:xacro="http://www.ros.org/wiki/xacro">


   <!-- Used for fixing robot to Gazebo 'base_link' -->
  <link name="world"/>

  <joint name="fixed" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
  </joint>

   <!-- Import all Gazebo-customization elements, including Gazebo colors -->
  <xacro:include filename="$(find pa10_description)/urdf/pa10.gazebo" />
  <!-- Import Rviz colors -->
  <xacro:include filename="$(find pa10_description)/urdf/materials.xacro" />



  <link name="base_link">
    <visual>
      <origin rpy="0 0 3.14" xyz="0 0 0"/>
      <geometry>
        <mesh filename="file:///home/ubuntin/pa10_stl/S1.stl"/>


<!-- podemos usar tambien en vez de mesh que requiere un archivo, cilindros
    cajas o esferas!!! -->

      </geometry>
      <material name="">
```

```
      <color rgba="0.34 0.35 0.36 1.0"/>
    </material>
  </visual>
  <collision>
    <origin rpy="0 0 3.14" xyz="0 0 0"/>
    <geometry>
      <mesh
          filename="file:///home/ubuntin/pa10_stl_collision/S1_collision.stl"/>
    </geometry>
  </collision>
  <inertial>
    <origin xyz="0 0 1" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia
   ixx="1" ixy="0.0" ixz="0.0"
   iyy="1" iyz="0.0"
   izz="1"/>
  </inertial>
 </link>
<link name="link_1">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0.117"/>
    <geometry>
      <mesh filename="file:///home/ubuntin/pa10_stl/S2.stl"/>

    </geometry>
 <material name="">
      <color rgba="0.33 0.96 0.53 1.5"/>
    </material>
  </visual>
  <collision>
    <origin rpy="0 0 0" xyz="0 0 0.117"/>
    <geometry>
      <mesh
          filename="file:///home/ubuntin/pa10_stl_collision/S2_collision.stl"/>
    </geometry>
  </collision>
  <inertial>
    <origin xyz="0 0 1" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia
   ixx="1" ixy="0.0" ixz="0.0"
   iyy="1" iyz="0.0"
   izz="1"/>
  </inertial>
 </link>
<link name="link_2">
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="file:///home/ubuntin/pa10_stl/S3.stl"/>
```

```
      </geometry>
      <material name="">
        <color rgba="0.34 0.35 0.36 1.0"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh
            filename="file:///home/ubuntin/pa10_stl_collision/S3_collision.stl"/>
      </geometry>
    </collision>
    <inertial>
      <origin xyz="0 0 1" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
     ixx="1" ixy="0.0" ixz="0.0"
     iyy="1" iyz="0.0"
     izz="1"/>
    </inertial>
  </link>
<link name="link_3">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0.133"/>
      <geometry>
        <mesh filename="file:///home/ubuntin/pa10_stl/E1.stl"/>

      </geometry>
   <material name="">
        <color rgba="0.33 0.96 0.53 1.5"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0.133"/>
      <geometry>
        <mesh
            filename="file:///home/ubuntin/pa10_stl_collision/E1_collision.stl"/>
      </geometry>
    </collision>
    <inertial>
      <origin xyz="0 0 1" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
     ixx="1" ixy="0.0" ixz="0.0"
     iyy="1" iyz="0.0"
     izz="1"/>
    </inertial>
  </link>
<link name="link_4">
    <visual>
```

```
      <origin rpy="0 0 0" xyz="0 0.003 0"/>
      <geometry>
        <mesh filename="file:///home/ubuntin/pa10_stl/E2.stl"/>

      </geometry>
    <material name="">
        <color rgba="0.34 0.35 0.36 1.0"/>
      </material>
     </visual>
     <collision>
      <origin rpy="0 0 0" xyz="0 0.003 0"/>
      <geometry>
        <mesh
            filename="file:///home/ubuntin/pa10_stl_collision/E2_collision.stl"/>
      </geometry>
     </collision>
     <inertial>
      <origin xyz="0 0 1" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
     ixx="1" ixy="0.0" ixz="0.0"
     iyy="1" iyz="0.0"
     izz="1"/>
    </inertial>
  </link>
<link name="link_5">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0.195"/>
      <geometry>
        <mesh filename="file:///home/ubuntin/pa10_stl/W1.stl"/>

      </geometry>
    <material name="">
        <color rgba="0.33 0.96 0.53 1.5"/>
      </material>
     </visual>
     <collision>
      <origin rpy="0 0 0" xyz="0 0 0.195"/>
      <geometry>
        <mesh
            filename="file:///home/ubuntin/pa10_stl_collision/W1_collision.stl"/>
      </geometry>
     </collision>
     <inertial>
      <origin xyz="0 0 1" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
     ixx="1" ixy="0.0" ixz="0.0"
     iyy="1" iyz="0.0"
     izz="1"/>
    </inertial>
```

```
  </link>
<link name="link_6">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="file:///home/ubuntin/pa10_stl/W2.stl"/>

      </geometry>
   <material name="">
        <color rgba="0.34 0.35 0.36 1.0"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh
            filename="file:///home/ubuntin/pa10_stl_collision/W2_collision.stl"/>
      </geometry>
    </collision>
    <inertial>
      <origin xyz="0 0 1" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
     ixx="1" ixy="0.0" ixz="0.0"
     iyy="1" iyz="0.0"
     izz="1"/>
    </inertial>
  </link>

  <link name = "ee_link"/>



<joint name="S1" type="revolute">
    <origin rpy="0 0 0" xyz="0 0 0.2"/>
    <parent link="base_link"/>
    <child link="link_1"/>
    <axis xyz="0 0 1"/>
    <limit effort="0" lower="-3.089" upper="3.089" velocity="1.0"/>
  </joint>
<joint name="S2" type="revolute">
    <origin rpy="0 0 0" xyz="0 0 0.117"/>
    <parent link="link_1"/>
    <child link="link_2"/>
    <axis xyz="0 1 0"/>
    <limit effort="0" lower="-1.64" upper="1.64" velocity="1.0"/>
  </joint>
<joint name="S3" type="revolute">
    <origin rpy="0 0 0" xyz="0 0 0.317"/>
    <parent link="link_2"/>
    <child link="link_3"/>
```

```
    <axis xyz="0 0 1"/>
    <limit effort="0" lower="-3.036" upper="3.036" velocity="1.0"/>
  </joint>
<joint name="E1" type="revolute">
    <origin rpy="0 0 0" xyz="0 0 0.133"/>
    <parent link="link_3"/>
    <child link="link_4"/>
    <axis xyz="0 1 0"/>
    <limit effort="0" lower="-2.39" upper="2.39" velocity="1.0"/>
  </joint>
<joint name="E2" type="revolute">
    <origin rpy="0 0 0" xyz="0 0 0.285"/>
    <parent link="link_4"/>
    <child link="link_5"/>
    <axis xyz="0 0 1"/>
    <limit effort="0" lower="-4.45" upper="4.45" velocity="1.0"/>
  </joint>
<joint name="W1" type="revolute">
    <origin rpy="0 0 0" xyz="0 0 0.195"/>
    <parent link="link_5"/>
    <child link="link_6"/>
    <axis xyz="0 1 0"/>
    <limit effort="0" lower="-2.878" upper="2.878" velocity="1.0"/>
  </joint>
  <joint name = "W2" type="revolute">
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <parent link="link_6"/>
    <child link="ee_link"/>
    <axis xyz="0 0 1"/>
    <limit effort="0" lower="-2.878" upper="2.878" velocity="1.0"/>
  </joint>



</robot>
```

Apéndice B

Archivo de lanzamiento del asistente moveit.

En este apéndice se muestra el archivo para lanzar por completo los nodos y elementos de configuración del robot tanto en simulación como con el controlador real.

```
<launch>
  <!-- The planning and execution components of MoveIt! configured to run
     -->
  <!-- using the ROS-Industrial interface. -->

  <!-- Non-standard joint names:
       - Create a file [robot_moveit_config]/config/joint_names.yaml
           controller_joint_names: [joint_1, joint_2, ... joint_N]
       - Update with joint names for your robot (in order expected by rbt
           controller)
       - and uncomment the following line: -->
  <rosparam command="load" file="$(find
     pa10_moveit_config)/config/joint_names.yaml"/>

  <!-- the "sim" argument controls whether we connect to a Simulated or
     Real robot -->
  <!--  - if sim=false, a robot_ip argument is required -->
  <arg name="sim" default="true" />
  <arg name="robot_ip" unless="$(arg sim)" />

  <!-- load the robot_description parameter before launching ROS-I nodes -->
  <include file="$(find pa10_moveit_config)/launch/planning_context.launch"
     >
    <arg name="load_robot_description" value="true" />
  </include>

  <!-- run the robot simulator and action interface nodes -->
  <group if="$(arg sim)">
    <include file="$(find
       industrial_robot_simulator)/launch/robot_interface_simulator.launch"
```

```
      />
  </group>

  <!-- run the "real robot" interface nodes -->
  <!--   - this typically includes: robot_state, motion_interface, and
      joint_trajectory_action nodes -->
  <!--   - replace these calls with appropriate robot-specific calls or
      launch files -->
  <group unless="$(arg sim)">
    <include file="$(find
        industrial_robot_client)/launch/robot_interface_download.launch" >
      <arg name="robot_ip" value="$(arg robot_ip)"/>
    </include>
  </group>

  <!-- publish the robot state (tf transforms) -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
      type="robot_state_publisher" />

  <include file="$(find pa10_moveit_config)/launch/move_group.launch">
    <arg name="publish_monitored_planning_scene" value="true" />
  </include>

  <include file="$(find pa10_moveit_config)/launch/moveit_rviz.launch">
    <arg name="config" value="true"/>
  </include>

  <include file="$(find
      pa10_moveit_config)/launch/default_warehouse_db.launch" />

</launch>
```

Bibliografía

[1] R.I. ROS-Industrial. Ros industrial website. http://wiki.ros.org/Industrial. Accessed: 2017-06-26.

[2] R. I. ROS-Answers. Ros answers website. http://answers.ros.org/questions. Accessed: 2017-06-26.

[3] R.I. Industrial-Moveit-Organization. Ros moveit website. http://moveit.ros.org/. Accessed: 2017-06-26.

[4] R.I. ROS-Organization. Ros industrial tutorials website. http://wiki.ros.org/Industrial/Tutorials/Create_a_MoveIt_Pkg_for_an_Industrial_Robot. Accessed: 2017-06-26.

[5] G. A. Vanderhoorn(g.a.vanderhoorn@tudelft.nl). Repositorio protocolo simple message. https://github.com/gavanderhoorn/rep-ros-i/blob/retrospective_simple_msg/rep-ixxxx.rst#joint-traj-pt. Accessed: 2017-06-26.

[6] Ros Organization Tutorials. Tutoriales de iniciacion en ros. http://wiki.ros.org/ROS/Tutorials. Accessed: 2017-06-26.

[7] Martin Martin Meiras. Archivos de descripcion en ros mitsubishi-pa10. https://github.com/meimar18/PA10ControllerServer. Accessed: 2017-06-26.

[8] Martin Martin Meiras. Archivos servidor mitsubishi-pa10. https://github.com/meimar18/PA10ControllerServer. Accessed: 2017-06-26.