Universidad Zaragoza

Trabajo Fin de Grado

Manipulación de objetos deformables en entornos multi-robot

Deformable object manipulation in multi-robot environments

Autor

Andrés Otero García

Directores

Gonzalo López Nicolás

María del Rosario Aragüés Muñoz

Grado en Ingeniería Informática
ESCUELA DE INGENIERÍA Y ARQUITECTURA
2021

AGRADECIMIENTOS

Agradezco a:

mis tutores Gonzalo y Rosario, por su pasión y ganas de ayudarme, además de hacer que este Trabajo haya sido mucho más ameno.

mis profesores, sobre todo a aquellos que me han guiado por el camino y no se han conformado con sólo resolver mis dudas.

mi padre José Antonio, por su gran apoyo en todas las decisiones que hasta ahora he tenido que tomar en la vida y su interés para resolver cualquier problema.

mi madre Alicia, por todo su amor y la ayuda que me ha prestado en todas las etapas de mi vida, incluyendo la académica, por todas esas tardes conmigo para hacer los deberes y aprender siempre más.

mi hermana Alicia, por su apoyo incondicional, por siempre ser un perfecto ejemplo a seguir y por toda la alegría que me ha traído, especialmente en los momentos que más los necesitaba de mi etapa universitaria.

también al resto de mi familia, por siempre hacerme tan feliz.

Santi, por acompañarme siempre con una sonrisa en los momentos en los que más lo necesitaba y por ser el mejor compañero que podría pedir en mi vida universitaria.

Melany y Enri, por haberme acogido como un amigo más, por las largas sesiones de estudio juntos y por todos los buenos momentos.

Fernando, Pedro, Alba, Néstor, Cristina, Irene y Leti, por acompañarme desde el principio hasta el final, que han hecho que el Grado sea una experiencia mucho más agradable.

al resto de mis compañeros y amigos, de Zaragoza y de Cork, por escucharme siempre y estar ahí, a pesar del paso del tiempo.

Gracias.

Objetivos, herramientas y resultados

RESUMEN

Se estudia un caso en el que múltiples robots IRB120 de la empresa ABB manipulan simultáneamente un objeto de tipo deformable, como pudiera ser una tela. El objetivo es desarrollar una simulación en el que al menos dos robots puedan realizar movimientos en sincronización para realizar cambios en el objeto deformable, como pudieran ser un desplazamiento o una deformación, sin que estos colisionen con el objeto durante la manipulación. Se plantea además que sea una simulación que no esté intrínsecamente relacionada con los robots que se vayan a utilizar, y que permita enviar comandos de forma independiente a todos los robots. Se utiliza para esto el sistema operativo robótico (ROS™) para la comunicación entre las distintas partes de la simulación, mientras que para albergar el entorno de la simulación, así como sus físicas, se utiliza la herramienta Gazebo. Para la planificación de trayectorias de movimientos del robot se ha empleado la biblioteca MoveIt, y para la realización de pruebas, la interfaz RViz. Se ha desarrollado una serie de programas que simulan los robots de forma independiente, así como un objeto deformable con un modelo Masa-Muelle-Amortiguador y se han realizado múltiples pruebas para evaluar su comportamiento. Los primeros programas ejecutan diversos movimientos en un entorno de varios robots de forma independiente, mientras que también se han desarrollado otras pruebas, en las que dos robots mueven un objeto de tipo tela elástica a la vez, moviéndola y estirándola de formas diversas para finalmente soltarla. Además, se ha realizado una implementación para observar el comportamiento de la tela en presencia de gravedad, con las cuatro esquinas sin movimientos permitidos. Podemos observar que los comportamientos son los esperados para una tela y los robots son capaces de evitar colisiones con la misma.

Índice

Capítulo 1

Introducción y objetivos

Desde su aparición en el año 1938 [1], los robots industriales han despertado un gran interés por sus capacidades para realizar diferentes tareas de forma automática y precisa que han mejorado el rendimiento de las fábricas [2]. Así, es normal ver múltiples robots a lo largo del proceso industrial que pueden llegar a colaborar en muchas ocasiones para conseguir realizar maniobras que serían más difíciles o imprecisas si se realizaran con un único robot [3] [4].

Por otro lado, la programación de tareas para un robot manipulando objetos rígidos es la más común y que más avanzada se encuentra, en contraposición a los objetos deformables, que son aquellos que, sometidos a fuerzas externas, son capaces de cambiar de forma y, en función de su elasticidad, de volver a su forma inicial en ausencia de éstas, es decir, son flexibles (en la Figura 1.1 se observa como dos robots manipulan un plátano, que es un objeto deformable) [3]. Estos objetos suponen un problema mayor que los rígidos, pues no es sencillo determinar en qué posición se encontrarán las partes que componen el objeto, mientras que en un objeto rígido siempre se mantiene la misma proporción entre distintos puntos del mismo. Es por esto que los objetos deformables son un campo en el que la investigación sigue avanzando y se desarrollan modelos y métodos nuevos para encontrar mejores soluciones a las que tradicionalmente se les ha ido dando a tareas pequeñas, como doblar o cortar telas [4] [5] [6].

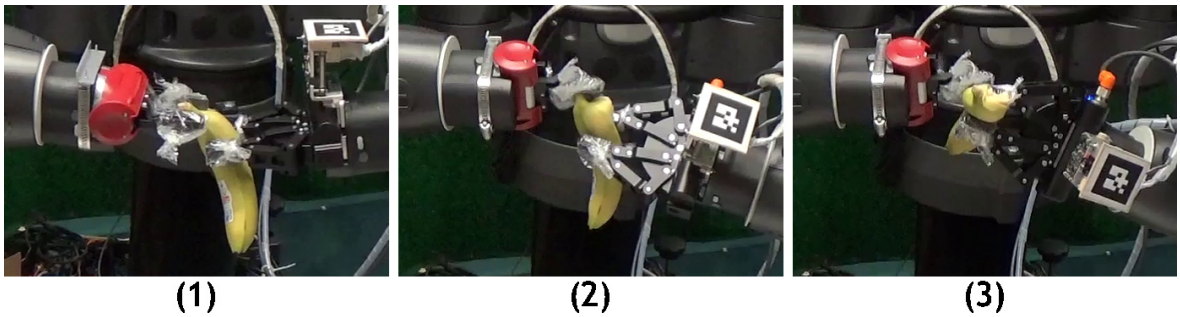


Figura 1.1: Manipulación de un plátano por dos brazos robóticos [7]

Existen además, gran cantidad de formas de hacer que un robot realice movimientos repetitivos por medio de ingeniería automática (con dispositivos mecánicos, neumáticos, hidráulicos, electrónicos...) y sistemas más complejos como autómatas programables o controladores lógicos programables [8]. Sin embargo, todas estas soluciones, a pesar de ser robustas y generalmente sencillas de mantener, también son siempre dependientes del robot que se utilice, así como de todos los componentes físicos y de *hardware* para el correcto funcionamiento del sistema automático.

1.1. Objetivos

En este Trabajo se intenta dar una solución genérica a la manipulación con varios robots industriales de objetos deformables, en particular los de tipo tela elástica, y que pueda ser extendida sencillamente a cualquier otro robot que se desee utilizar y también al entorno en el que se desarrolle el proceso industrial. Se marca como objetivo el aprendizaje del Sistema Operativo Robótico (*ROS*™) y su uso para la simulación, que permite la creación de entornos con robots industriales, junto con el simulador de físicas Gazebo y la biblioteca de manipulación de robots MoveIt. Éstos se pueden combinar para obtener una simulación de robots industriales, así como para controlar robots reales y programar sus tareas de un modo que no dependa del robot utilizado ni de las conexiones entre éste y el ordenador que lo controle (véase Capítulo 2).

Se plantea la realización de una simulación con al menos dos robots (pero que permita la utilización de un número arbitrario) en los que ambos puedan ser controlados de manera independiente y que consigan tomar el objeto para desplazarlo y que se produzcan deformaciones en el mismo (estirarlo o comprimirlo). También se desea una implementación realista pero sencilla para el comportamiento del objeto de tipo tela elástica y también parametrizada, para poder cambiar las características del objeto en diferentes ejecuciones.

Finalmente, todas las pruebas deberán funcionar con el modelo IRB 120 de la empresa de tecnologías ABB, que posee 6 grados de libertad y es el último robot desarrollado por la misma en la fecha de redacción de este Trabajo de Fin de Grado [9]. Además, la propia empresa ha desarrollado una serie de archivos para su utilización en ROS, Gazebo y MoveIt, si bien no están verdaderamente adaptados para un entorno multi-robot. Por este motivo, será necesario modificarlos, pero se intentará que sea lo mínimo posible. Cabe destacar también que se poseen dos ejemplares del robot en el laboratorio L0.06 del edificio Ada Byron de la Universidad de Zaragoza, por lo que se podría llegar en un futuro a realizar pruebas con éstos sin realizar muchos cambios en la estructura de los archivos desarrollados.

1.2. Organización de esta Memoria

Esta memoria se ha redactado siguiendo una estructura que pueda ayudar al lector a entender el proceso con el que se desarrolló el proyecto. De esta forma, en primer lugar se describen las herramientas utilizadas en el Capítulo 2. Posteriormente se encuentran las diferentes fases del proceso, comenzando por la configuración y puesta en marcha de una simulación con varios robots, presentes en el Capítulo 3, pasando por el desarrollo del objeto deformable y su inclusión en la simulación en el Capítulo 4, y su comportamiento en el Capítulo 5, para finalmente explicar cómo interactuarán los robots y el objeto en el Capítulo 6. Después se dedica el Capítulo 7 a describir experimentos llevados a cabo así como sus resultados y el Capítulo 8 a detallar las conclusiones obtenidas y trabajo futuro que podría realizarse para extender o mejorar los resultados y la usabilidad de este Trabajo. Cabe destacar que o bien dentro de todos los capítulos, o bien en el Apéndice C se detallan todas las soluciones que se han intentado dar a los objetivos planteados, empezando por el primer intento y explicando sus resultados hasta llegar a la solución que ha permitido un comportamiento correcto del sistema.

Se incluyen además otros cuatro anexos para completar algunos aspectos, como la estructura de los ficheros desarrollados en el Apéndice A, la forma de instalarlos y realizar las pruebas en el Apéndice B, unas pautas o recomendaciones por si se desea modificar o extender los ficheros en el Apéndice E y un listado de algunos de los fragmentos de código que pudieran ser más interesantes para el lector, en el Apéndice D.

Capítulo 2

Herramientas

Para la puesta en marcha de la simulación planteada, se han empleado unas semanas para aprender a utilizar varias herramientas. En concreto, el sistema ROS™, y los programas Gazebo y MoveIt y cómo interactúan entre ellos. A continuación, se dedica este capítulo a describir su funcionamiento y algunos conceptos de utilidad que aparecerán en múltiples ocasiones en el resto de esta memoria.

2.1. *Robot Operating System* (ROS™)

El sistema operativo robótico, más conocido por sus siglas en inglés ROS, es un entorno de trabajo de código abierto creado para el diseño y desarrollo de software en el ámbito de la robótica [10]. Surgió a raíz de los esfuerzos de varias instituciones, como del *STanford Artificial Intelligence Robot* (*STAIR*) de la Universidad de Stanford o el programa *Personal Robots* (PR) del MIT [11] y actualmente está principalemente operada por *Open Robotics* [12] (antes conocida como *Open Source Robotics Foundation* [13]). Desde entonces ha crecido hasta ser vastamente utilizado en el entorno de investigación en robótica, así como utilizado por grandes empresas, como Bosch o BMW [14] [15].

En su nivel más inferior, ROS está basado en una interfaz de paso de mensajes (*MPI*), es decir, posee **nodos** que ejecutan ciertas funciones o actividades e intercambian información entre ellos por medio de **mensajes**, permitiendo así el trabajo de manera distribuida [16]. De esta manera, ROS resulta muy adecuado para proyectos con entornos multi-robot, puesto que se pueden controlar o simular varios robots de forma separada e independiente. Además, ABB ofrece paquetes para el control de muchos de sus robots industriales, incluido el IRB 120 [17]. Se utilizará la distribución *Melodic*, compatible con la versión de Ubuntu a utilizar, que en este caso es Ubuntu 18.04 [18]. A continuación se explica en detalle cómo ocurren dichos intercambios.

2.1.1. Intercambio de información en ROS

En ROS, cada nodo es un proceso que realiza cierta computación. Además, siempre existe un proceso denominado ***master***, que aloja la información imprescindible para que los nodos sean capaces de establecer conexiones entre ellos y también otros parámetros de ejecución. Un nodo se conecta con el *master* para obtener la manera con la que se podrá conectar a otro nodo (en gran parte de las ocasiones, las conexiones se realizan por el protocolo TCP/IP), así, el *master* actúa como un servidor *DNS* [19].

Los paquetes de información o mensajes siempre tienen un formato determinado a priori y se distribuyen por un "tema" o ***topic***. Un *topic* es un bus de datos por el que se puede pasar un tipo de mensajes. Así, un nodo puede "subscribirse" a un *topic* si desea recibir la información que es enviada a través de este, o puede "publicar" mensajes en dicho *topic* [20].

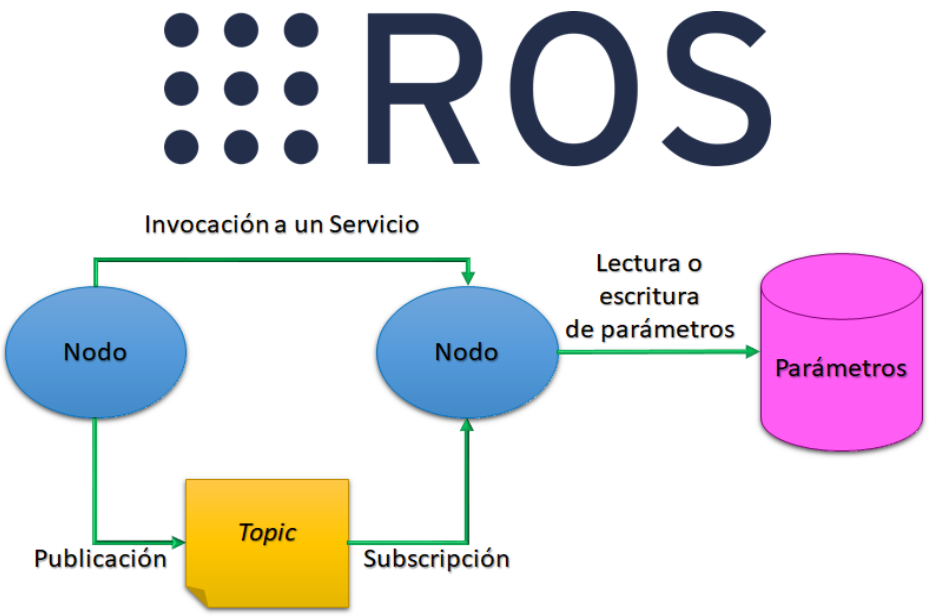


Figura 2.1: Esquema del tratamiento de información en ROS. Logo ROS: [21]. Esquema basado en: [19]

Sin embargo, este método no permite realizar intercambios de tipo "petición/respuesta" de forma inmediata, puesto que se trata de un sistema de intercambio *many-to-many* (es decir, la conexión se realiza entre múltiples nodos por lo que no existe un destinatario determinado) y es de un solo sentido. Por ello, cada nodo puede también anunciar sus propios **servicios**. Un servicio es un método de llamadas a procedimientos remotos (*RPC*) definida por un par de tipos de mensajes, uno para la petición y otro para la respuesta. Esto es, un nodo anuncia un servicio (bajo un nombre determinado) con un requerimiento sobre el tipo de mensaje que se debe mandar al establecer la comunicación. Al recibir la petición, el nodo realizará

cierta computación con los datos de la misma, y enviará otro mensaje del tipo especificado con el resultado [19].

Finalmente, cabe destacar la presencia de un **servidor de parámetros** en el *master*. Se trata de un conjunto de variables nombradas que pueden ser creadas, modificadas o leídas por cualquier nodo que se conecte al *master*. Así es posible la persistencia de datos para un intercambio asíncrono de información [22]. Se ha incluido un esquema del comportamiento descrito anteriormente, en la Figura 2.1.

2.2. Gazebo

Gazebo es un simulador de la dinámica de objetos y robots en tres dimensiones, que ofrece la habilidad de simular múltiples robots en diferentes entornos, además de simulación de físicas, utilización de sensores (como cámaras o láseres [23]) e interfaces de usuario y programática [24]. Podemos observar la interfaz gráfica en la Figura 2.2.

En él, existen modelos que están formados por enlaces o ***links*** que representan las características visuales y físicas de distintas partes del objeto, así como por articulaciones o ***joints*** que son las que unen los enlaces del objeto y permiten el movimiento de los mismos.

Se encuentra integrado en el sistema ROS gracias a la existencia de los paquetes *gazebo_ros_pkgs*, que aportan las interfaces necesarias para conseguir la simulación de un robot en Gazebo mediante mensajes y servicios de ROS [25]. Además, ABB proporciona paquetes ROS para la simulación de los robots IRB 120 en Gazebo [26]. Se utilizará la versión Gazebo 9.0.0, recomendada con ROS Melodic [27].

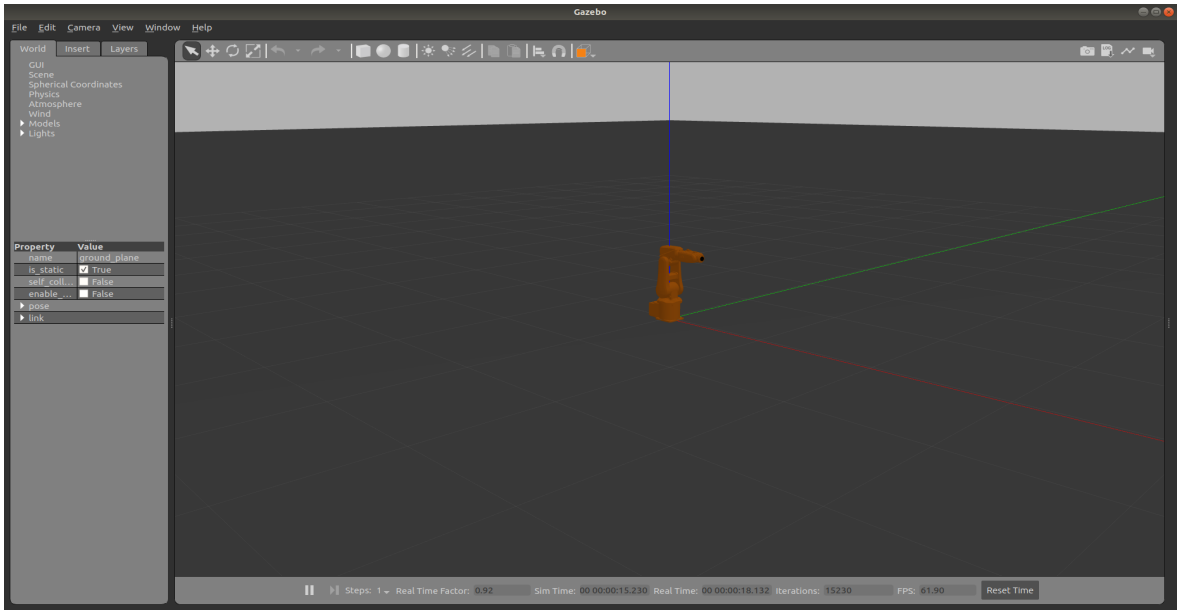


Figura 2.2: Captura del entorno de simulación Gazebo con un robot ABB IRB120

2.3. *MoveIt Motion Planning Framework*

MoveIt es una plataforma de código abierto destinada a la manipulación de robots con ROS [28]. Contiene las funcionalidades del cálculo de cinemáticas inversas, planificación de movimientos y evitación de obstáculos [29].

MoveIt ofrece tanto interfaces para poder ser utilizada de forma programática, mediante las interfaces ofrecidas para C++ y Python [30], como mediante la interfaz de usuario RViz, un visualizador 3D con diferentes métodos para manipular robots y otros objetos [29].

Resulta de principal interés el cálculo de las cinemáticas inversas y la planificación de movimientos para este proyecto, ya que de esta manera, podremos hacer uso tanto de la interfaz de programación en C++ para conseguir el movimiento de los robots, como de la interfaz de usuario RViz para la depuración manual de la configuración. Además, ABB ofrece paquetes de ROS preparados para la utilización de MoveIt y RViz para la manipulación de sus robots industriales, entre ellos el IRB 120 [31]. En la Figura 2.3 podemos observar un robot realizando un movimiento en la interfaz RViz.

Se utilizará la versión de MoveIt para ROS Melodic [32], junto con el lenguaje de programación C++, en su versión 11.

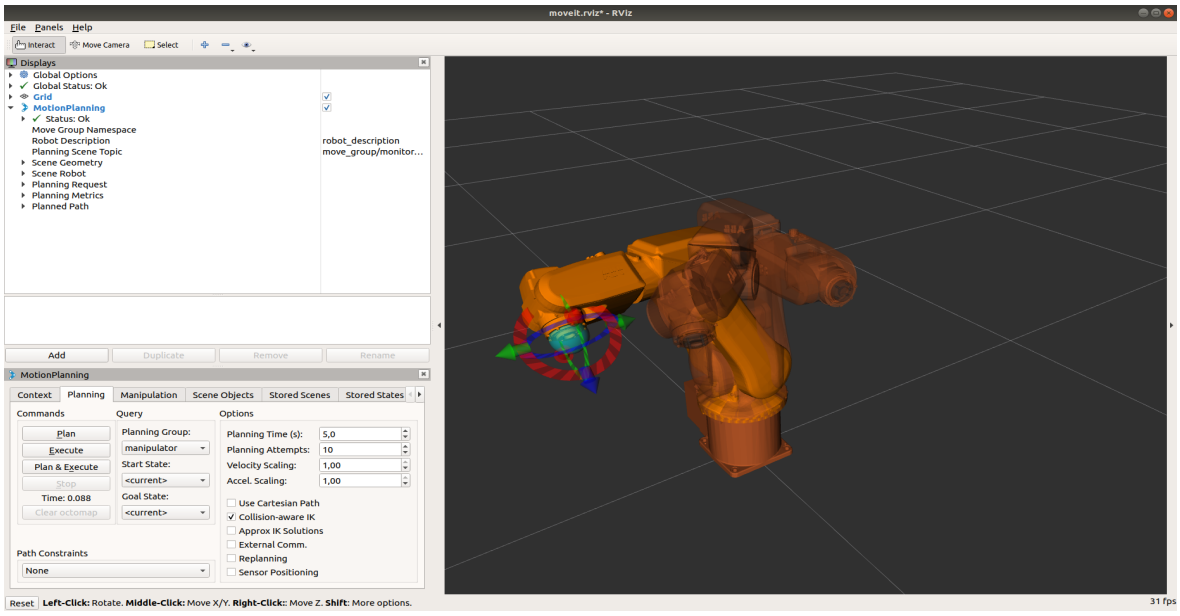


Figura 2.3: Captura del entorno de manipulación RViz. Se muestra un robot ABB IRB120 ejecutando una trayectoria (de la posición de la derecha hasta la de la izquierda, pasando por la del centro.

Capítulo 3

Configuración del entorno multi-robot

ROS es compatible con robots de diversos tipos, como robots aéreos, componentes electrónicos, robots terrestres, manipuladores industriales e incluso robots marítimos [33]. Entre sus manipuladores se encuentran los de varias empresas, como Fanuc, Universal Robots, Kuka y ABB [34]. Para desarrollar este Trabajo, se plantea una simulación con al menos dos robots que sea compatible con los robots IRB 120 de la empresa ABB, que tienen 6 grados de libertad, son ligeros, compactos y pueden realizar movimientos precisos a gran velocidad, por lo que podrían resultar adecuados para manipular objetos deformables que son más impredecibles que los rígidos [3] [9]. En la Figura 3.1 se pueden observar tres ejemplares en distintas configuraciones posibles.

Figura 3.1: Tres robots IRB 120 en distintas configuraciones. Origen: [35].

Tanto ABB como el resto de empresas ofrecen paquetes para el control de sus robots en ROS, pero son en gran parte para controlar exclusivamente un robot por simulación. Resulta bastante sencillo utilizar varios robots de diferentes modelos sin cambiar los ficheros. Sin embargo, existe una serie de problemas en los ficheros que no permiten introducir varios robots del *mismo modelo* en una única simulación, por lo

que es necesario realizar varios cambios en los mismos.

Finalmente, ROS ofrece un tutorial para la utilización de su interfaz, así como de RViz para entornos multi-robot [36]. El método de afrontar el problema de varios robots en dicho tutorial supone la creación de un nuevo robot que a su vez contenga ambos robots y asegurándose de que todas las articulaciones tienen nombres únicos. Esto supondría que si se quisiese añadir un nuevo robot, sería preciso repetir este proceso y por tanto no sería una solución muy flexible ni escalable. Por ello, se plantea una solución en la que los robots se encuentren diferenciados en el espacio de nombres de ROS, que puedan ser controlados individualmente por diferentes instancias de MoveIt y RViz, pero que se encuentren en el mismo entorno de Gazebo. A continuación se describe la problemática con los archivos originales, y algunas de las modificaciones necesarias para el correcto funcionamiento de este entorno.

3.1. Paquetes de ROS para ABB IRB 120

Como se ha mencionado en el capítulo anterior, ABB proporciona los paquetes necesarios para el control y la simulación con Gazebo y MoveIt en ROS para múltiples de sus robots, incluyendo el ABB IRB 120.

Los nombres de estos paquetes son:

- *abb_irb_120_support*: para el control del robot por medio de ROS [17].
- *abb_irb_120_gazebo*: para la simulación del robot en Gazebo [26].
- *abb_irb_120_moveit_config*: para la manipulación del estado del robot por medio de MoveIt y RViz [31].

En ellos, existen archivos de diferentes tipos:

- Archivos *launch*: Archivos en el formato XML que son utilizados para el lanzamiento de múltiples nodos y para cargar parámetros en el servidor de ROS, necesarios para la simulación o control de robot real [37]. Se ejecutan por el comando *"roslaunch"* [38].
- Archivos *yaml*: Archivos que poseen diferentes datos, como las articulaciones del robot (véase [39]) y sus características físicas (véase [40]) o el algoritmo para calcular cinemáticas (véase [41]), entre otros.
- Archivos en formato *srdf* o *urdf*: Archivos que poseen la descripción del aspecto, características físicas y articulaciones del modelo para su representación en la simulación. Hacen uso de los lenguajes universales de descripción de robots URDF

[42] y SRDF [43], así como lenguaje de macros XML "xacro", que permite una mayor parametrización de dichos lenguajes [44].

Si bien es cierto que algunos de estos archivos poseen indicios de una estructura para la utilización de los mismos en un entorno con múltiples robots (por ejemplo, el archivo para la descripción del robot y sus articulaciones *irb120_3_58_macro.xacro* [45] añade un prefijo especificado por un parámetro al nombre de todas las articulaciones y enlaces), su uso sin modificaciones no es posible ya que la mayoría de éstos no están adaptados a una situación en la que hubiera recursos duplicados, tales como *topics*, nodos, enlaces, articulaciones o el propio modelo del robot (dos recursos no pueden compartir nombre en el mismo nivel de la jerarquía de nombres [46]). E incluso si fuera posible, los controladores no sabrían qué robot o articulación deberían controlar. Además, MoveIt y RViz están diseñados para el control de un único robot, por lo que esta tarea no es trivial.

3.1.1. Puesta en marcha de un robot en Gazebo y MoveIt

El paquete *gazebo_ros* ofrece varios nodos y servicios cuya funcionalidad es colocar un modelo, ya sea un objeto cualquiera o un robot, en el entorno de simulación de Gazebo [47].

En el caso de los paquetes de ABB, se hace uso del nodo *spawn_model* al que se le debe pasar un parámetro del servidor denominado *robot_description* que contenga toda la información del robot [48]. También es necesario la inicialización de un nodo *robot_state_publisher*, que se encarga de calcular las cinemáticas directas del robot (haciendo uso de las posiciones de las articulaciones del robot) para poder ser usadas por el entorno RViz entre otros [49]. Por último, será necesario cargar los controladores para poder manipular el robot.

El primer controlador es un *joint_state_controller*, que se encarga de leer la posición de todas las articulaciones y publicarla en un topic [50] [51]. El segundo es un *joint_trajectory_controller*, que se utiliza para enviar trayectorias completas al robot [51].

Para controlarlo con MoveIt, lo más importante es que exista un nodo del tipo *move_group*, que nos proporcionará la habilidad de manipular el robot por medio de RViz o de otros nodos [52]. En la Figura 3.2 se incluye un esquema muy sencillo del funcionamiento para un robot.

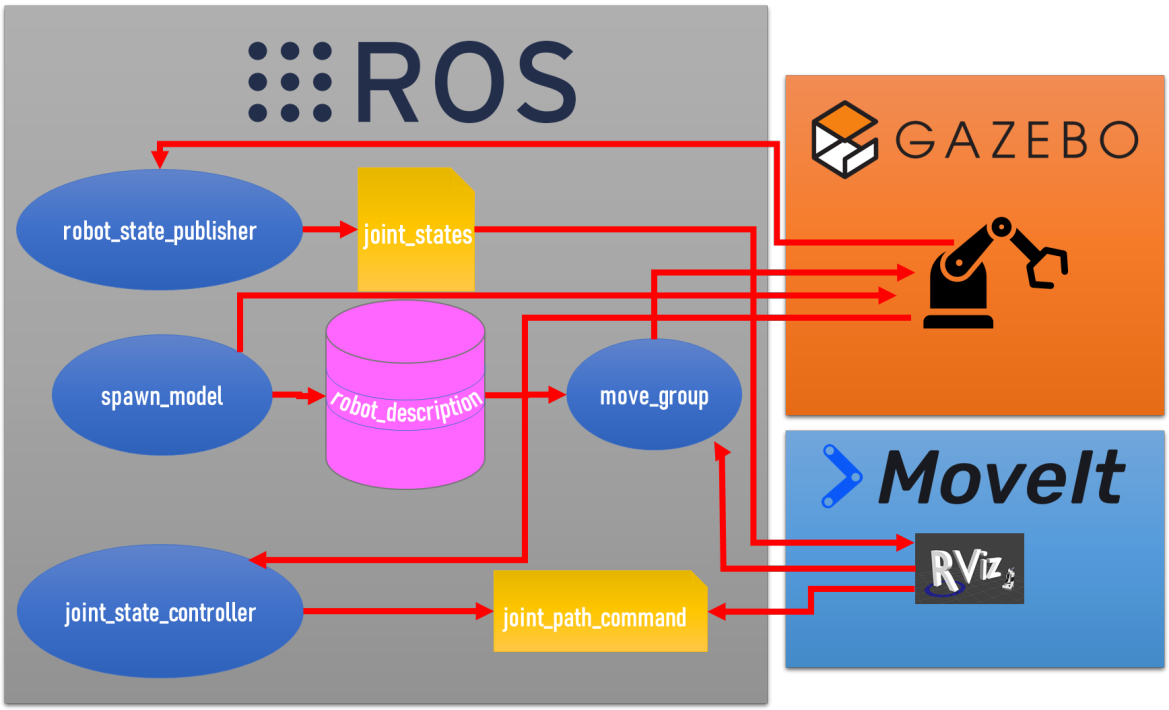


Figura 3.2: Esquema básico del funcionamiento coordinado de ROS, Gazebo y MoveIt para controlar el robot. Se observan los nodos "*robot_state_publisher*" "*spawn_model*", "*joint_state_controller*" y "*move_group*". Se incluyen también los *topics* "*joint_states*" y "*joint_path_command*". Aparece el servidor de parámetros, haciendo énfasis en *robot_description* y por el otro lado, Gazebo y MoveIt junto a RViz. Logo ROS: [21] Logo Gazebo: [53] Logo MoveIt: [54] Logo RViz: [55]

3.2. Paquete *multiple_abb_irb120*

Para realizar todas las modificaciones necesarias para el correcto funcionamiento del entorno multi-robot, se ha creado un paquete llamado *multiple_abb_irb120*. Este paquete contiene únicamente los archivos que han sufrido algún cambio respecto a los originales, de forma que se utilizan estos últimos en la medida de lo posible. Para ver la versión final de los archivos que se incluyen en este paquete, véase el Apéndice A.

3.2.1. Puesta en marcha de dos robots en Gazebo

Si intentamos lanzar los nodos para la simulación de un único robot dos veces, nos encontraremos con una serie de conflictos de nombres, por lo que es necesario encontrar una solución para que o bien los nombres se encuentren separados en diferentes espacios de nombres, o bien que todos posean nombres únicos.

En primer lugar se realizaron pruebas que duplicaban el código para cada robot que se deseaba tener en la simulación. Esta tarea no resultó ser trivial, puesto que surgió una gran cantidad de problemas, que se detallan en la Subsección C.1.1. Además, una de las desventajas de esta solución sería que duplica gran parte del código, y si en algún

momento se quisieran añadir más robots, requeriría hacerlo manualmente.

Por ello, se creó una organización de los archivos que permitiera una creación recursiva de los robots en Gazebo. Así se diseñó el fichero *recursive_spawn.launch* (véase Sección D.4), que invoca al fichero *spawn_irb120.launch* (véase Sección D.5) con los parámetros del nombre del robot y su posición en el mundo y también se invoca a sí mismo, hasta que el parámetro del número de robots que se debieran invocar sea 0 (véase Apéndice A para la organización final de los archivos desarrollados). Por su lado, el fichero *spawn_irb_120.launch* (véase Sección D.5) es el resultado de varios intentos de introducir todos los nodos, parámetros y servicios necesarios para un robot en un *namespace* propio, para que así aparecieran en Gazebo (por ejemplo, haciendo uso del parámetro "*ns*" al crear un nodo y cambiando directamente el nombre de ciertos parámetros, como el *robot_description*). Se puede observar el resultado en la Figura 3.3.

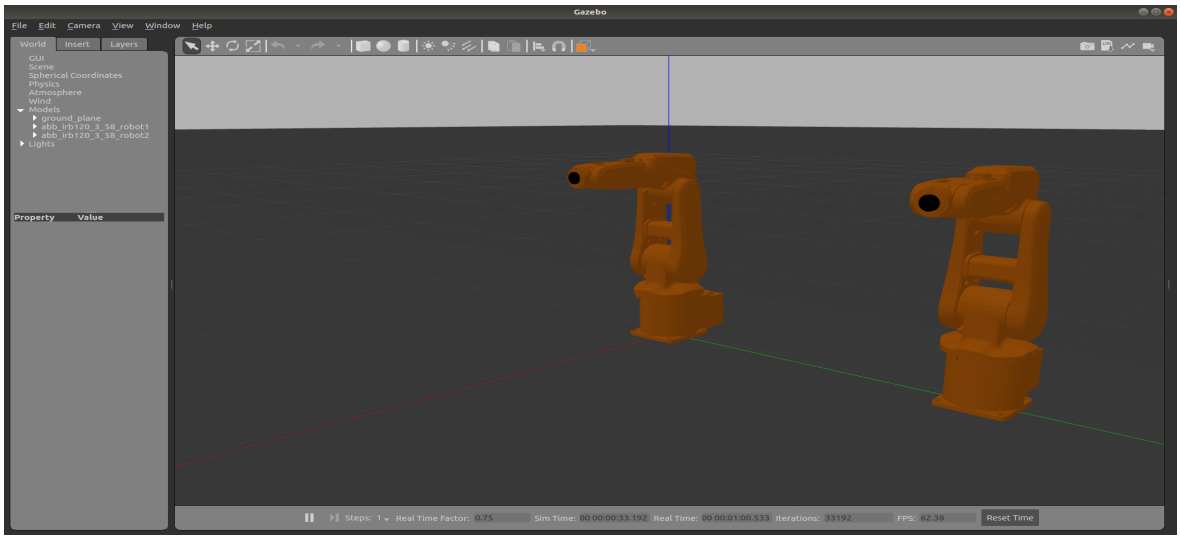


Figura 3.3: Captura del entorno de simulación Gazebo con dos robots ABB IRB120 después de la ejecución de los ficheros *launch* creados.

De esta manera, la versión final posee un fichero *setup_gazebo.launch* (véase Sección D.7) que se encarga de cargar el mundo de la simulación y que posee un parámetro que indica el número de robots a colocar en el mundo. Los robots aparecen en la posición $p = (0, i - 1, 0)$, con $i \in \mathbb{N}$ e $i > 0$ el número del robot, de forma que el primer robot aparece en el origen del sistema de coordenadas y el resto se colocan separados por un metro a lo largo del eje Y. La Figura 3.3 ilustra este comportamiento para dos robots.

Finalmente, se desarrolló un fichero de ***bash*** con un parámetro que se introduce por la línea de comandos, que se encarga de crear un fichero *launch* temporal que al lanzarlo inicialice Gazebo con el número de robots indicado por el parámetro y deje

todos los nodos en los espacios de nombres necesarios para que MoveIt y RViz puedan ejecutarse con normalidad (véase Sección D.6). Los robots se inicializarán con el nombre de $robot_i$, siendo i un número natural desde 1 hasta el número total de robots. Así, en una simulación con 2 robots, sus nombres serán *robot1* y *robot2*. La decisión de por qué se desarrolló un archivo de tipo *bash* se explican en la Subsección C.1.1.

Estos archivos sufrirán cambios también para poder lanzar MoveIt con ambos robots. Se recuerda al lector, que puede consultar la estructura de los ficheros en el Apéndice A.

3.2.2. Puesta en marcha de MoveIt para ambos robots

De la misma forma que en el paso anterior, todos los nodos, servicios y parámetros utilizados por MoveIt deben introducirse en un espacio de nombres o *namespace* para que Gazebo y MoveIt se comuniquen entre sí.

Se hizo uso de los comandos `rosparam list`, `rostopic list` y `rosnode list` para observar las discordancias entre Gazebo y MoveIt. Así, tras varios intentos (véase Subsección C.1.2) y haciendo uso de *namespaces* o modificando el nombre de algunos nodos, parámetros o *topics*, se hicieron coincidir los nombres entre los dos sistemas para conseguir una comunicación correcta entre ellos y también con RViz.

Así, se dispone de un fichero *moveit_planning_execution_gazebo.launch* (véase Sección D.8) que se encarga de lanzar todos los nodos y parámetros necesarios para la puesta en marcha de MoveIt para el robot indicado por su parámetro *robot_name*. Además inicializa RViz con dicho robot incluido para poder manipularlo manualmente. De este modo deberemos ejecutar este fichero tantas veces como robots tengamos en la simulación, indicándole manualmente el nombre del robot.

3.2.3. Conexión a los robots desde C++

Para poder mandar comandos a los robots desde C++ se utilizó la interfaz *MoveGroupInterface* de MoveIt. Para su puesta en marcha, se siguió uno de sus tutoriales que explica en detalle las funcionalidades de dicha interfaz (véase [56]).

En primer lugar, es necesario instanciar la clase, indicándole el nombre del grupo que forma el robot que queremos controlar [57]. Éste está definido en el fichero *SRDF* utilizado por el paquete de la configuración de MoveIt de ABB [58], y es referido en múltiples ocasiones por otros ficheros. Para que dichos grupos tuvieran nombres únicos, se realizaron cambios en todos ellos, haciendo uso de parámetros para transmitir el nombre del robot de fichero a fichero. Además, fue preciso reescribir el fichero *srdf* en un fichero de tipo *xacro* para poder añadirle el parámetro necesario (véase Sección D.9).

De esta forma, la interfaz en C++ puede distinguir qué robot se desea controlar. Todo esto se realiza en una clase creada para este propósito, denominada *RobotInterface*.

De esta manera, se desarrolló un programa llamado *robots_moving_demo*, que se detalla en el Capítulo 7. El sistema de comunicación global y de simulación se encuentra detallado en el esquema de la Figura 3.4.

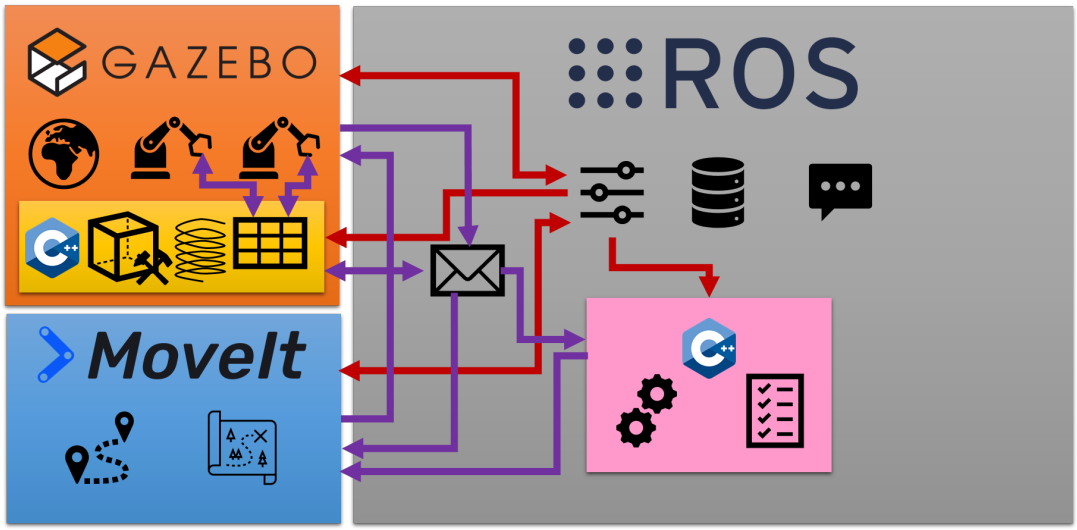


Figura 3.4: Esquema del funcionamiento de este trabajo.
Muestra las conexiones por medio de ROS y sus topics (representado con un sobre) entre el nodo de ROS, Gazebo y MoveIt. Aparece también un *plugin* de Gazebo (véase Capítulo 5). Gazebo se encarga de la simulación del mundo, objeto deformable y los robots. El nodo se encarga de programar las tareas a realizar. MoveIt se encarga de planificar rutas. ROS tiene el modo de comunicación por servicios (en el esquema, el bocadillo con los tres puntos), los topics (sobre) y parámetros (las barras deslizadoras). Los caminos por topics o por conexión directa están en morado, mientras que los caminos por parámetros están en rojo carmesí. Logo ROS: [21] Logo Gazebo: [53] Logo MoveIt: [54] Logo C++: [59] Vectores obtenidos de Microsoft Office Powerpoint.

Capítulo 4

Creación del objeto deformable

El modelo del objeto deformable será simulado por medio de una malla de puntos (véase Sección 5.1) en los que los puntos tendrán forma de esferas pequeñas, para lo cual, se pueden utilizar modelos independientes (un modelo por esfera) o bien un modelo que tenga tantos enlaces como esferas sean necesarias. Gazebo posee varios métodos para crear un nuevo modelo e incluirlo en la simulación. Aunque la solución más sencilla pudiera ser utilizar los ficheros *launch* para que el mundo de la simulación se inicialice con el modelo ya incluido, el formato no es muy flexible y no permite crear modelos que pudieran tener un número variable de enlaces, si se quiere probar el segundo tipo de modelos. A continuación se describen los métodos con los que se ha trabajado.

4.1. Creación por medio de servicios ROS

El primer enfoque que se probó fue el uso de los servicios en ROS que el simulador Gazebo expone. En particular, el servicio */gazebo/spawn_sdf_model* [47].

En la Sección 3.1 se mencionan los formatos URDF y SRDF. Ambos se utilizan en la descripción de modelos de robots [42] [43]. Sin embargo, existe además un formato denominado **SDF**, el cual es más genérico que los otros dos, puesto que se puede utilizar para describir tanto objetos simples, como robots complejos [60]. De esta manera, es posible definir una esfera de pequeñas dimensiones en este formato. Se creó un nodo inicial en C++ que hacía uso de los métodos de la API de ROS para llamar al servicio mencionado anteriormente, que lee el fichero y lo coloca en la simulación [61]. Así, se pudo generar una cuadrícula de un tamaño predefinido en el código.

La principal desventaja de este método es que las esferas aparecían de una forma excesivamente lenta (aproximadamente 26 segundos para un conjunto de 100 esferas). Por ello fue descartada rápidamente.

4.2. Creación por medio de un *plugin* de Gazebo

Gazebo es una herramienta muy potente, pues ofrece una serie de físicas y comportamientos preestablecidos, como gravedad, viento o detección de colisiones [62]. Pero además permite un control mayor de la simulación por medio de ***plugins*** [63]. Existen varios predefinidos [23] y también es posible la creación de *plugins* propios [64]. Existen varios tipos de ellos para cumplir distintos objetivos [64], y son de principal interés:

- ***WorldPlugin***: se lanza al inicializar el mundo de la simulación (véase la sección "*Run the Code*" en [65]) y puede cumplir varios objetivos, como control de físicas, generación de modelos, su modificación, entre otros [65].

- ***ModelPlugin***: se lanza al inicializar un modelo, y es capaz de modificarlo, cambiar sus propiedades físicas, aplicar fuerzas, aceleración o velocidad, entre otros [66].

- ***VisualPlugin***: se lanza al inicializar un objeto de tipo *visual*, el cual siempre debe pertenecer a otro objeto, como un modelo. Se utiliza para personalizar el aspecto de un modelo [64] [67].

A continuación se detalla la solución final que se tomó para la creación del modelo, pero también se incluye en la Sección C.2 otra de los enfoques que se le dio a esta tarea.

4.2.1. Creación del modelo con un *WorldPlugin*

Los *WorldPlugin* nos ofrecen la posibilidad de modificar el mundo de la simulación al inicializarse [65]. También existen muchas funciones en la clase *World* del namespace *gazebo::physics*, y en el namespace *gazebo::msgs* que sirven para completar esta tarea [68] [69]. Así, se creó un plugin para generar un nuevo modelo y añadir todas los enlaces de la tela. Esto también causó algunos de los problemas descritos en la Sección C.2 inicialmente, por lo que se usaron las funciones del namespace *physics* lo mínimo posible y se delegaron muchas de las tareas en el namespace *sdf*, que posee las funcionalidades para definir un modelo *SDF* programáticamente [70]. Así, podemos crear un modelo con múltiples enlaces con una forma esférica. Gracias a éste, se pudo completar el objetivo: el plugin lee un fichero denominado *grid.config* (véase Sección D.1) que posee los parámetros de la posición inicial de la tela, su tamaño, el número de esferas y otros parámetros necesarios para la simulación de la misma y los carga en el servidor de parámetros y finalmente la tela aparece inmediatamente al inicializar Gazebo.

Método	**Tiempo de ejecución (s)**
Servicios	25.9478
WorldPlugin	0.1988

Tabla 4.1: Comparación entre el tiempo de ejecución que los métodos de creación del modelo necesitan para generar una cuadrícula de 10x10 esferas

Cabe destacar que se debe crear un enlace inicial (antes de cualquier esfera) para que sea el enlace canónico, que es el que define el sistema de referencia local del resto de enlaces [47]. Se ha colocado en la posición (0, 0, 0) para que coincida con la referencia global.

Finalmente, se les ha añadido colores distintos a las esferas del objeto para que sean más distinguibles entre sí. La versión final se puede observar en la Figura 4.1.

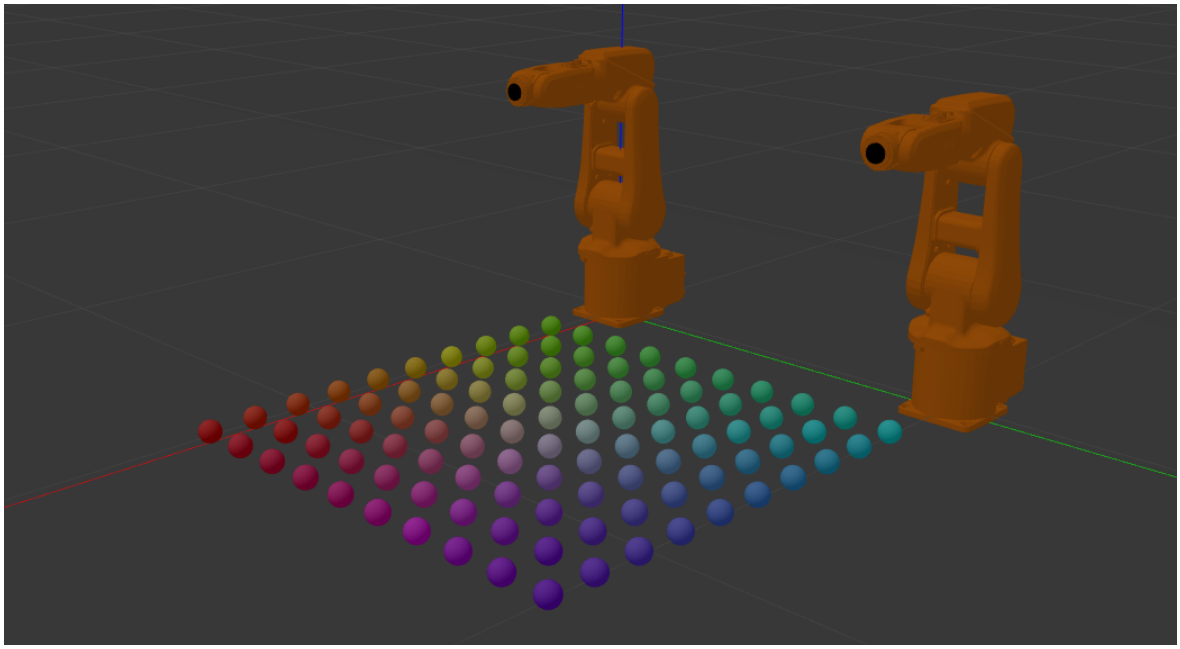


Figura 4.1: Versión final del objeto deformable, junto a los dos robots. Se trata de una tela de 10x10 esferas, de 2x2 metros.

Capítulo 5

Simulación del objeto deformable

La simulación del objeto deformable y su creación son tareas que fueron desarrolladas en paralelo, por lo que todas las opciones exploradas en el capítulo anterior (Capítulo 4) se han probado para la simulación también. Para conseguir un comportamiento realista de un objeto deformable (se implementará una tela elástica), se pueden utilizar gran cantidad de modelos distintos, pero en nuestro caso, se va a utilizar el modelo *Mass-Spring-Damping* o de "masa-muelle-amortiguador" y se han analizado dos métodos distintos para implementarlo. Todo ello es descrito a continuación.

5.1. Modelo *Mass-Spring-Damping*

El modelo *Mass-Spring-Damping* (Masa-Muelle-Amortiguador) es un modelo para representar las fuerzas a las que se encuentra sometido un objeto deformable [71]. El principio básico para representar el objeto es la presencia de unas masas puntuales a lo largo de todo el objeto que se encuentran interconectadas por muelles que siguen la ley de Hooke (aunque existen modelos que utilizan muelles de otros tipos para conseguir simulaciones más realistas de otro tipo, como el tejido humano [71]) y con un amortiguador [71] [72]. De esta manera, tenemos tres fuerzas distintas a las que puede estar sometida cada una de las masas puntuales:

- Causadas por el muelle: Se oponen a los movimientos de compresión / descompresión causados por otras fuerzas [73].

- De amortiguamiento: Sirven para representar la fricción a la que estaría sometido el objeto deformable. Depende de la diferencia de las velocidades entre masas [74].

- Externas: Producidas por un agente externo que manipula el objeto o el viento, u otras fuerzas como la gravedad [74].

Si juntamos los tres términos, obtenemos la siguiente ecuación:

$$\overrightarrow{F_a} = \overrightarrow{F_{ext_a}} + \sum_{b \in V_a} (\overrightarrow{f_{b \to a}} + \overrightarrow{d_{b \to a}}) \tag{5.1}$$

Donde $\overrightarrow{F_{ext_a}}$ es la suma de todas las fuerzas externas aplicadas a la masa a, V_a es el conjunto de todos las masas vecinas a la masa a, $\overrightarrow{f_{b \to a}}$ es la fuerza producida por la deformación del muelle situado entre la masa a y la masa b, y $\overrightarrow{d_{b \to a}}$ es la fuerza producida por el amortiguamiento entre las masas a y b. Así:

$$\overrightarrow{f_{b \to a}} = k_{ab} \cdot (\|\overrightarrow{x_b} - \overrightarrow{x_a}\| - \|\overrightarrow{x_{b0}} - \overrightarrow{x_{a0}}\|) \cdot \frac{\overrightarrow{x_b} - \overrightarrow{x_a}}{\|\overrightarrow{x_b} - \overrightarrow{x_a}\|} \tag{5.2}$$

Donde k_{ab} es la constante de elasticidad del muelle entre las masas a y b, $\overrightarrow{x_i}$ y $\overrightarrow{x_{i0}}$ son las posiciones actual e inicial de una masa i. Esto significa que se producen fuerzas que dependen de las distancias inicial y final entre las masas, y se alejarán en caso de una compresión y acercarán en caso de elongación [75].

Además:

$$\overrightarrow{d_{b \to a}} = D_{ab} \cdot (\overrightarrow{v_b} - \overrightarrow{v_a}) \tag{5.3}$$

Donde D_{ab} es la constante de amortiguamiento entre las masas a y b, y $\overrightarrow{v_i}$ es la velocidad actual de una masa i. Así, se produce una fuerza que suaviza la deformación, al oponerse a los cambios de velocidad entre masas, pero no evita el movimiento global del objeto deformable [73] [74].

Este modelo resulta interesante, pues es de fácil implementación debido a su simpleza, pero causa un comportamiento lo suficientemente realista para experimentar con un objeto deformable. Su comportamiento se ilustra en la Figura 5.1.

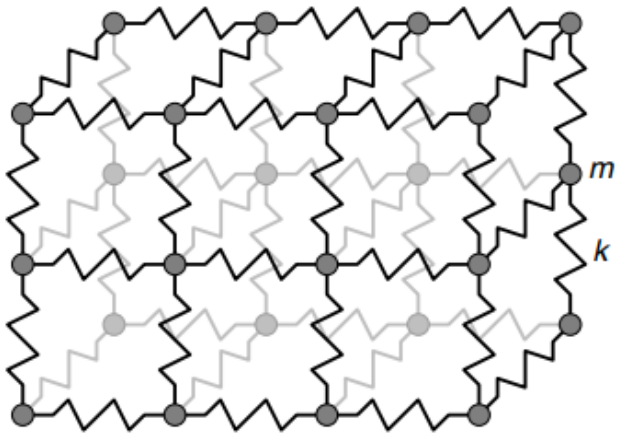


Figura 5.1: Representación de un objeto tridimensional deformable con el modelo *Mass-Spring-Damping*. Se muestran masas puntuales conectadas por un enlace muelle-amortiguador a 6 vecinos como máximo: dos en vertical, dos en horizontal y dos en profundidad. Origen: [71].

Mientras que la Figura 5.1 ilustra un comportamiento de un objeto tridimensional con 6 vecinos: Arriba, abajo, a la izquierda, a la derecha, adelante y atrás. En nuestro caso, implementaremos un objeto bidimensional que tendrá 8 vecinos: Arriba, Abajo, a la izquierda, a la derecha, arriba a la izquierda, abajo a la izquierda, abajo a la derecha y arriba a la derecha, como muestra la Figura 5.2

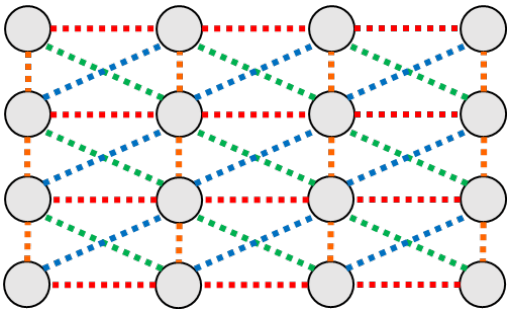


Figura 5.2: Representación de un objeto bidimensional deformable con el modelo *Mass-Spring-Damping*. Se muestran masas puntuales conectadas por un enlace muelle-amortiguador a 8 vecinos como máximo: dos en vertical, dos en horizontal y cuatro en diagonal.

5.2. Estudio de implementación en Matlab

La implementación del modelo ha estado muy influenciada por dos proyectos en el entorno Matlab, desarrollados para la simulación de objetos deformables de tipo tela en una simulación en 2D y 3D respectivamente [76] [77]. La Figura 5.3 demuestra el comportamiento de una de estas implementaciones ([77]).

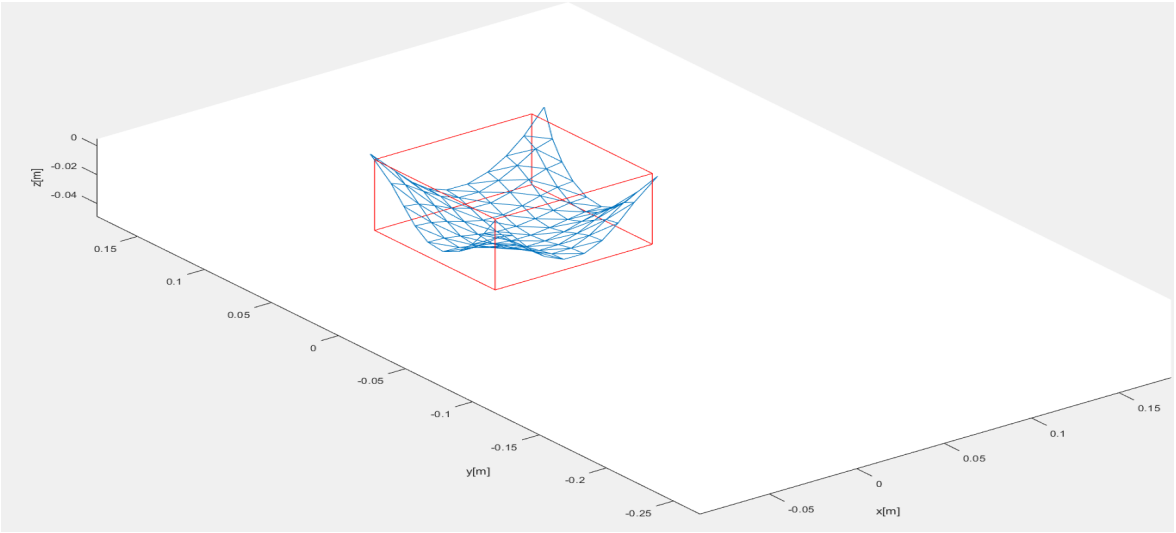


Figura 5.3: Implementación en Matlab del método *Mass-Spring-Damping* para un objeto deformable de tipo tela, con las cuatro esquinas fijadas y con gravedad activada.

5.3. Simulación por medio de servicios ROS

Como se ha descrito en la Sección 4.1, Gazebo expone servicios muy útiles para realizar modificaciones desde el exterior de la simulación por medio de ROS. En este caso, es destacable el servicio */gazebo/set_model_state*, que permite cambiar el estado en el que se encuentra un modelo de la simulación [47]. Así podemos enviar peticiones que cambien la posición en la que las esferas se encuentren. Nuevamente, la velocidad de ejecución fue muy pobre y no demostraba correctamente el comportamiento que una tela debiera tener.

5.4. Simulación por *plugins* de Gazebo

Si bien el *ModelPlugin* resultó una prueba fallida para generar el modelo (véase Sección C.2), sigue siendo una opción muy adecuada para indicar a la simulación el estado de las esferas. De esta manera, el *WorldPlugin* descrito en la Sección 4.2 le indica al modelo (en el formato *SDF*) que debe ejecutar un *plugin* al inicializarse. Por esto, el *ModelPlugin* puede acceder fácilmente a los enlaces y modificar sus propiedades con las funciones de la clase *Model* del namespace *gazebo::physics*.

El primer paso es obtener punteros a todos los enlaces. Para ello, se invoca a la función *GetLink* [78] con el nombre del enlace, que se les ha dado mediante el *WorldPlugin* de la Subsección 4.2.1. Estos enlaces se almacenan en una clase *Grid* que posee una matriz de objetos de tipo *GridVertex*, los cuales albergan el puntero al objeto de tipo *gazebo::physics::Link* (un enlace [79]) y ofrecen métodos para enviar a la simulación el resultado calculado por el modelo de objeto deformable.

Por otro lado, la clase *MassSpringDamping* es la encargada de implementar el modelo descrito en la Sección 5.1. Los parámetros de rigidez k_{ij}, masa m y amortiguamiento D_{ij} (descritos en la Ecuación 5.1) se añaden al fichero *grid.config* (véase Sección D.1), así como un parámetro adicional que indica si se desea que el objeto se vea afectado por la gravedad o no. Para poder acceder a los datos de posición, velocidad y fuerza, así como modificarlos, esta clase recibe un puntero al objeto *Grid*. Surge aquí una decisión que se debe tomar: ¿debería el modelo modificar directamente la posición de cada esfera? ¿o debería indicarle a Gazebo la fuerza a la que está sometida?

En este caso se ha seguido un enfoque hacia la segunda opción, calcular las fuerzas y dejar que Gazebo simule las físicas según dichas fuerzas. La principal razón es que el cálculo de la velocidad y posición precisan de la medición del tiempo transcurrido entre un instante de tiempo y el siguiente manualmente, lo cual puede introducir

ciertas imprecisiones. De esta manera, cada *GridVertex* posee un campo *force_cache* que almacena la fuerza calculada por el objeto *MassSpringDamping*, y posteriormente, se actualizan todos las esferas a la vez, enviando este valor a la simulación en Gazebo.

Capítulo 6

Interacción Robot - Objeto deformable

Para conseguir una simulación realista, es necesario que los robots conozcan la posición del objeto deformable para poder aproximarse y "agarrarlo". Pero además, en ese caso, el objeto deformable también deberá saber la posición del robot, para seguir sus movimientos si dicho robot ha agarrado el objeto. ROS nos ofrece una buena forma de lidiar con la comunicación entre los nodos que conforman el control de los robots y la simulación del objeto deformable.

6.1. Obtención de la posición del objeto deformable

Como se ha mencionado en capítulos anteriores, Gazebo ofrece una serie de *topics* y servicios que cumplen diversos objetivos. Para que el robot sea capaz de conocer la posición a la que deberá acercarse para agarrar el objeto deformable, existen:

- *Topics* ([47] - *Gazebo Published Topics*):
  - */gazebo/model_states*: Publica los estados de todos los modelos de la simulación.
  - */gazebo/link_states*: Publica los estados de todos los enlaces de la simulación.
- Servicios ([47] - *Services: State and property setters*):
  - */gazebo/get_model_state*: Devuelve el estado de un modelo.
  - */gazebo/get_link_state*: Devuelve el estado de un enlace.
  - */gazebo/model_states*: Devuelve el estado de todos los modelos de la simulación.

- */gazebo/link_states*: Devuelve el estado de todos los enlaces de la simulación.

Debido a los resultados obtenidos con el uso de servicios en el desarrollo de capítulos anteriores (véanse la Sección 4.1 y la Sección 5.3), no se han realizado pruebas con el uso de los servicios, y se ha hecho uso exclusivo de los *topics* para este propósito. Así, los dos *topics* ofrecidos por Gazebo nos ofrecen una manera rápida de obtener la posición de los objetos en la simulación. Si inspeccionamos los mensajes que se intercambian por el primero de ellos, podemos observar que la información es demasiado genérica y no nos aporta ninguna manera de conocer la posición de las esferas, pues solo incluye la pose del conjunto del modelo [80]. Por ello, se ha utilizado el *topic* "*/gazebo/link_states*".

El nodo que controla el movimiento que deberían hacer los robots se subscribe al *topic* y obtiene las poses de todos los enlaces cuyo nombre comience por "*grid::link_*", pues es la combinación del nombre del modelo y del enlace ([47] - *Services: State and property setters*). Al llegar un mensaje, las posiciones se guardan en una matriz dentro de una clase llamada *GridState* de forma que los robots puedan saber en todo momento cuál fue la última posición que se ha recibido de cada esfera. La posición en la matriz depende del número incluido en el nombre del enlace.

Finalmente, se utiliza la posición de una de las esferas como posición objetivo en las funciones de la interfaz de MoveIt. Esto sin embargo produce una situación indeseable, que es que el robot golpee en su trayectoria a una de las esferas y mueva el conjunto antes de agarrarlo, además de que MoveIt moverá el robot hasta que el centro del extremo del robot esté exactamente en el centro de la esfera (véase Figura 6.1), provocando el mismo efecto. La solución a estos problemas se detalla en la Sección 6.2.

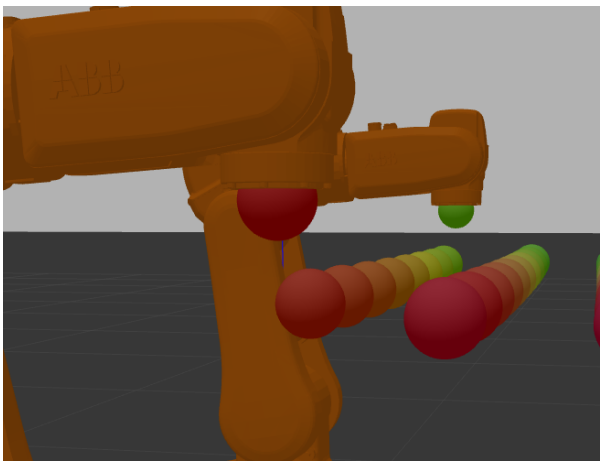


Figura 6.1: Posicionamiento de las esferas dentro del extremo del robot. Los robots han "agarrado" las esferas roja y verde y la mitad de las mismas se encuentra en el interior del robot.

6.2. Planificación de trayectorias con evitación de colisiones

Para que MoveIt realice una trayectoria hasta una esfera sin golpear a las demás, se plantean dos posibilidades. Una primera que sería más sencilla se trataría de realizar una ruta manual que pasase por puntos que de antemano se conozca que sea segura para el movimiento del robot.

Otra solución más genérica y que ha sido la que se ha decidido implementar, es hacer saber a MoveIt, gracias a las funciones de la interfaz *PlanningSceneInterface*, [81]) que existen obstáculos en el camino que debe realizar. Para esto, se deben crear mensajes del tipo *CollisionObject*, los cuales almacenan la forma del objeto (ya sean modelos, planos o primitivas como esferas o cubos) o su posición en el mundo, entre otros. También posee *flags* que indican la operación que se pretende hacer con dicho objeto: añadirlo a la escena, eliminarlo, moverlo o encadenarlo a otro objeto de la escena [56] [82].

Los archivos descritos en el Capítulo 5 crean, gracias al prefijo *tf_prefix* una "escena" para cada robot, de forma que se puede instanciar una *PlanningSceneInterface* para cada uno de ellos, y así indicarle situaciones distintas a cada robot, como se puede observar en la Figura 6.2. Así, se ha modificado la clase *GridState* para que su constructor se encargue de añadir las esferas a las escenas indicadas en la posición relativa al robot. Posteriormente, se crea un proceso asíncrono que mueve periódicamente dichos obstáculos a la posición determinada por la matriz de posiciones, que se actualiza según el procedimiento descrito en la Sección 6.1. En caso de que un robot se encuentre agarrando el objeto deformable, estas esferas son eliminadas de la escena para dejarle al robot total libertad de movimiento para poder manipular el objeto de la manera que necesite.

Por otro lado, para que el extremo del robot no choque con la esfera a agarrar, se ha añadido un enlace (sin ningún tipo de geometría) al robot que simula la presencia de una herramienta, a una distancia determinada del extremo del robot. Además, se añade una articulación que une este enlace con el denominado "*tool0*", utilizado por los paquetes de ROS para estandarizar el final de sus robots con ROS Industrial [83]. Así, MoveIt planeará la ruta para quedarse a la distancia a la que se encuentra dicho enlace. Esto no podrá ser utilizado por Gazebo ni los plugins, al ser un enlace sin componentes inerciales [84] y una articulación fija (pues las articulaciones fijas se conectan para reducir tiempos de compilación [85]).

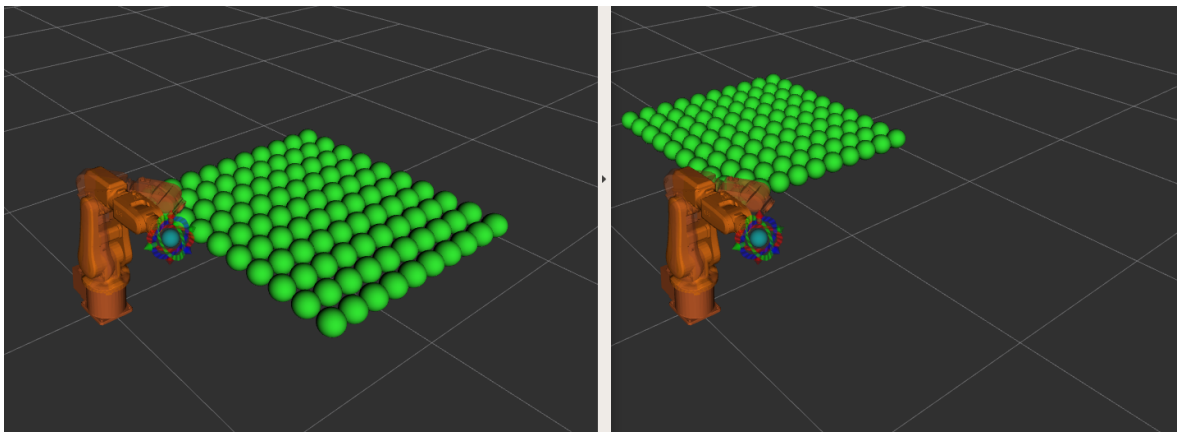


Figura 6.2: Presencia de obstáculos en MoveIt (en la interfaz RViz) para planeado de rutas seguro. A la izquierda se encuentra la ventana del robot1, y a la derecha el robot2. Las esferas están en las posiciones relativas a los robots y tienen un tamaño superior al de las esferas en Gazebo para evitar colisiones.

6.3. Petición de agarre

El primer paso para que una esfera sea "agarrada" es que la simulación del objeto sepa que se ha producido el agarre y evite calcular las fuerzas para dicha esfera. Entonces, debemos establecer un método para comunicar desde el nodo de los robots al *plugin* que simula el objeto. En este caso, se ha creado un topic bajo el nombre "*/grid/grab_petitions*" con un tipo de mensaje personalizado "*GrabPetition*", que está compuesto por dos enteros i y j que representan la posición en la cuadrícula de la esfera que se quiere agarrar, un booleano *grab*, una cadena de caracteres *link_name* y otra *robot_name* (véase Sección D.3). En el momento en el que el *plugin* del modelo lee un mensaje en el topic anterior, la esfera indicada se marca como "agarrada" o "soltada" según el parámetro *grab*. Además, en caso de que la petición sea de agarre, el *plugin* obtendrá la pose del enlace con el nombre *link_name* del robot *robot_name*. En caso contrario, el comportamiento vendrá definido por el modelo *Mass-Spring-Damping* de nuevo. Al no hacer uso de una herramienta de manipulación (y no poder usar el procedimiento descrito en la Sección 6.2) para la simulación, el enlace que se utilizará será el último del robot, en este caso, "*link_6*" [86].

6.4. Obtención de la posición de los robots

Una vez una esfera es "agarrada" por uno de los robots, su comportamiento esperado sería que se moviera a la vez que el robot. Para poder conseguirlo, las esferas también deben conocer el estado de los robots. Para esto, se obtiene un puntero al mundo de la simulación desde el modelo asociado al *plugin*, y con este puntero, se obtienen los

punteros al modelo del robot y del enlace indicados por la petición (descrito en la Sección 6.3). De esta manera, sabemos la posición en la que se encuentra el extremo del robot (siempre y cuando este enlace exista) y se podrá colocar la esfera en una posición ligeramente separada del mismo por medio de la composición de la pose de la esfera y la del robot, La Figura 6.3 muestra el mismo caso de estudio que la Figura 6.1. Previamente, el extremo de los robots se colocaba en una posición directamente en contacto con la esfera, y una vez "agarrada", ésta se colocaba en el interior del extremo. Ahora, los robots se mueven hasta una posición cercana y las esferas mantienen esa distancia de forma satisfactoria.

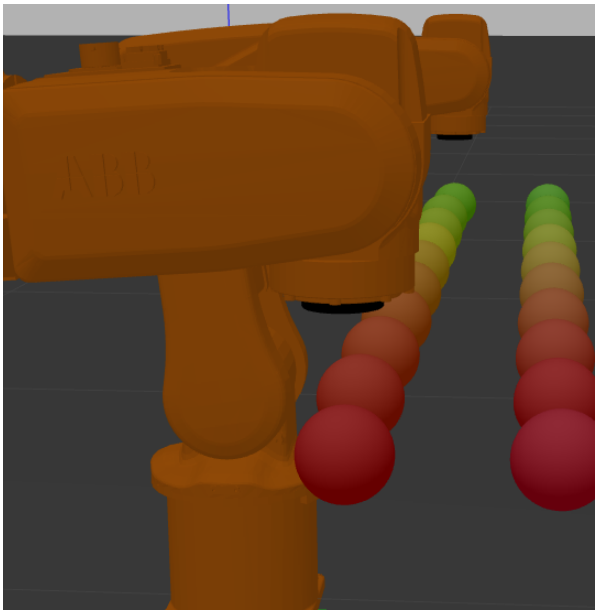


Figura 6.3: Posición final de la esfera tras haber sido "agarrada" por un robot. En la imagen, el robot más próximo ha cogido la esfera roja, y el más lejano ha cogido la verde.

Capítulo 7

Experimentos realizados y resultados

Para comprobar el correcto funcionamiento del sistema diseñado, se han desarrollado varios programas y se han utilizado varios métodos de prueba. En este capítulo se explicará en qué consisten estos experimentos y se analizarán los resultados obtenidos. Para ver cómo se deben ejecutar, véase el Apéndice B.

7.1. Creación de robots en la simulación

El primer hito clave de este Trabajo se basaba en la puesta en marcha de un entorno con varios robots (no necesariamente 2). Se han realizado experimentos para probar que los nombres de nodos, parámetros y *topics* no coincidían entre sí al añadir más de un robot, por lo que los ficheros *launch* deberán funcionar para cualquier número de robots indicado.

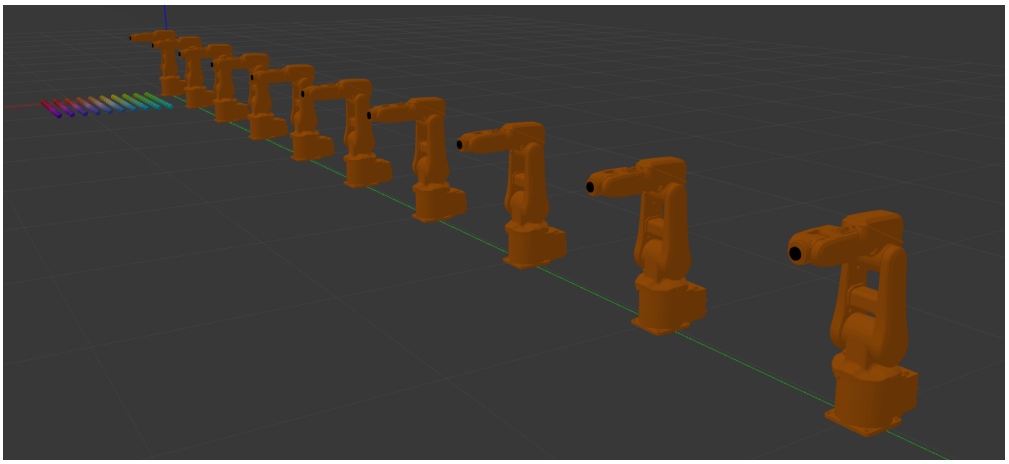


Figura 7.1: Simulación de objeto deformable con 10 robots.

Resultados

Se puede colocar un número arbitrario de robots en la simulación gracias a los ficheros *launch* y la terminal no deberá mostrar errores de conflictos de nombres (sólo mostrará errores y avisos procedentes de la forma de ABB para definir los robots, puesto que estos paquetes son para ROS Kinetic [17], por lo que pueden estar ciertamente desactualizados). En la Figura 7.1 se observa un ejemplo con 10 robots.

7.2. Movimiento de los robots en entorno multi-robot

Los robots creados en la sección anterior deben poder ser controlados individualmente (Capítulo 3). La prueba más básica que se puede realizar es, una vez abierta la simulación, utilizar la interfaz gráfica de RViz para indicar a uno de los robots una *pose* objetivo (compuesta de una posición y una orientación) y hacer click en los botones *Plan* y *Execute* y así comprobar el estado de la conexión entre MoveIt y Gazebo (véase Figura 7.2).

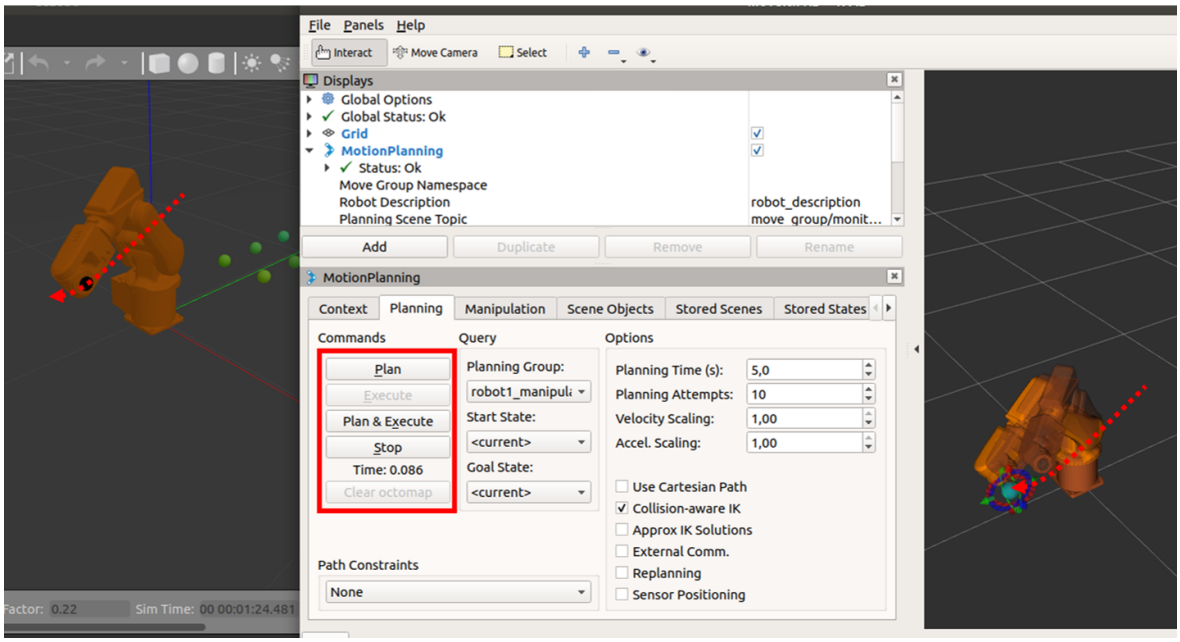


Figura 7.2: Resultado de una petición de movimiento manual a través de RViz. A la derecha se observa RViz con un robot que se ha controlado para pasar a otra posición. Después de pulsar los botones de *Plan* y *Execute*, el robot ha comenzado a moverse también en Gazebo (a la izquierda).

Por otro lado también podremos ejecutar el programa *robots_moving_demo*, para probar las conexiones con un nodo escrito en C++ gracias a la interfaz de MoveIt. Este toma control de dos robots y hace que vayan a la posición de las cuatro esquinas

de un cuadrado (dejando a MoveIt planear la ruta que considere óptima) y después dibujen el cuadrado forzando a los robots a hacer movimientos cartesianos en línea recta utilizando la función *computeCartesianPath* de MoveIt, aunque no sea óptima (es decir, que no sea la ruta más rápida), y que pase por todos los puntos que se le indique [56]).

Finalmente, también tenemos el programa *robots_waving_demo*, que es una versión un poco distinta del programa anterior, que mueve un número predeterminado de robots (indicado por un argumento en la terminal) en un movimiento senoidal.

Resultados

Al realizar la primera prueba, el botón *Plan* deberá provocar que el robot en RViz muestre la trayectoria que realizará, y *Execute* hará que esta se ejecute en Gazebo (véase Figura 7.2). Si se introduce una pose a la que el robot no puede llegar, RViz y la terminal donde se estuviera ejecutando mostrarán mensajes de error.

El programa *robots_moving_demo* deberá cumplir la especificación anterior y finalizar correctamente. El comportamiento está ilustrado por la Figura 7.3.

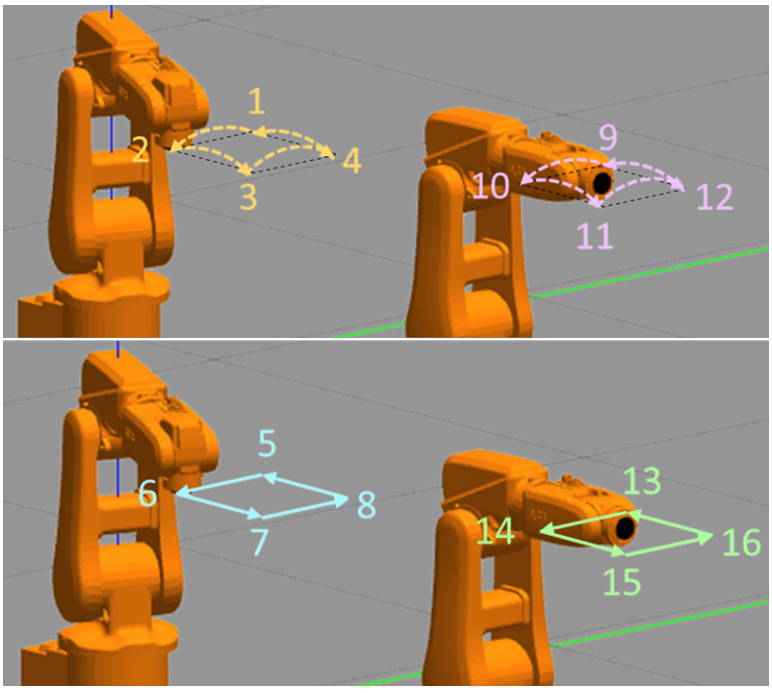


Figura 7.3: Trayectorias realizadas por el programa robots_moving_demo. La primera trayectoria, en color amarillo, es realizada por el robot1 y se mueve a las posiciones 1-4 con la punta del robot apuntando hacia el suelo por el camino óptimo. Después, el mismo robot realizará un cuadrado perfecto de las posiciones 5-8 marcadas en cian. De la misma forma, el robot2 replicará posteriormente los mismos movimientos que el robot1 en las trayectorias marcadas en rosa y verde. En la figura, el robot1 se encuentra realizando la trayectoria cian.

Por su lado, el programa *robots_waving_demo* hace que los robots traten de sincronizarse para formar una onda senoidal con sus extremos de manipulación. Las limitaciones de MoveIt impiden un control de la velocidad en movimientos cartesianos (es decir, movimientos en los que todos los puntos de paso están definidos), por lo que no siempre estarán sincronizados, pero hay distintos puntos de la trayectoria vertical en los que los robots esperarán a que los demás completen su movimiento. Se observan unas capturas de esta simulación en la Figura 7.4.

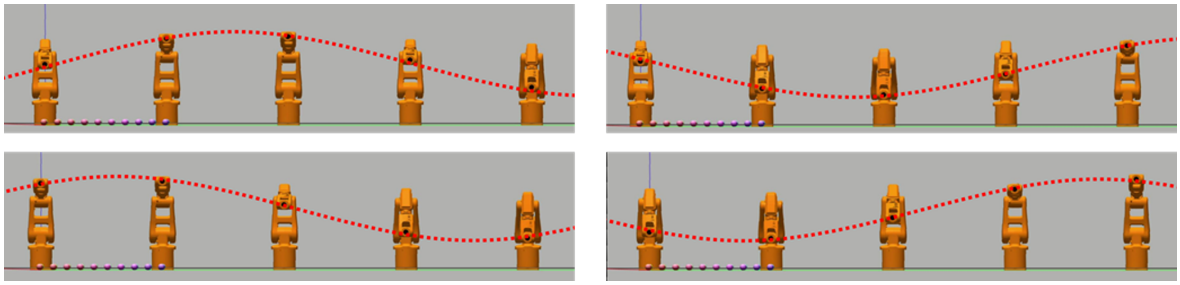


Figura 7.4: Trayectorias realizadas por el programa robots_waving_demo. Los robots se intentarán sincronizar para crear la forma de una onda senoidal.

7.3. Creación y movimiento del objeto deformable

Para probar la creación del objeto deformable, se puede utilizar el fichero *test_grid.launch*, que coloca en la simulación un objeto con las características indicadas por el fichero *test_grid.config*. El primero tiene los extremos de la tela "anclados" por lo que no se moverán y nos permitirá observar cómo interactúa el objeto en presencia de gravedad. El segundo tiene la gravedad desactivada por defecto (siempre podemos modificar el fichero *grid.config*) y colocará también los robots en el entorno.

En cualquiera de los dos podemos seleccionar una de las esferas y moverla a otra posición para comprobar como intenta volver a su posición inicial.

Resultados

Deberemos obtener un objeto deformable en la simulación de Gazebo dependiendo de los parámetros que se hubieran especificado.

En la Figura 7.5 y en la Figura 7.6, podemos observar una tela de 20x20 esferas que tiene sus cuatro extremos (las esferas roja, verde, cian y morada) "anclados" de forma que no se pueden mover. La gravedad ha provocado que el resto de esferas caigan hacia abajo, y en su caso, reboten hacia arriba si han descendido demasiado, tal y como lo haría una tela elástica real.

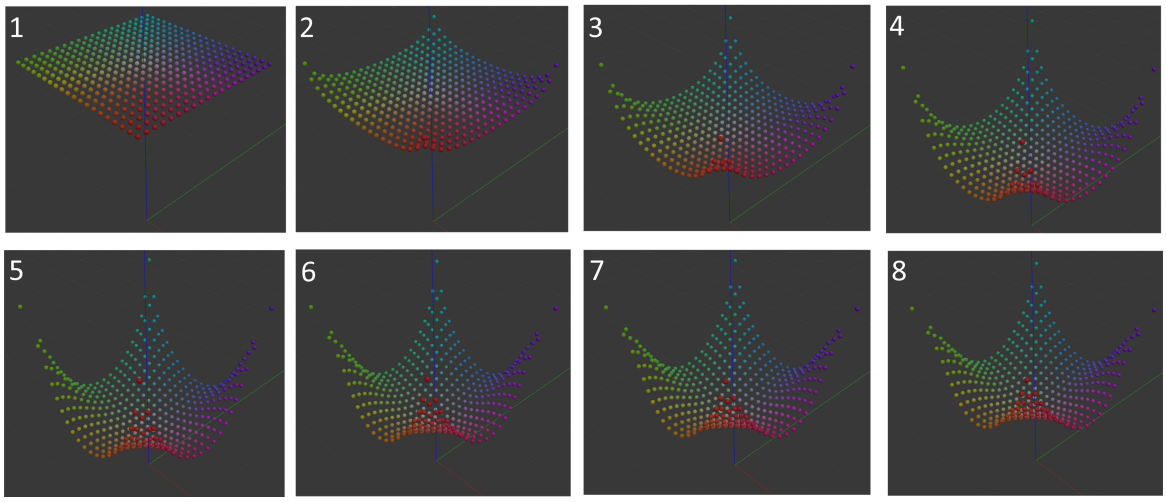


Figura 7.5: Distintos fotogramas de la simulación con *text_grid.launch*.
El objeto mide 2x2 metros y 20x20 esferas, el centro se sitúa inicialmente en (0,0,2.5). Las esferas tienen una masa (m) de 0.08kg, y la tela tiene una rigidez (k_{ij}) de 100N/m y un amortiguamiento (D_{ij}) de 10Ns/m. A partir del fotograma 5, la tela sube hacia arriba, pues se ha estirado mucho hacia abajo y está rebotando.

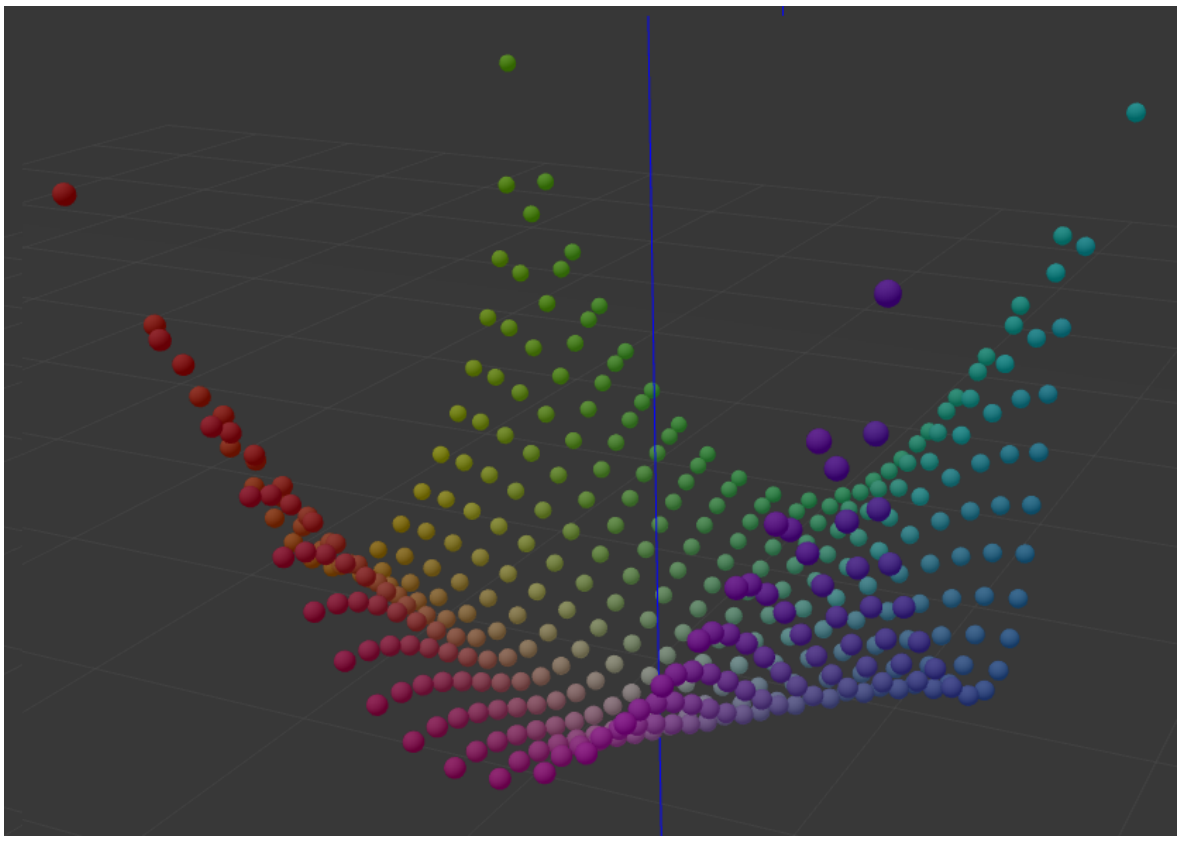


Figura 7.6: Resultado de lanzar el fichero *text_grid.launch*.
El objeto mide 2x2 metros y 20x20 esferas, el centro se sitúa inicialmente en (0,0,2.5). Las esferas tienen una masa (m) de 0.08kg, y la tela tiene una rigidez (k_{ij}) de 100N/m y un amortiguamiento (D_{ij}) de 10Ns/m.

Sin gravedad, al mover una de las esferas, observaremos que lentamente volverá a su posición inicial como se observa en la Figura 7.7. Sin embargo, si se realiza en el **eje normal** al plano formado por el objeto deformable, las esferas se quedan en una posición distinta a la inicial por las características del modelo. Esto ocurre porque el modelo *Mass Spring Damping* está basado en muelles rectos, por lo que los movimientos en direcciones tangentes al plano siempre tendrán muelles cuyas direcciones o suma de direcciones coincidan con estas. Por otro lado, la dirección normal nunca tendrá muelles que tiren naturalmente en esa dirección, por lo que el objeto se acercará en los tres ejes. La distancia llegará a ser igual a la inicial y quedará en reposo en un lugar distinto al inicial, como se observa en la Figura 7.8.

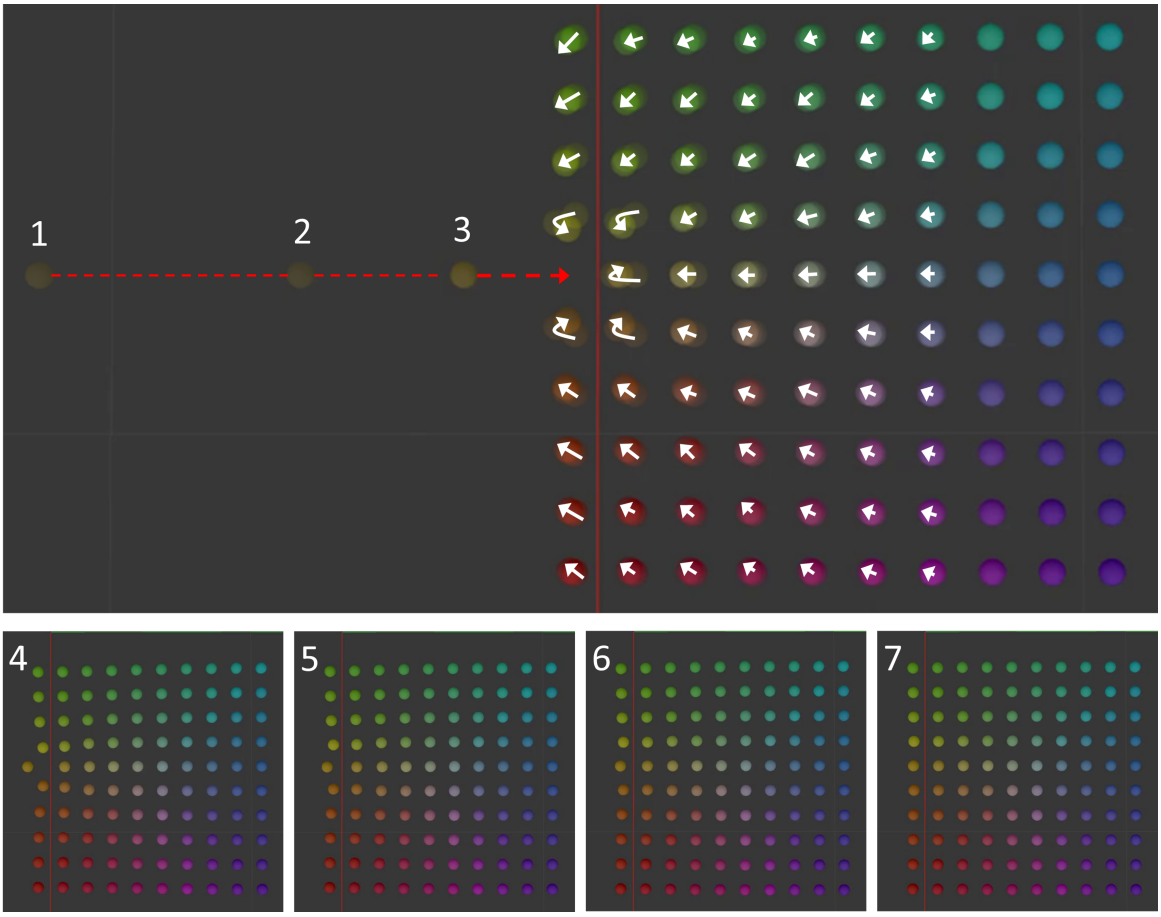


Figura 7.7: Evolución de la posición de una esfera al ser desplazada en el eje Y (tangente al plano de la cuadrícula) y su efecto en el resto de esferas. Esta composición de imágenes muestra las posiciones por las que pasa la esfera amarillenta al ser desplazada en el eje Y. En la parte superior, se muestran las direcciones en las que se ha visto desplazado el resto de esferas, desde la posición de reposo inicial. La posición final (7) es igual a la inicial y las demás esferas no han visto su posición final modificada.

Se han realizado pruebas de los programas de la Sección 5.2 para comprobar que los comportamientos coincidían.

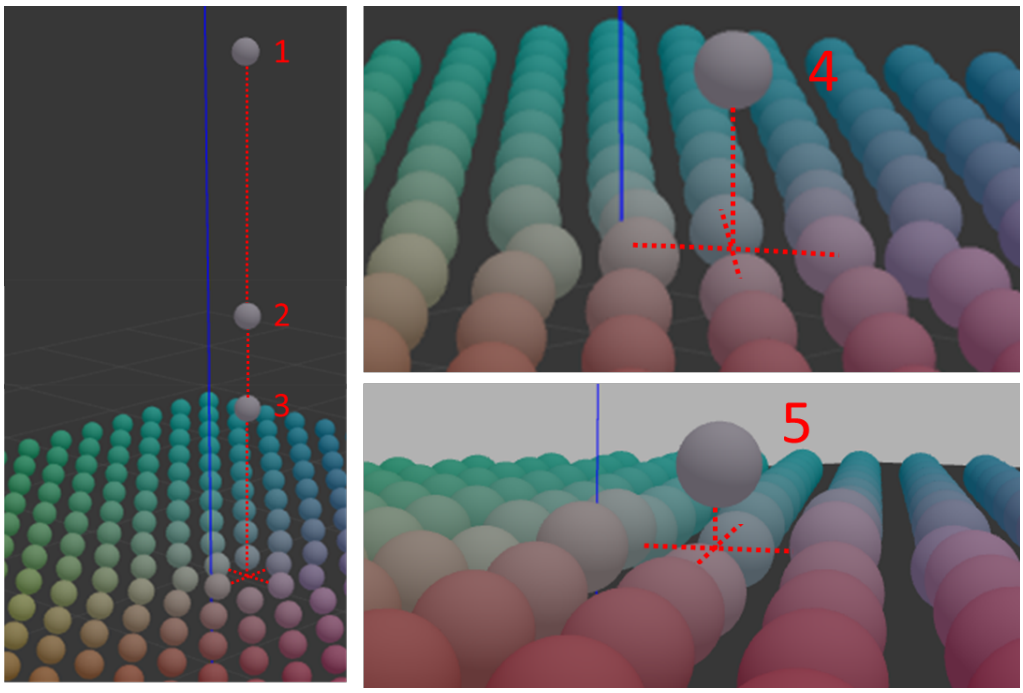


Figura 7.8: Evolución de la posición de una esfera al ser desplazada en el eje normal al plano de la cuadrícula. Esta composición de imágenes muestra las posiciones por las que pasa la esfera blanquecina al ser desplazada en el eje Z. La posición final (5) no se corresponde con la inicial y además el resto de esferas también han resultado desplazadas.

Cabe destacar que dependiendo de los parámetros, el sistema puede llegar a ser **inestable** al colisionar con algún robot o con el suelo si los parámetros de la simulación no están bien ajustados (por ejemplo, cuando las fuerzas de amortiguamiento son superiores a las de rigidez).

7.4. Manipulación del objeto por robots

Se dispone del programa de prueba *grid_demo* que, en presencia de una simulación con un objeto deformable y dos robots, hace que los robots se acerquen al objeto deformable, lo cojan y realicen una serie de movimientos síncronos para moverlo. También se ha desarrollado el programa *grid_wave_demo*, diseñado para usarse con gravedad, puesto que los robots cogerán el objeto deformable y lo subirán hasta una posición más elevada (adecuado cuando el objeto se encuentra en el suelo) y lo agitarán un poco de izquierda a derecha hasta soltarlo.

Resultados

Los robots se moverán lentamente hasta el objeto, lo cogerán y, en primer lugar, realizarán simultáneamente un movimiento en forma de cuadrado (ilustrado en la

Figura 7.9). Después, tirarán del objeto en direcciones opuestas (véase Figura 7.10), y el robot2 lo soltará para observar la reacción del objeto, como se observa en la Figura 7.11. Finalmente la misma figura muestra cómo el *robot1* también soltará el objeto y el programa finalizará.

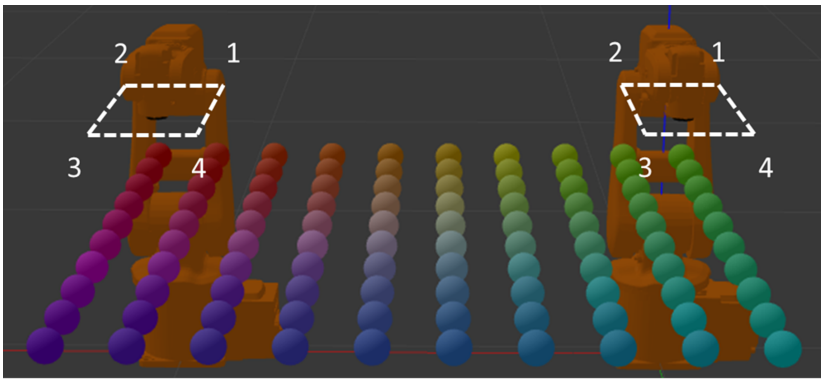


Figura 7.9: Fotogramas de una simulación *grid_demo* - Movimiento síncrono. Se observan los dos robots preparados para coger las esferas. Una vez las cojan, realizarán a la vez el cuadrado 1-4 y volverán al punto 1. Esta simulación se ha realizado sin gravedad.

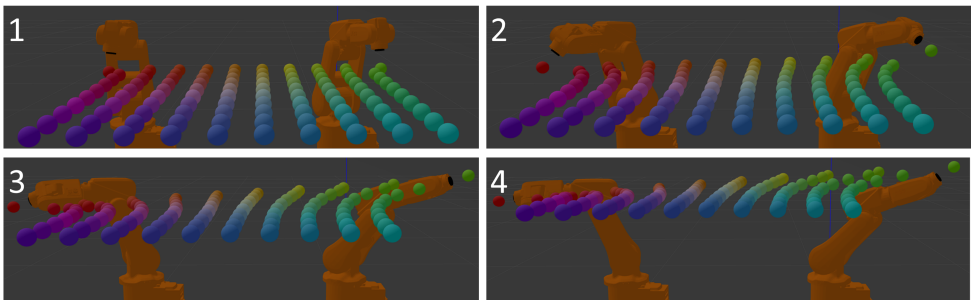


Figura 7.10: Fotogramas de una simulación *grid_demo* - Estiramiento. Se muestran cuatro fotogramas del comportamiento del objeto estirándose por la acción de dos robots.

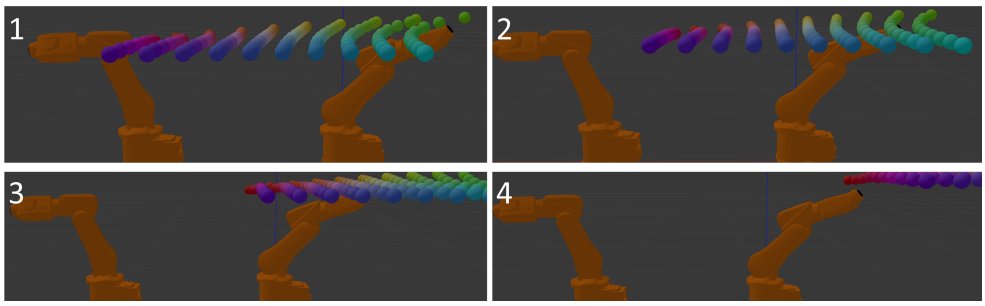


Figura 7.11: Fotogramas de una simulación *grid_demo* - Lanzamiento. Se observa una secuencia de fotogramas en las que el objeto deformable, sin gravedad es "lanzado" por los robots, al haberse estirado y después soltado por uno de sus extremos.

Se ha incluido un fichero adicional en el directorio *worlds* denominado *grid_table.world*, que genera un mundo idéntico al original, pero que además incluye una de las mesas predefinidas por Gazebo para poder realizar pruebas más cómodamente con gravedad, sin que los robots deban coger el objeto en el suelo (véase Apéndice B para su ejecución). Se puede observar un ejemplo de simulación con gravedad y con mesa en la Figura 7.12.

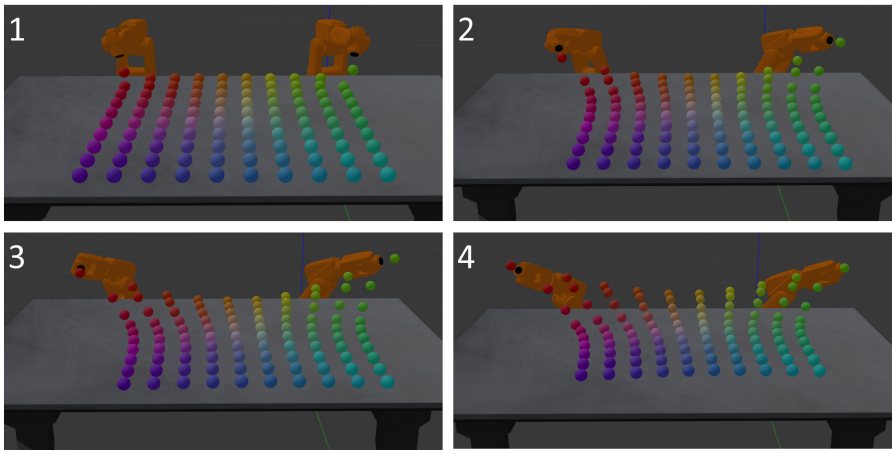


Figura 7.12: Fotogramas de una simulación *grid_demo* - Estiramiento con mesa. Se muestran cuatro fotogramas del comportamiento del objeto estirándose sobre una mesa por la acción de dos robots.

Por otro lado, el programa *grid_wave_demo* también manipula correctamente el objeto y lo levanta del suelo (siempre y cuando no haya una mesa y exista gravedad), sin embargo el movimiento de los robots resulta tan lento que no se llega a apreciar el movimiento de "agitado" del objeto. La Figura 7.13 consta de una serie de fotogramas del comportamiento anterior, que queda demostrado satisfactoriamente.

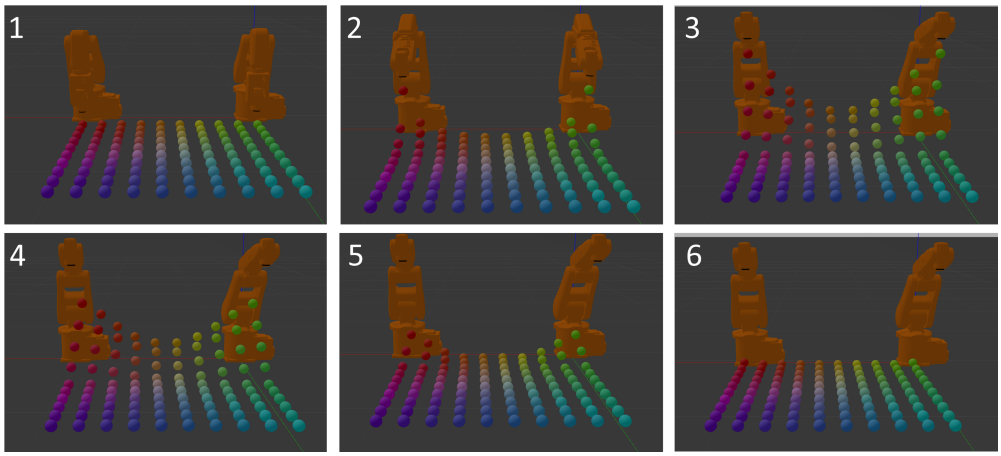


Figura 7.13: Fotogramas de la simulación *grid_wave_demo* - Levantamiento de tela grande. Se muestran seis fotogramas del comportamiento del objeto cuando dos robots la levantan por sus extremos.

7.5. Simulación Final

El programa *small_cloth_manipulation* (véase Sección D.11) está diseñado para trabajar con una tela elástica pequeña, cuyos parámetros están definidos en el archivo *small_grid.config* (véase Sección D.2). Los robots deberán tomar la tela y realizar manipulaciones similares a los de los apartados anteriores. Uno de los motivos que llevó a la utilización de un objeto de dimensiones menores es para observar el objeto una vez sea levantado por los robots y así comprobar su comportamiento como si se tratase de un objeto real.

Resultados

Los robots se acercarán lentamente hasta la tela y la agarrarán por el extremo que tengan más cerca. Entonces, la levantarán y mantendrán en el aire durante unos instantes (véase Figura 7.14). Después agitarán el objeto varias veces de adelante hacia atrás en el eje de coordenadas X (véase Figura 7.15). Al finalizar, tirarán de la tela en direcciones opuestas del eje Y (véase Figura 7.16), y harán una trayectoria hacia abajo y adelante en el eje X para conseguir dejar la tela estirada en el suelo (véase Figura 7.17). Finalmente, ambos robots volverán a la posición inicial.

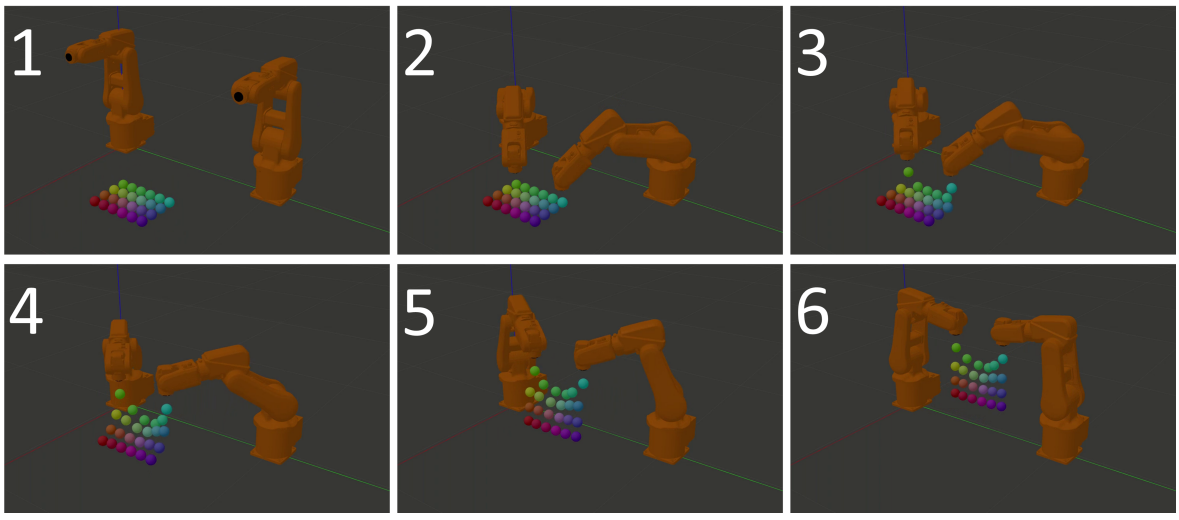


Figura 7.14: Fotogramas de la simulación final - Agarre de la tela. Se muestran seis fotogramas en los que los robots cogen una tela de 20x30 centímetros y 4x6 esferas.

Cabe destacar que en el momento en el que los robots agitan el objeto, los robots realizan trayectorias completamente rectilíneas, puesto que tienen alcance suficiente como para chocar entre sí, lo cual produciría una situación de inestabilidad para los robots, y que en el caso de robots reales podría causar una situación extremadamente peligrosa.

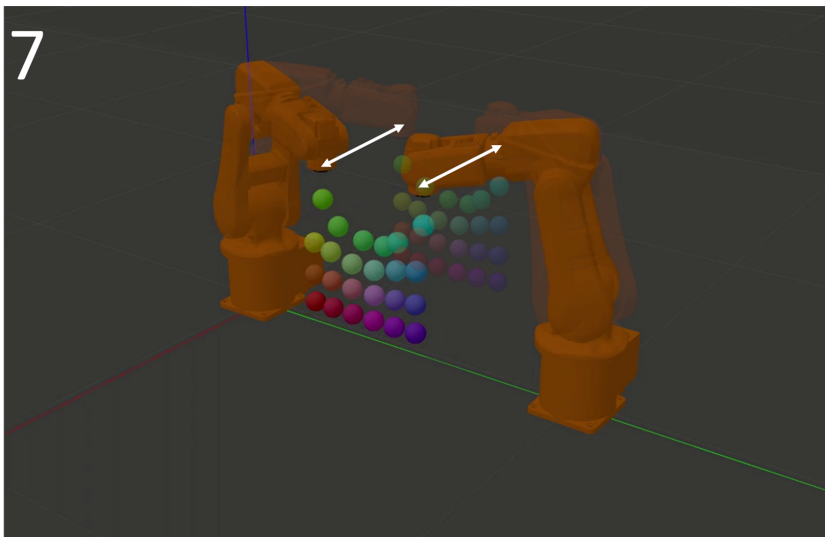


Figura 7.15: Composición de fotogramas de la simulación final - Zarandeo de la tela. Se muestra el movimiento de zarandeo con los robots de una tela de 20x30 centímetros y 4x6 esferas.

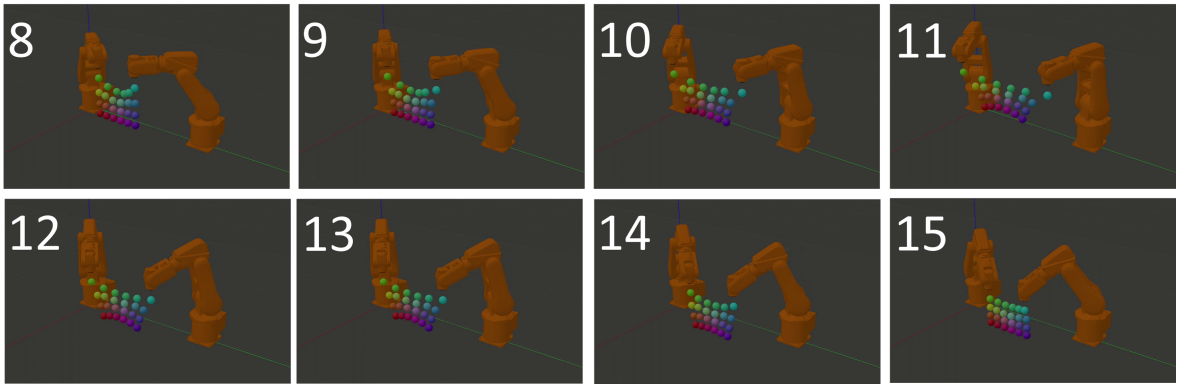


Figura 7.16: Fotogramas de la simulación final - Estiramiento de la tela. Se muestran ocho fotogramas en los que los robots estiran una tela de 20x30 centímetros y 4x6 esferas.

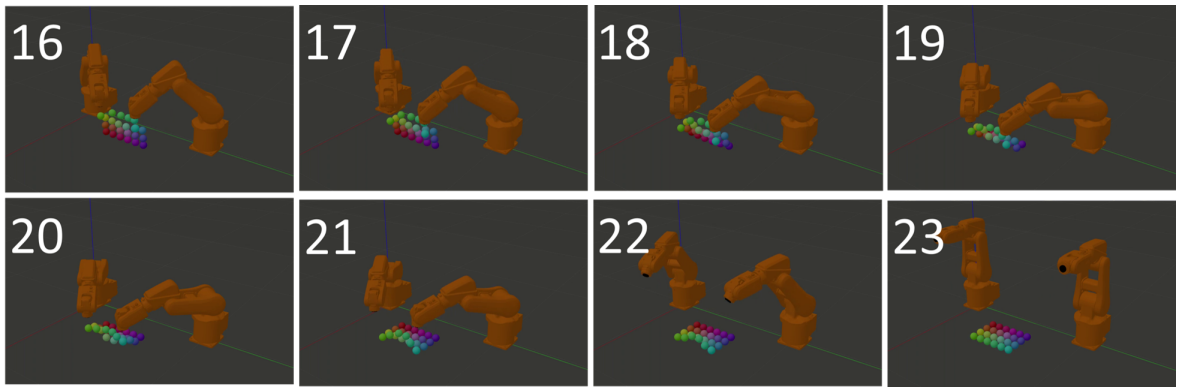


Figura 7.17: Fotogramas de la simulación final - Recolocación de la tela. Se muestran ocho fotogramas en los que los robots colocan una tela de 20x30 centímetros y 4x6 esferas de nuevo en el suelo, la sueltan y vuelven a su posición inicial.

Capítulo 8

Conclusiones y trabajo futuro

Para la realización de este proyecto, se han marcado como objetivos la simulación de un objeto deformable con la utilización de varios robots para su manipulación, todo de una forma generalizable para que pueda ser usada con cualquier robot sin necesidad de rehacer todo el trabajo. En primer lugar se han aprendido las tecnologías ROS, Gazebo y MoveIt. Después, se han modificado los archivos del robot IRB120 en ROS de ABB para que permitan incluir un número personalizado de estos robots en una simulación en el programa Gazebo, y para que se puedan controlar individualmente con MoveIt y RViz, así como con la interfaces de MoveIt en C++, todo esto por medio de ROS. También se ha incluido en la simulación una implementación del modelo masa-muelle-amortiguador de objeto deformable con forma de tela y cuyas características están parametrizadas, por medio de un plugin de Gazebo incluido en el modelo de la tela elástica. Además, se han desarrollado varios programas de prueba para observar el comportamiento de la simulación.

8.1. Valoración de los resultados

Podemos concluir que se han obtenido unos resultados positivos, puesto que la prueba final demuestra que el objeto es capaz de deformarse gracias a los movimientos que realizan los robots tras agarrarlo. Además el proyecto permitiría la sustitución de los IRB120 de ABB por otros robots en la simulación, o incluso con robots reales.

Por otro lado, los principios que llevaron a la creación de ROS nos resultan perfectos para este Trabajo. Sin embargo, el desarrollo de entornos multi-robot en ROS 1 no es una práctica estandarizada y es un proceso lento y arduo. Gazebo y MoveIt nos permiten realizar una simulación muy completa y de gran complejidad, con muchas formas de interactuar con la misma y añadir elementos adicionales que se adecuarían más a un entorno real (como el ejemplo con la mesa).

También resulta adecuado el algoritmo de *Mass-Spring-Damping*, pues es sencillo

pero su comportamiento es muy similar a un objeto real, salvo los casos en los que el sistema se volviera inestable. Además sus parámetros nos permiten realizar muchas pruebas para adecuarlos a nuestro objeto. Si deseáramos objetos de más complejidad, se podrían crear objetos de 3 dimensiones, pues el código ya tiene herramientas para crear cubos.

Finalmente, en este Trabajo, la estructura para la interacción entre el objeto y los robots es muy precisa, ya que asume que, al ser una simulación, los robots tendrán pleno conocimiento de la posición del objeto.

8.2. Trabajo futuro

Debido a las limitaciones de ROS 1, en el futuro se podría realizar una versión de este Trabajo en ROS 2, el cual se está desarrollando con este caso de uso en mente (y con él, Gazebo 11.x, MoveIt2, RViz2) [87] y así reducir la cantidad de código de ABB que ha tenido que ser modificado.

Por otro lado, los robots siempre deben estar separados para evitar colisiones y es por esto que en el futuro se podría implementar un método para que los robots sepan la posición en la que se encuentran, para poder cooperar más fácilmente, por ejemplo, para realizar costuras, que son trabajos de gran precisión y para los que se necesitarían los dos robots juntos.

También se podría desarrollar una forma de introducir modelos complejos (o *meshes*) y tratar los vértices del mismo como las esferas de este Trabajo. Igualmente, se podrían probar distintos algoritmos para objetos deformables, como *As Rigid As Possible* (ARAP), que es un sistema más complejo, pero sirve para crear objetos más consistentes, puesto que este algoritmo conserva las formas mejor [88], o incluso se podría modificar el algoritmo actual para usar otro tipo de muelles, para simular otro tipo de objetos, como el tejido humano (véase [71]).

ROS nos ofrece la posibilidad de ejecutar experimentos con robots reales, por lo que en un Trabajo futuro, se podrían realizar pruebas con robots ABB IRB 120 con un objeto como pudiera ser una tela gruesa. Se debería adaptar el código para poder interactuar con la herramienta que se acople a los robots, como pudieran ser ventosas o preferiblemente, pinzas.

En la simulación, los robots conocen su entorno en todo momento, sin embargo, en el mundo real esto sería imposible sin una forma de detección, por ello, un Trabajo futuro podría incluir un sistema de percepción, como podrían ser sistemas de visión o ultrasonidos para observar el entorno y tomar una decisión para poder manipular un objeto deformable real.

Figura 8.1: Perspectivas futuras. De arriba abajo, y de izquierda a derecha: ROS 2 [21]; Colaboración entre robots [89]; Cámara Kinect de XBOX 360 [90]; Robot ABB IRB120 de la Universidad de Zaragoza, interactuando con una persona [91].

Capítulo 9

Bibliografía

[1] An Automatic Block-Setting Crane. *Meccano Magazine*, 23(3):172, 1938.

[2] Lene Kromann, Nikolaj Malchow-Møller, Jan Rose Skaksen, and Anders Sørensen. Automation and productivity—a cross-country, cross-industry comparison. *Industrial and Corporate Change*, 29(2):265–287, 07 2019.

[3] Rafael Herguedas, Gonzalo López Nicolás, Rosario Aragüés, and Carlos Sagüés. Survey on multi-robot manipulation of deformable objects, 2019. IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2019).

[4] Miguel Aranda, Juan Antonio Corrales, and Youcef Mezouar. Deformation-based shape control with a multirobot system, 2019. IEEE International Conference on Robotics and Automation.

[5] Daniel Seita, Pete Florence, Jonathan Tompson, Erwin Coumans, Vikas Sindhwani, Ken Goldberg, and Andy Zeng. Learning to rearrange deformable cables, fabrics, and bags with goal-conditioned transporter networks. *CoRR*, abs/2012.03385, 2020.

[6] Dominik Henrich and Heinz Wörn. *Robot manipulation of deformable objects*. Springer Science & Business Media, 2012.

[7] Akihiko Yamaguchi. Science of robot cooking. `http://akihikoy.net/p/cook.html`, 2016. Consultado el 30 de Agosto de 2021.

[8] Florencio Jesús Cembranos Nistal. *Automatismos eléctricos, neumáticos e hidráulicos*. Editorial Paraninfo, 2008.

[9] IRB 120. https://new.abb.com/products/robotics/robots-industriales/irb-120. Descripción del Robot Industrial IRB 120 de ABB. Consultado el: 26 de Agosto de 2021.

[10] About ROS. https://www.ros.org/about-ros/. Descripción del Sistema Operativo Robótico. Consultado el: 27 de Julio de 2021.

[11] ROS history. https://www.ros.org/history/. Historia de la creación del Sistema Operativo Robótico. Consultado el: 27 de Julio de 2021.

[12] ROS @ OSRF. https://web.archive.org/web/20140815190347/https://www.osrfoundation.org/blog/ros-at-osrf.html. Transición de la administración de ROS de Willow Garage a OSRF. Captura recuperada por medio de Wayback Macine del 15 de Agosto de 2014.

[13] Welcome to Open Robotics. https://www.osrfoundation.org/welcome-to-open-robotics/. Cambio de nombre de la Open Source Robotics Foundation a Open Robotics. Consultado el: 27 de Julio de 2021.

[14] Is ROS for me? https://www.ros.org/is-ros-for-me/. ¿Es ROS para mí? Ventajas de su uso. Consultado el: 27 de Julio de 2021.

[15] Christina Cardoza. Inside the robot operating system, the robotics industry and the open source robotics foundation. *SD Times - Software Development News*, 2015.

[16] ROS core components. https://www.ros.org/core-components/. Características principales de ROS. Consultado el: 27 de Julio de 2021.

[17] ROS wiki - abb_irb120_support. http://wiki.ros.org/abb_irb120_support?distro=kinetic.

[18] ROS wiki - installation. http://wiki.ros.org/es/ROS/Installation. Instalación de ROS. Consultado el: 27 de Agosto de 2021.

[19] ROS concepts. http://wiki.ros.org/ROS/Concepts. Conceptos básicos de ROS. Consultado el: 27 de Julio de 2021.

[20] ROS topics. http://wiki.ros.org/Topics. Definición del concepto de Topic en ROS. Consultado el: 27 de Julio de 2021.

[21] ROS press kit. https://www.ros.org/press-kit/. Pack de imágenes de ROS para prensa. Obtenido el: 20 de Agosto de 2021.

[22] ROS parameter server. http://wiki.ros.org/Parameter%20Server. Definición del Parameter Server de ROS. Consultado el: 27 de Julio de 2021.

[23] Plugins available in gazebo_plugins. http://gazebosim.org/tutorials?tut=ros_gzplugins#Pluginsavailableingazebo_plugins. Plugins disponibles para Gazebo. Consultado el: 3 de Agosto de 2021.

[24] Gazebo beginner: Overview. http://www.gazebosim.org/tutorials?tut=guided_b1. Introducción a Gazebo. Consultado el: 3 de Agosto de 2021.

[25] Tutorial: ROS integration overview. http://gazebosim.org/tutorials?tut=ros_overview. Integración Gazebo + ROS. Consultado el: 3 de Agosto de 2021.

[26] ROS wiki - abb_irb120_gazebo. http://wiki.ros.org/abb_irb120_gazebo?distro=kinetic. Paquete abb_irb120_gazebo. Consultado el: 4 de Agosto de 2021.

[27] Gazebo - installing gazebo_ros_pkgs (ROS 1). http://gazebosim.org/tutorials?tut=ros_installing. Instalación de MoveIt. Consultado el: 27 de Agosto de 2021.

[28] GitHub - ros-planning/moveit. https://github.com/ros-planning/moveit. Repositorio de MoveIt en GitHub. Consultado el: 3 de Agosto de 2021.

[29] YouTube - MoveIt capabilities overview. https://youtu.be/7KvF7Dj7bz0. Vídeo en YouTube con las capacidades de MoveIt. Consultado el: 3 de Agosto de 2021.

[30] Moveit tutorials (ROS Melodic). http://docs.ros.org/en/melodic/api/moveit_tutorials/html/index.html. Tutoriales de MoveIt. Consultado el: 3 de Agosto de 2021.

[31] ROS wiki - abb_irb120_moveit_config. http://wiki.ros.org/abb_irb120_moveit_config?distro=kinetic. Paquete abb_irb120_moveit_config. Consultado el: 4 de Agosto de 2021.

[32] MoveIt tutorials - installation (ROS Melodic). http://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/getting_started/getting_started.html. Instalación de MoveIt. Consultado el: 27 de Agosto de 2021.

[33] ROS robots. https://robots.ros.org/. Algunos de los robots compatibles con ROS.

[34] Robots - MoveIt. https://moveit.ros.org/robots/. Robots compatibles con MoveIt. Consultado el: 14 de Septiembre de 2021.

[35] ABB - información detallada para IRB 120. https://new.abb.com/products/es/3HAC031431-001/irb-120. Consultado el: 14 de Septiembre de 2021.

[36] ROS training for industry - motion planning with a multi robot system. https://ut-ims-robotics.github.io/ros_training/html/day5/multirobot_mp.html. Tutorial de utilización de un modelo dual para el control de múltiples robots en MoveIt - Universidad de Tartu. Consultado el: 10 de Agosto de 2021.

[37] ROS wiki - roslaunch/XML. http://wiki.ros.org/roslaunch/XML. Formato XML para su ejecución con *roslaunch*. Consultado el: 4 de Agosto de 2021.

[38] ROS wiki - roslaunch. http://wiki.ros.org/roslaunch. Descripción del comando *roslaunch*. Consultado el: 4 de Agosto de 2021.

[39] GitHub - ros-industrial/abb_experimental - joint_names_irb120_3_58.yaml. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_support/config/joint_names_irb120_3_58.yaml. Fichero *joint_names_irb120_3_58.yaml* del paquete *abb_irb120_support* (rama *kinetic-devel*). Consultado el: 4 de Agosto de 2021.

[40] GitHub - ros-industrial/abb_experimental - joint_limits.yaml. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_moveit_config/config/joint_limits.yaml. Fichero *joint_limits.yaml* del paquete *abb_irb120_moveit_config* (rama *kinetic-devel*. Consultado el: 4 de Agosto de 2021.

[41] GitHub - ros-industrial/abb_experimental - kinematics.yaml. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_moveit_config/config/kinematics.yaml. Fichero *kinematics.yaml* del paquete *abb_irb120_moveit_config* (rama *kinetic-devel*. Consultado el: 4 de Agosto de 2021.

[42] ROS wiki - URDF. http://wiki.ros.org/urdf. Descripción del paquete URDF en ROS. Consultado el: 4 de Agosto de 2021.

[43] ROS wiki - SRDF. http://wiki.ros.org/srdf. Descripción del paquete SRDF en ROS. Consultado el: 4 de Agosto de 2021.

[44] ROS wiki - xacro. http://wiki.ros.org/xacro. Descripción del paquete xacro en ROS. Consultado el: 4 de Agosto de 2021.

[45] GitHub - ros-industrial/abb_experimental - irb120_3_58_macro.xacro. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_gazebo/urdf/irb120_3_58_macro.xacro. Fichero *irb120_3_58_macro.xacro* del paquete *abb_irb120_gazebo* (rama *kinetic-devel*. Consultado el: 4 de Agosto de 2021.

[46] ROS wiki - names. http://wiki.ros.org/Names. Descripción de la jerarquía de nombres en ROS. Consultado el: 4 de Agosto de 2021.

[47] Gazebo - tutorial: ROS communication - services. http://gazebosim.org/tutorials/?tut=ros_comm. Tutorial para la comunicación entre ROS y Gazebo. Consultado el: 10 de Agosto de 2021.

[48] GitHub - ros-industrial/abb_experimental - load_irb120_3_58.launch. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_gazebo/launch/load_irb120_3_58.launch. Fichero *load_irb120_3_58.launch* del paquete *abb_irb120_gazebo* (rama *kinetic-devel*). Consultado el: 10 de Agosto de 2021.

[49] ROS wiki - robot_state_publisher. http://wiki.ros.org/robot_state_publisher. Descripción del paquete *robot_state_publisher*. Consultado el: 10 de Agosto de 2021.

[50] GitHub - ros-controls/ros_controllers - joint_state_controller.h. https://github.com/ros-controls/ros_controllers/blob/melodic-devel/joint_state_controller/include/joint_state_controller/joint_state_controller.h. Fichero *oint_state_controller.h* del paquete *ros_controllers* (rama *melodic-devel*). Consultado el: 10 de Agosto de 2021.

[51] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernández Perdomo. ros_control: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2017.

[52] MoveIt - concepts. https://moveit.ros.org/documentation/concepts/. Conceptos básicos de MoveIt. Consultado el: 16 de Septiembre de 2021.

[53] Gazebo - media. http://gazebosim.org/media. Pack de imágenes de Gazebo. Obtenido el: 29 de Agosto de 2021.

[54] MoveIt - press kit. https://moveit.ros.org/about/press_kit/. Pack de imágenes de MoveIt para prensa. Obtenido el: 29 de Agosto de 2021.

[55] GitHub - ros-visualization/rviz. https://github.com/ros-planning/moveit/blob/melodic-devel/moveit_ros/planning_interface/move_group_interface/src/move_group_interface.cpp#L1232. Repositorio oficial de RViz. Consultado el: 16 de Septiembre de 2021.

[56] MoveIt - move group C++ interface. http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/move_group_interface/move_group_interface_tutorial.html. Tutorial de la interfaz de MoveIt para C++). Consultado el: 9 de Agosto de 2021.

[57] ROS documentation - moveit::planning_interface::MoveGroupInterface class reference (melodic). http://docs.ros.org/en/melodic/api/moveit_ros_planning_interface/html/classmoveit_1_1planning__interface_1_1MoveGroupInterface.html. Manual de usuario de la interfaz MoveGroupInterface de MoveIt). Consultado el: 10 de Agosto de 2021.

[58] GitHub - ros-industrial/abb_experimental - abb_irb120_3_58.srdf. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_moveit_config/config/abb_irb120_3_58.srdf. Fichero *abb_irb120_3_58.srdf* del paquete *abb_irb120_moveit_config* (rama *kinetic-devel*). Consultado el: 6 de Agosto de 2021.

[59] GitHub - isocpp/logos. https://github.com/isocpp/logos. Repositorio *logos* del estándar ISO para C++. Obtenido el: 29 de Agosto de 2021.

[60] Gazebo - make a model. http://gazebosim.org/tutorials?tut=build_model. Vista general de la creación de modelos para Gazebo. Consultado el: 4 de Agosto de 2021.

[61] Gazebo - tutorial: Using roslaunch to start Gazebo, world files and URDF models - using roslaunch to spawn URDF robots. http://gazebosim.org/tutorials?tut=ros_roslaunch&cat=connect_ros#UsingroslaunchtoSpawnURDFRobots. Tutorial para generar robots URDF en Gazebo mediante ROS. Consultado el: 4 de Agosto de 2021.

[62] Gazebo tutorials - category: Physics library. http://gazebosim.org/tutorials?cat=physics. Todos los tutoriales de Gazebo sobre la simulación de físicas. Consultado el: 11 de Agosto de 2021.

[63] Gazebo - tutorial: Using gazebo plugins with ros. http://gazebosim.org/tutorials?tut=ros_gzplugins. Tutorial con ejemplos del uso de *plugins* en Gazebo. Consultado el: 11 de Agosto de 2021.

[64] Gazebo - overview of gazebo plugins. http://gazebosim.org/tutorials/?tut=plugins_hello_world. Vista general de plugins en Gazebo. Ejemplo para crear un primer plugin. Consultado el: 11 de Agosto de 2021.

[65] Gazebo - world plugins. http://gazebosim.org/tutorials?tut=plugins_world. Ejemplo para crear un plugin de tipo *WorldPlugin*. Consultado el: 11 de Agosto de 2021.

[66] Gazebo - model plugins. http://gazebosim.org/tutorials?tut=plugins_model. Ejemplo para crear un plugin de tipo *ModelPlugin*. Consultado el: 11 de Agosto de 2021.

[67] Gazebo - rendering::Visual class reference. http://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1rendering_1_1Visual.html. Manual de Usuario de la clase gazebo::rendering::Visual. Consultado el: 11 de Agosto de 2021.

[68] Gazebo - physics::World class reference. http://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1physics_1_1World.html. Manual de Usuario de la clase gazebo::physics::World. Consultado el: 13 de Agosto de 2021.

[69] Gazebo - gazebo::msgs namespace reference. https://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/namespacegazebo_1_1msgs.html. Manual de Usuario del namespace gazebo::msgs. Consultado el: 13 de Agosto de 2021.

[70] Sdformat - sdf::v11 namespace reference. http://osrf-distributions.s3.amazonaws.com/sdformat/api/dev/namespacesdf_1_1v11.html. Manual de Usuario del namespace sdf::v11. Consultado el: 13 de Agosto de 2021.

[71] Sarah FF Gibson and Brian Mirtich. A survey of deformable modeling in computer graphics. Technical report, Citeseer, 1997.

[72] Matthias Teschner. Simulation in computer graphics. University of Freiburg. https://cg.informatik.uni-freiburg.de/teaching.htm#material.

[73] C. Henry. Mass-spring-system model for real time expressive behaviour synthesis why and how to use physical model in pure data. 2015.

[74] Joseph C. Watkins. The mass-spring oscillator. University of Arizona. Retrieved from: https://www.math.arizona.edu/~jwatkins/h-ode.pdf.

[75] M. Alex O. Vasilescu. Physically-based modeling: Mass-spring systems. Massachusetts Institute of Technology. Retrieved from: http://alumni.media.mit.edu/~maov/classes/comp_photo_vision08f/lect/28_mass_springs_all.pdf.

[76] Auralius Manurung. Deformable object with interconnected mass-spring-damper. https://github.com/auralius/matlab-mass-spring-damper-network-deformable-object, 2021. Consultado el: 18 de Agosto de 2021.

[77] Rafael Herguedas, Gonzalo López-Nicolás, and Carlos Sagüés. Collision-free transport of 2d deformable objects, 2021. The 21st International Conference on Control, Automation and Systems (ICCAS 2021).

[78] Gazebo - physics::Model class reference. http://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1physics_1_1Model.html. Manual de Usuario de la clase gazebo::physics::Model. Consultado el: 13 de Agosto de 2021.

[79] Gazebo - physics::Link class reference. https://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1physics_1_1Link.html. Manual de Usuario de la clase gazebo::physics::Link. Consultado el: 17 de Agosto de 2021.

[80] gazebo_msgs/ModelStates message definition. http://docs.ros.org/en/jade/api/gazebo_msgs/html/msg/ModelStates.html. Definición del mensaje *gazebo_msgs/ModelStates*. Consultado el: 18 de Agosto de 2021.

[81] MoveIt - moveit::planning_interface::PlanningSceneInterface class reference. http://docs.ros.org/en/indigo/api/moveit_ros_planning_interface/html/classmoveit_1_1planning__interface_1_1PlanningSceneInterface.html. Manual de Usuario de la clase

moveit::planning_interface::PlanningSceneInterface. Consultado el: 29 de Agosto de 2021.

[82] moveit_msgs/CollisionObject Message. `http://docs.ros.org/en/melodic/api/moveit_msgs/html/msg/CollisionObject.html`. Descripción del mensaje de tipo moveit_msgs/CollisionObject Message. Consultado el: 29 de Agosto de 2021.

[83] ROS Wiki - Create a URDF for an Industrial Robot. `http://wiki.ros.org/Industrial/Tutorials/Create%20a%20URDF%20for%20an%20Industrial%20Robot`. Tutorial de ROS para la creación de modelos en URDF para un robot industrial. Consultado el: 29 de Agosto de 2021.

[84] Gazebo - Tutorial: Using a URDF in Gazebo. `http://gazebosim.org/tutorials/?tut=ros_urdf`. Tutorial de Gazebo sobre el uso de URDF. Consultado el: 29 de Agosto de 2021.

[85] GitHub - ignitionrobotics/sdformat - URDF to SDF conversion ignores links without inertia #199. `https://github.com/ignitionrobotics/sdformat/issues/199`. Propuesta (*Issue*) del repositorio *ignitionrobotics/sdformat* con comentarios de los colaboradores sobre el comportamiento de las articulaciones en Gazebo. Consultado el: 29 de Agosto de 2021.

[86] GitHub - ros-industrial/abb_experimental - irb120_3_58_arm_controller.yaml. `https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_gazebo/config/irb120_3_58_arm_controller.yaml`. Fichero *irb120_3_58_arm_controller.yaml* del paquete *abb_irb120_gazebo* (rama *kinetic-devel*. Consultado el: 4 de Agosto de 2021.

[87] Brian Gerkey. Why ros 2? `https://design.ros2.org/articles/why_ros2.html`. Explicación de las causas que llevan al desarrollo de ROS 2. Consultado el: 29 de Agosto de 2021.

[88] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of EUROGRAPHICS/ACM SIGGRAPH Symposium on Geometry Processing*, pages 109–116, 2007.

[89] YouTube - ABB Robotics - new small robot - IRB 120. `https://www.youtube.com/watch?v=-39W3fdD5WA&t=68s&ab_channel=ABBRobotics`. Vídeo

promocional en YouTube con las capacidades del robot ABB IRB120. Consultado el: 29 de Agosto de 2021.

[90] Wikimedia Commons contributors. Kinect Sensor at E3 2010. https://commons.wikimedia.org/w/index.php?title=File:Kinect_Sensor_at_E3_2010_(front).jpg&oldid=508623234. Imagen de una cámara Kinect de XBOX 360. Consultado el: 29 de Agosto de 2021.

[91] Rosario Aragüés Muñoz, López Nicolás Gonzalo, and Sagüés Blázquiz Carlos. La Universidad de Zaragoza participa en un proyecto europeo para automatizar procesos industriales y mejorar la calidad de vida de los trabajadores. https://www.unizar.es/noticias/la-universidad-de-zaragoza-participa-en-un-proyecto-europeo-para-automatizar-procesos-0. Consultado el: 29 de Agosto de 2021.

[92] GitHub - ros-industrial/abb_experimental - irb120_control.launch. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_gazebo/launch/irb120_control.launch. Fichero *irb120_control.launch* del paquete *abb_irb120_gazebo* (rama *kinetic-devel*. Consultado el: 4 de Agosto de 2021.

[93] ROS wiki - remap. http://wiki.ros.org/roslaunch/XML/remap. Descripción de la cláusula *remap*. Consultado el: 9 de Agosto de 2021.

[94] GitHub - ros-industrial/abb_experimental - irb120_3_58_gazebo.launch. https://github.com/ros-industrial/abb_experimental/blob/kinetic-devel/abb_irb120_gazebo/launch/irb120_3_58_gazebo.launch. Fichero *irb120_3_58_gazebo.launch* del paquete *abb_irb120_gazebo* (rama *kinetic-devel*). Consultado el: 6 de Agosto de 2021.

[95] ROS wiki - group. http://wiki.ros.org/roslaunch/XML/group. Descripción de la cláusula *group*. Consultado el: 9 de Agosto de 2021.

[96] ROS wiki - coordinate frame conventions. http://wiki.ros.org/geometry/CoordinateFrameConventions. Convenciones aplicadas a los espacios de coordenadas de ROS. Consultado el: 9 de Agosto de 2021.

[97] GitHub - ros-industrial/abb_experimental. https://github.com/ros-industrial/abb_experimental. Repositorio *abb_experimental* desarrollado por ABB para el uso de sus robots en ROS. Consultado el: 27 de Agosto de 2021.

[98] ROS wiki - create a catkin workspace. http://wiki.ros.org/catkin/Tutorials/create_a_workspace. Tutorial de ROS para la creación de entornos de trabajo de catkin. Consultado el: 27 de Agosto de 2021.

[99] ROS Wiki - Creating a ROS msg and srv. http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv. Creación de mensajes y servicios propios Consultado el: 29 de Agosto de 2021.

[100] Microsoft. Visual Studio Marketplace - ROS. https://marketplace.visualstudio.com/items?itemName=ms-iot.vscode-ros. Extensión ROS en el mercado de extensiones de Visual Studio Code. Consultado el: 29 de Agosto de 2021.

[101] GitHub - andrewknoll/multiple_abb_irb120. https://github.com/andrewknoll/multiple_abb_irb120. Repositorio en GitHub de los ficheros de este Trabajo.

[102] MoveIt - low level controllers - remapping /joint_states topic. http://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/controller_configuration/controller_configuration_tutorial.html#remapping-joint-states-topic. Tutorial de MoveIt sobre cómo hacer un *remap* del topic */joint_states* para un nodo *move_group*. Consultado el: 14 de Septiembre de 2021.

[103] GitHub - ros-industrial/abb_experimental. https://github.com/ros-planning/moveit/blob/melodic-devel/moveit_ros/planning_interface/move_group_interface/src/move_group_interface.cpp#L1232. Línea 1232 del fichero *move_group_interface.cpp* que describe la clase *MoveGroupInterface* en la rama *melodic-devel* del repositorio oficial de *MoveIt*. Consultado el: 2 de Septiembre de 2021.

[104] GitHub - ros-planning/moveit - current_state_monitor.h (rama melodic-devel). https://github.com/ros-planning/moveit/blob/melodic-devel/moveit_ros/planning/planning_scene_monitor/include/moveit/planning_scene_monitor/current_state_monitor.h. Fichero *current_state_monitor.h* que describe la clase *CurrentStateMonitor* en la rama *melodic-devel* del repositorio oficial de *MoveIt*. Consultado el: 2 de Septiembre de 2021.

[105] GitHub - ros-planning/moveit - current_state_monitor.h (rama melodic-devel) - línea 142. https://github.com/ros-planning/moveit/blob/melodic-devel/

moveit_ros/planning_interface/move_group_interface/src/move_group_interface.cpp#L142. Línea 142 del fichero *move_group_interface.cpp* que implementa la clase *MoveGroupInterface* en la rama *melodic-devel* del repositorio oficial de *MoveIt*. Consultado el: 2 de Septiembre de 2021.

[106] GitHub - ros-planning/moveit - common_objects.cpp (rama melodic-devel) - línea 140. https://github.com/ros-planning/moveit/blob/melodic-devel/moveit_ros/planning_interface/common_planning_interface_objects/src/common_objects.cpp#L140. Línea 140 del fichero *common_objects.cpp* en la rama *melodic-devel* del repositorio oficial de *MoveIt*. Consultado el: 2 de Septiembre de 2021.

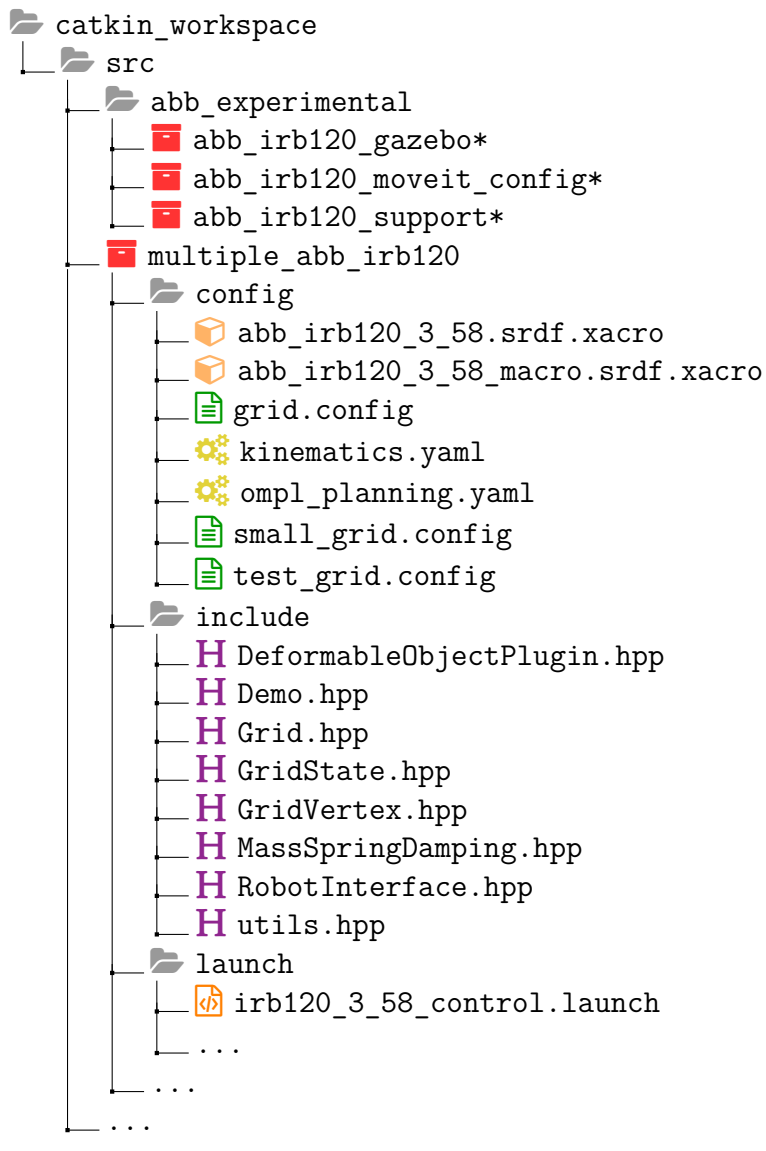
[107] Cplusplus - std::map::insert. https://www.cplusplus.com/reference/map/map/insert/. Función *insert* de la clase *std::map* de C++. Consultado el: 2 de Septiembre de 2021.

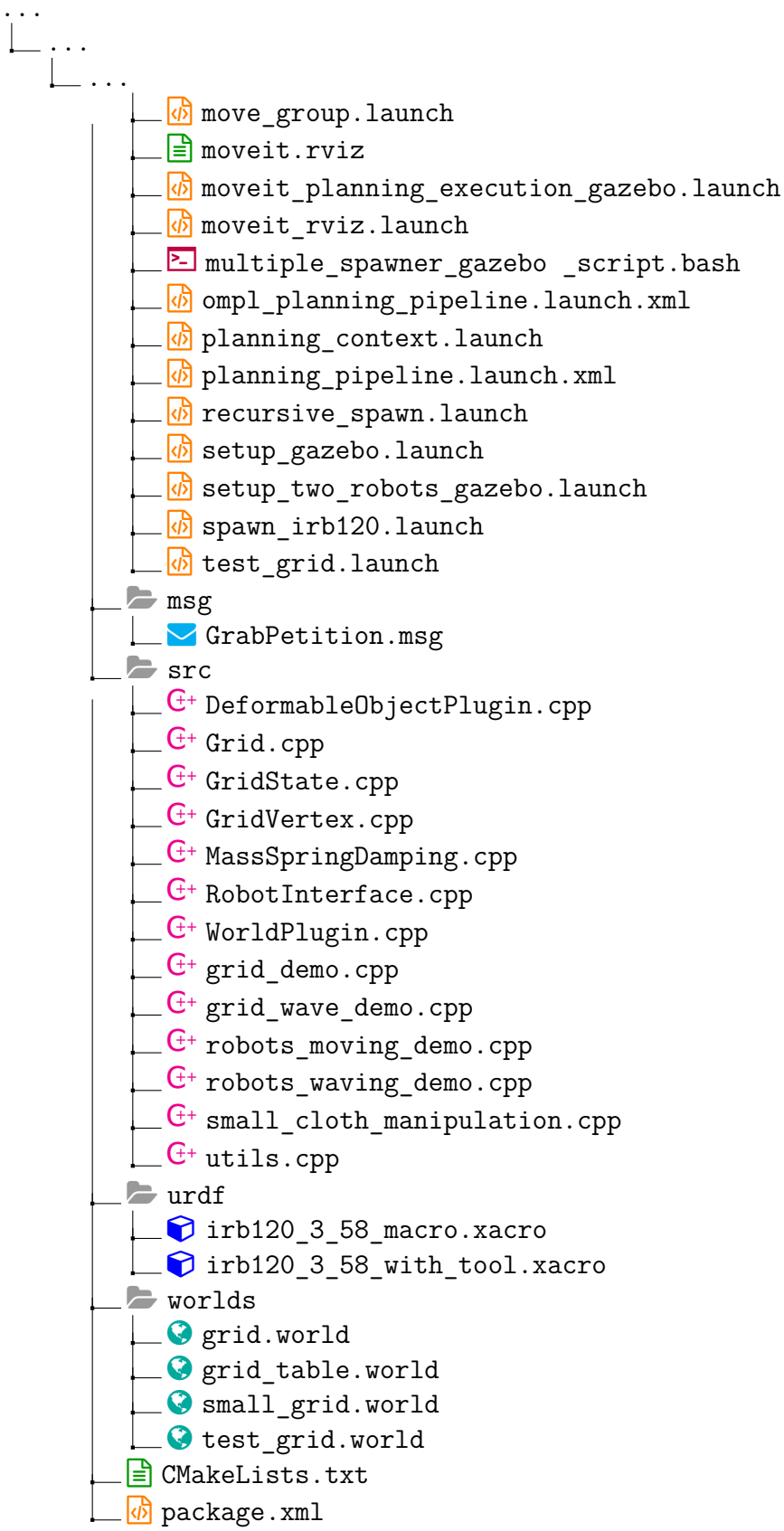
Anexos

Anexos A

Organización de los ficheros desarrollados

En este anexo se incluye en detalle la estructura de los ficheros que son estrictamente necesarios para el funcionamiento de este Trabajo.





* Se han omitido los contenidos de los paquetes desarrollados por ABB, que pueden ser obtenidos directamente desde el repositorio de la empresa en *GitHub* [97]. Para este Trabajo, se utilizó la versión para ROS Kinetic (al no existir versión para Melodic).

Anexos B

Instalación y Ejecución de los programas desarrollados

En este anexo se detalla la manera de obtener los ficheros desarrollados para el funcionamiento de los programas descritos en este Trabajo y también como ejecutarlos.

B.1. Instalación

En primer lugar, es necesario tener los ficheros descritos en el Apéndice A. Para descargarlos, se ha dispuesto de un repositorio en la plataforma *GitHub*, por lo que podemos ejecutar el comando:

```
git clone https://github.com/andrewknoll/multiple_abb_irb120
```

Si los paquetes *abb_irb120_gazebo*, *abb_irb120_moveit_config* y *abb_irb120_support* no se descargan correctamente, deberemos descargarlos manualmente y colocarlos en la carpeta *src* (obtenida desde el repositorio anterior). Si además surgiese cualquier otro problema, se propone seguir el tutorial de ROS para la creación de entornos de trabajo de *catkin* [98], y colocar tanto los paquetes de ABB como el contenido de la carpeta *multiple_abb_irb120* en la carpeta *src* del nuevo entorno de trabajo.

B.2. Configuración inicial

Para poder ejecutar las demostraciones, es necesario que usted tenga instalada la versión de ROS adecuada a su sistema operativo Ubuntu [18]. También deberá asegurarse de que usted tenga instalados Gazebo [27] y MoveIt [32]. Este Trabajo se ha probado en Ubuntu 18.04 con ROS Melodic, por lo que no se puede asegurar que funcione en otras versiones de ROS.

Antes de ejecutar cualquiera de los programas por primera vez (subsecuentes ejecuciones de otro programa no lo requerirán), deberá ejecutar los siguientes comandos:

– Deberá colocar su terminal en la carpeta en la que haya descargado los archivos

(se le denominará *catkin_workspace* a partir de ahora):
`cd catkin_workspace`

– Deberá realizar una compilación por medio de *catkin*:
`catkin_make`

– Una vez terminado el proceso, deberá ejecutar el siguiente comando para que ROS sea capaz de localizar los paquetes:
`source devel/setup.bash`

Tras la primera compilación, no será necesario ejecutar el segundo de los comandos, excepto cuando se hayan realizado cambios en el paquete.

B.3. Ejecución

A continuación se explica el método de ejecución de los distintos programas descritos en el Trabajo.

Nota: Para todas las pruebas se ha omitido que puede tener un proceso del tipo *roscore* abierto en una terminal, si desea un mayor control sobre las simulaciones.

B.3.1. Ejecución de trayectorias

En este apartado se demuestra cómo ejecutar un entorno multi-robot y enviar posiciones objetivo a los robots de forma manual.

En primer lugar, deberá abrir una terminal e iniciar el entorno en Gazebo:
`roslaunch multiple_abb_irb120 setup_two_robots_gazebo.launch`
Una ventana como la Figura B.1 se abrirá.

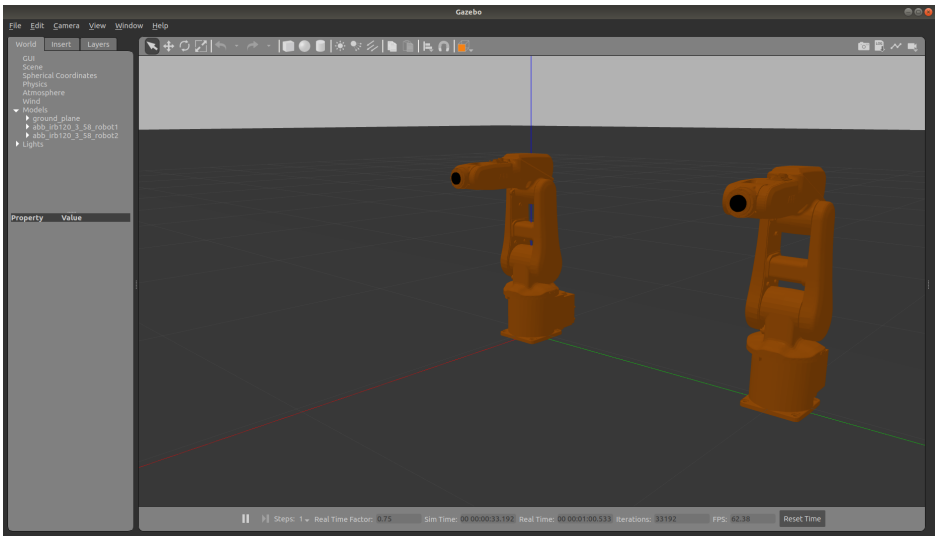


Figura B.1: Resultado de la ejecución de *setup_two_robots_gazebo.launch*

Asegúrese de que Gazebo no está pausado (la parte inferior de la ventana deberá indicar que el tiempo está avanzando).

A continuación deberá iniciar el control del robot al que desee mandar tareas. En nuestro caso, utilizaremos el robot1. Para ello, deberá ejecutar **en una nueva terminal** y sin cerrar la anterior, el siguiente comando: `roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \ robot_name:=robot1`

Ahora observará una ventana como la Figura B.2.

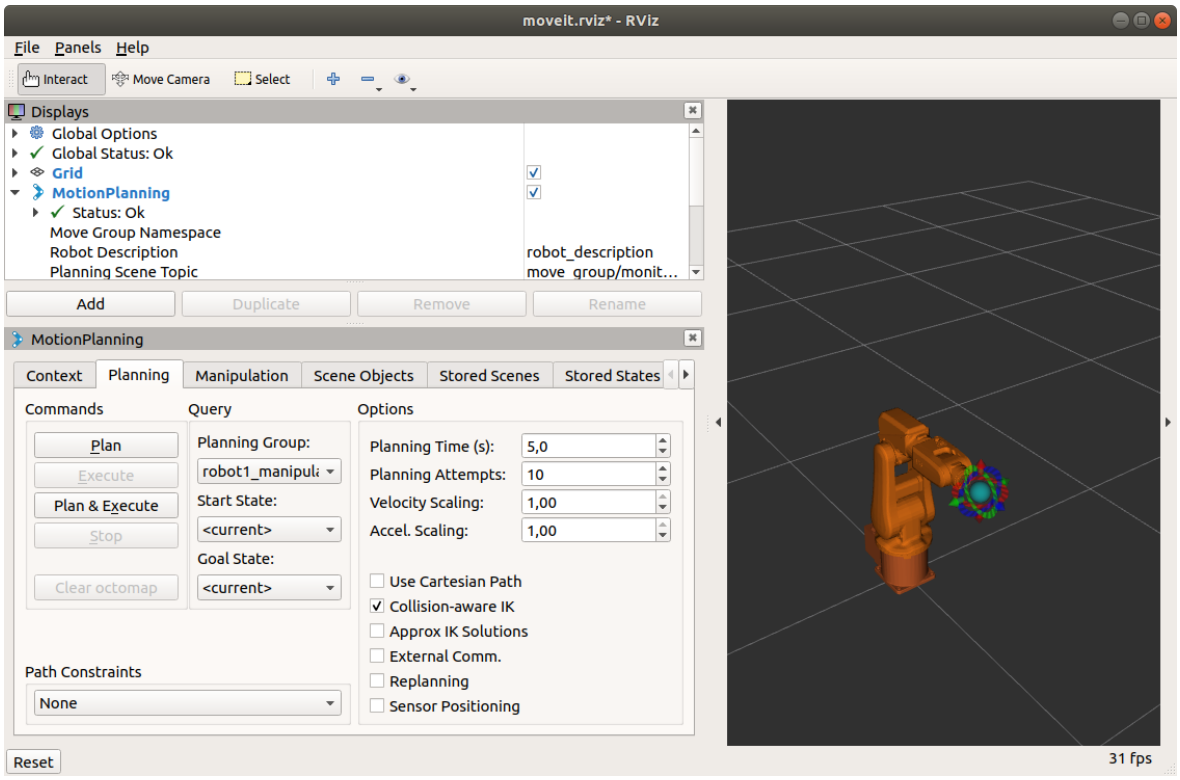


Figura B.2: Resultado de la ejecución de *moveit_planning_execution_gazebo.launch* con *robot_name:=robot1*

En esta ventana, podrá utilizar tanto la esfera turquesa situada al final del robot, como los ejes de rotación y traslación situados a su alrededor para modificar la posición del robot. También podrá navegar hasta la pestaña "*Joints*" donde podrá modificar manualmente el valor de cada articulación.

Finalmente, en la pestaña "Planning", deberá utilizar los botones "*Plan*" y "*Execute*" para que MoveIt calcule una trayectoria hasta dicho punto y después se ejecute en la simulación en Gazebo.

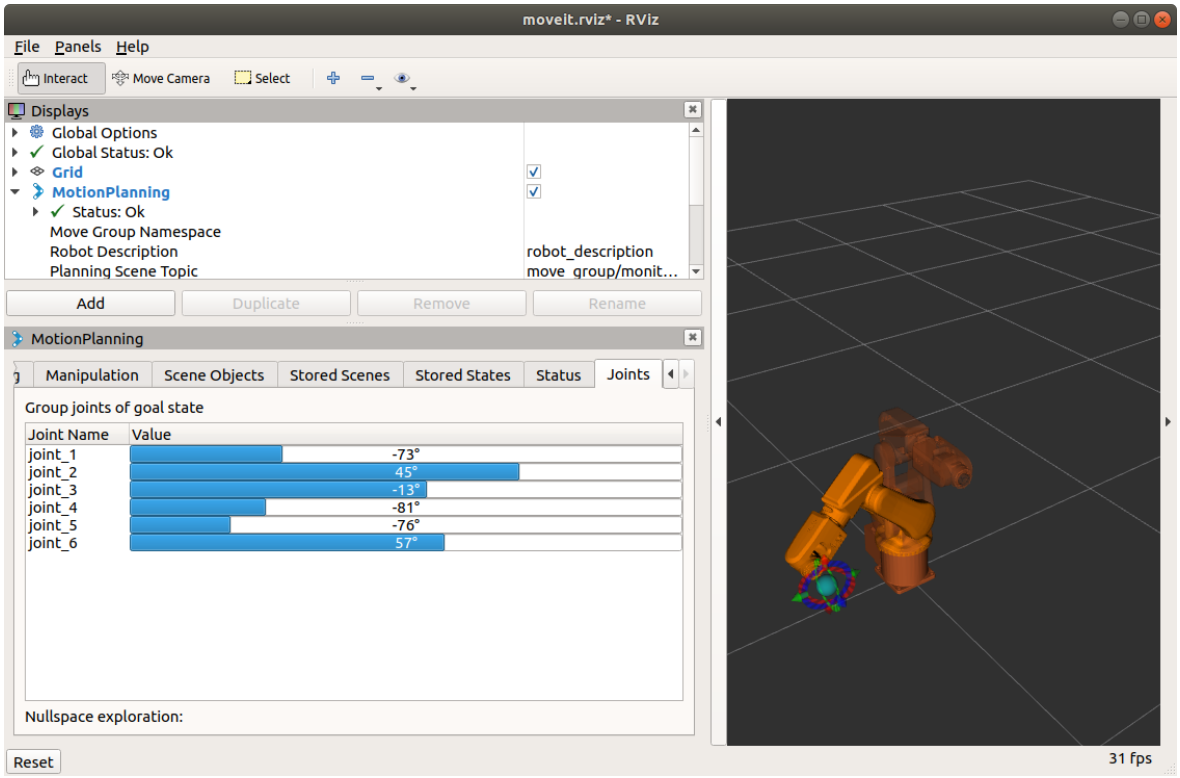


Figura B.3: Interfaz RViz con un robot al que se le ha modificado la posición.

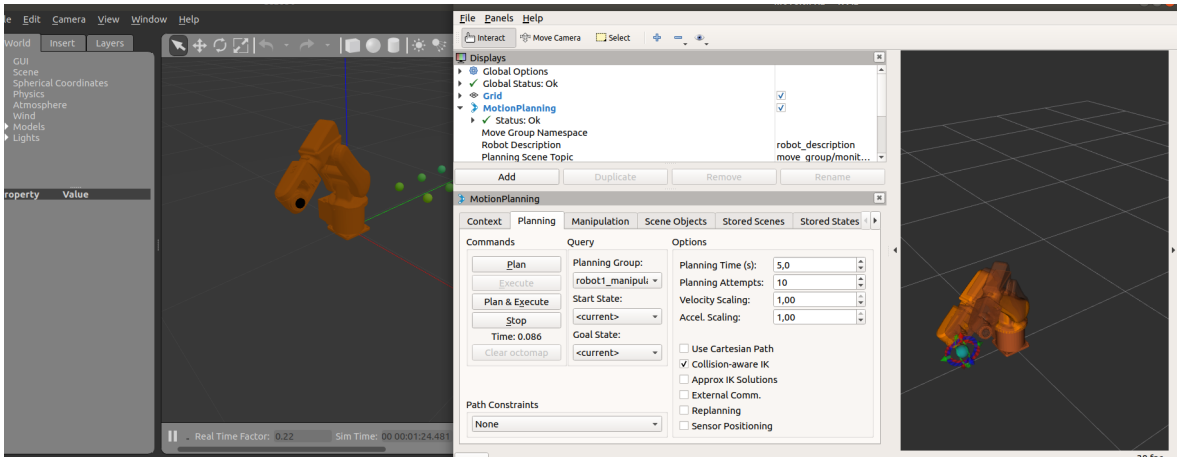


Figura B.4: Ejecución en Gazebo de la trayectoria indicada por MoveIt para una posición objetivo introducida en RViz.

B.3.2. *robots_moving_demo*

Deberá abrir **4 terminales** simultáneamente. En la primera introducirá este comando:

```
roslaunch multiple_abb_irb120 setup_two_robots_gazebo.launch
```

Una ventana como en la Figura B.1 se abrirá.

Asegúrese de que Gazebo no está pausado (la parte inferior de la ventana deberá indicar que el tiempo está avanzando). Entonces, en la segunda terminal, sin cerrar la

primera, deberá introducir el comando:

```
roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \
robot_name:=robot1
```

y de la misma forma, en la tercera sustituiremos “robot1” por “robot2”:

```
roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \
robot_name:=robot2
```

Cada uno de estos comandos abrirá una ventana de RViz como en la Figura B.2.

Finalmente, ejecutará el siguiente comando en la cuarta terminal, sin cerrar las anteriores:

```
rosrun multiple_abb_irb120 robots_moving_demo
```

Podrá observar el movimiento de los robots en la ventana de Gazebo. Para cerrarlo, deberá interrumpir la ejecución de todas las terminales, por medio de la combinación de teclas `Ctrl` + `C`.

B.3.3. *robots_waving_demo*

En primer lugar, deberá elegir un número de robots para colocar en la simulación. En este caso, se van a utilizar 5 robots. Deberá abrir 2 terminales más una terminal por robot simultáneamente, en nuestro caso, **7 terminales**. En la primera introducirá este comando:

```
./src/multiple_abb_irb120/launch/multiple_spawner_gazebo_script.bash x
```

siendo x el número de robots. En nuestro caso serán 5. Una ventana como en la Figura B.5 se abrirá.

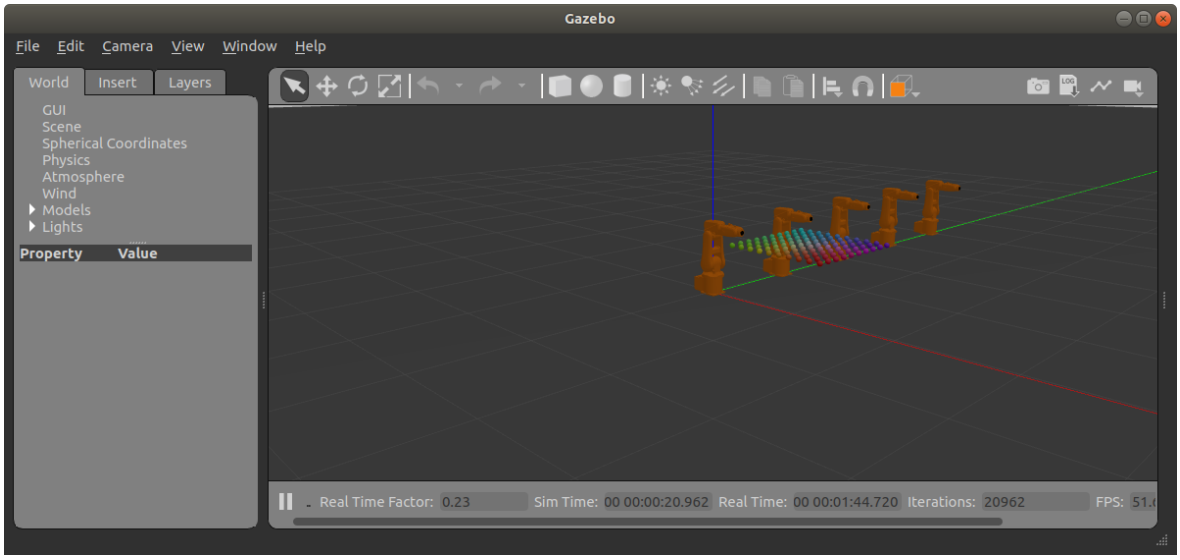


Figura B.5: Resultado de la ejecución de *multiple_spawner_gazebo_script.bash* con 5 robots

Entonces, deberá ejecutar, para cada robot, el siguiente comando en una terminal distinta:

```
roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \
robot_name:=robotx
```

siendo *robotx* el nombre del robot (en nuestro caso, *robot1*, *robot2*, *robot3*, *robot4* y *robot5*. Cada uno de estos comandos abrirá una ventana de RViz como en la Figura B.2.

Finalmente, ejecutará el siguiente comando en la cuarta terminal, sin cerrar las anteriores:

```
rosrun multiple_abb_irb120 robots_waving_demo robots:=x
```

siendo x el número de robots. En nuestro caso serán 5. Podrá observar el movimiento de los robots en la ventana de Gazebo. Para cerrarlo, deberá interrumpir la ejecución de todas las terminales, por medio de la combinación de teclas `Ctrl` + `C`.

B.3.4. *grid_demo*

Deberá abrir **4 terminales** simultáneamente. En la primera ejecutará el script de bash incluido con los ficheros:

```
./src/multiple_abb_irb120/launch/multiple_spawner_gazebo_script.bash 2
```

Es muy importante incluir un número **igual o mayor a 2** en este comando, pues es el número de robots que aparecerán en la simulación. Una ventana de este aspecto se abrirá.

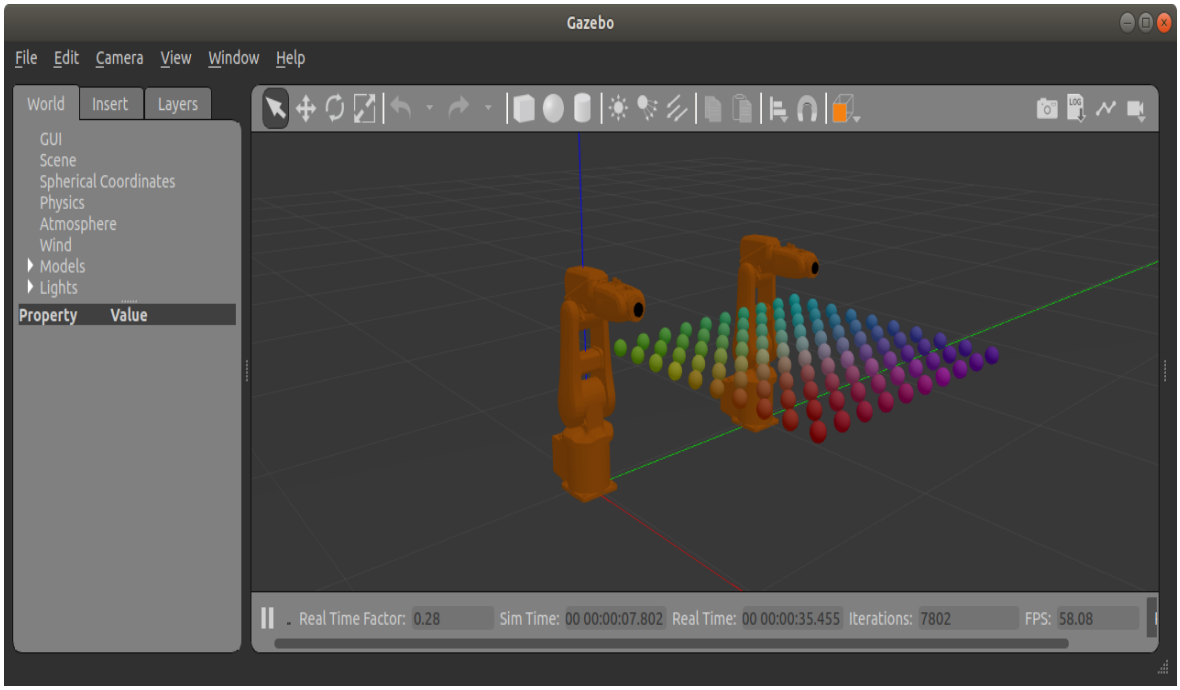


Figura B.6: Resultado de la ejecución de *multiple_spawner_gazebo_script.bash*

Entonces, en la segunda terminal, sin cerrar la primera, deberá introducir el comando:

```
roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \
robot_name:=robot1
```

y de la misma forma, en la tercera sustituiremos "robot1" por "robot2":

```
roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \
robot_name:=robot2
```

Cada uno de estos comandos abrirá una ventana de RViz como la de la Figura B.2.

Finalmente, ejecutará el siguiente comando en la cuarta terminal, sin cerrar las anteriores:

```
rosrun multiple_abb_irb120 grid_demo
```

Podrá observar el movimiento de los robots en la ventana de Gazebo.

Para cerrarlo, deberá interrumpir la ejecución de todas las terminales, por medio de la combinación de teclas `Ctrl` + `C`.

B.3.5. *test_grid*

Deberá abrir una terminal e introducir el siguiente comando:

`roslaunch multiple_abb_irb120 test_grid.launch`. (también tiene como opción utilizar el comando

`./src/multiple_abb_irb120/launch/multiple_spawner_gazebo_script.bash 0 -d`

si prefiere, pero el resultado será el mismo). Podrá observar una simulación en Gazebo como la siguiente:

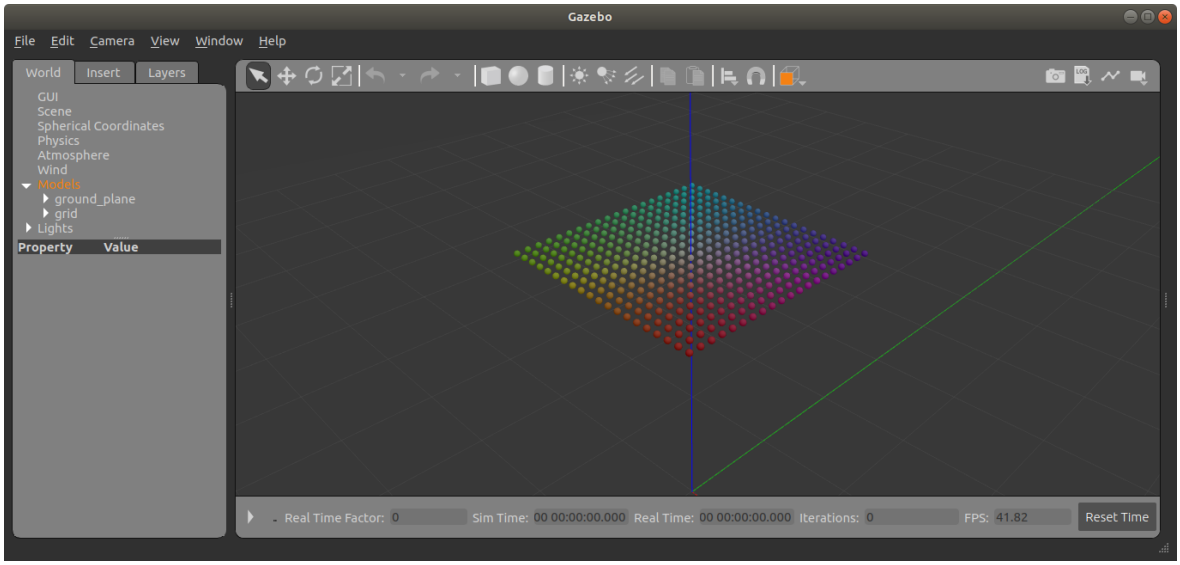


Figura B.7: Resultado de la ejecución de *multiple_spawner_gazebo_script.bash*

Para observar el comportamiento de la tela, deberá pulsar el botón de reproducción (▶) situado en la parte inferior de la ventana. La tela deberá caer lentamente y rebotará varias veces hasta quedarse en una posición de reposo.

B.3.6. Manipulación manual del objeto deformable

Si deseamos manipular nuestro objeto deformable de forma manual, podemos utilizar cualquiera de las dos demostraciones anteriores para generar nuestro objeto deformable.

También es posible la modificación de los parámetros en los archivos *grid.config* y *test_grid.config*.

En este caso, se ha modificado el archivo *test_grid.config* para que el parámetro *gravity* sea igual a 0. De esta forma, haremos que el objeto no se vea afectado por la gravedad.

En la simulación, podremos seleccionar una esfera desde la lista de enlaces del modelo, o bien haciendo click sobre la misma varias veces hasta que se encuentre rodeada por un cubo.

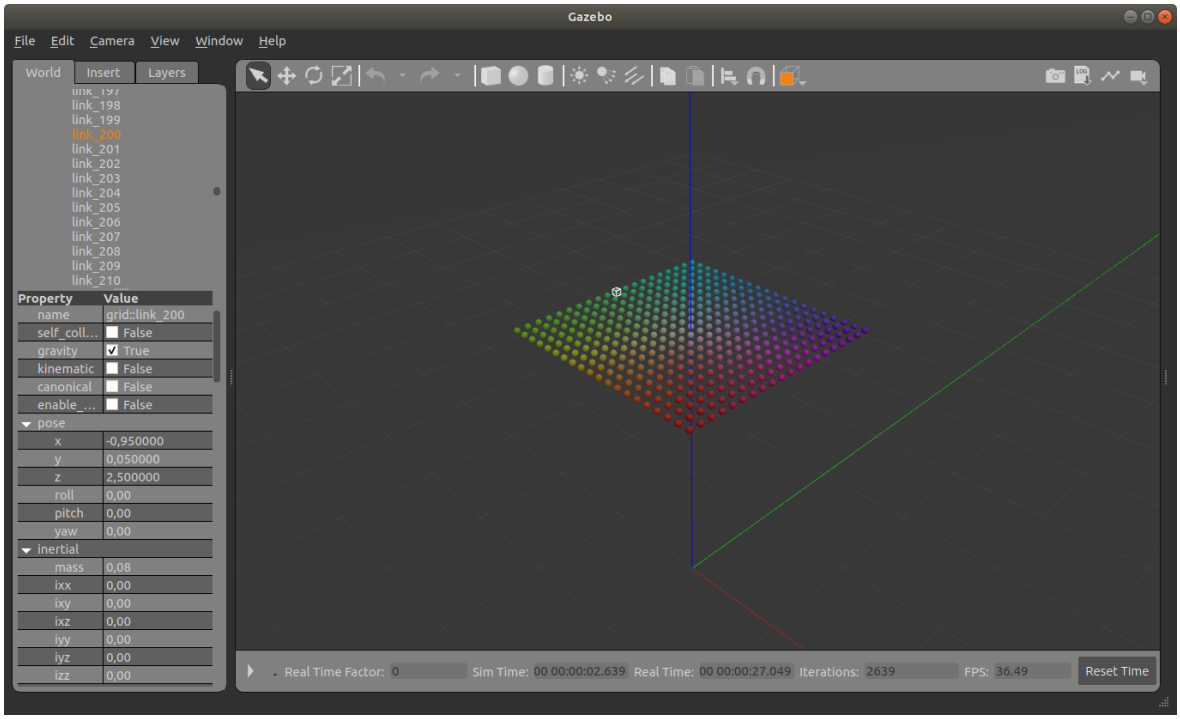


Figura B.8: Selección manual de la esfera 200 en Gazebo.

Con el panel inferior izquierdo, podremos modificar la posición de la esfera y (con la simulación no pausada) observar el comportamiento del conjunto.

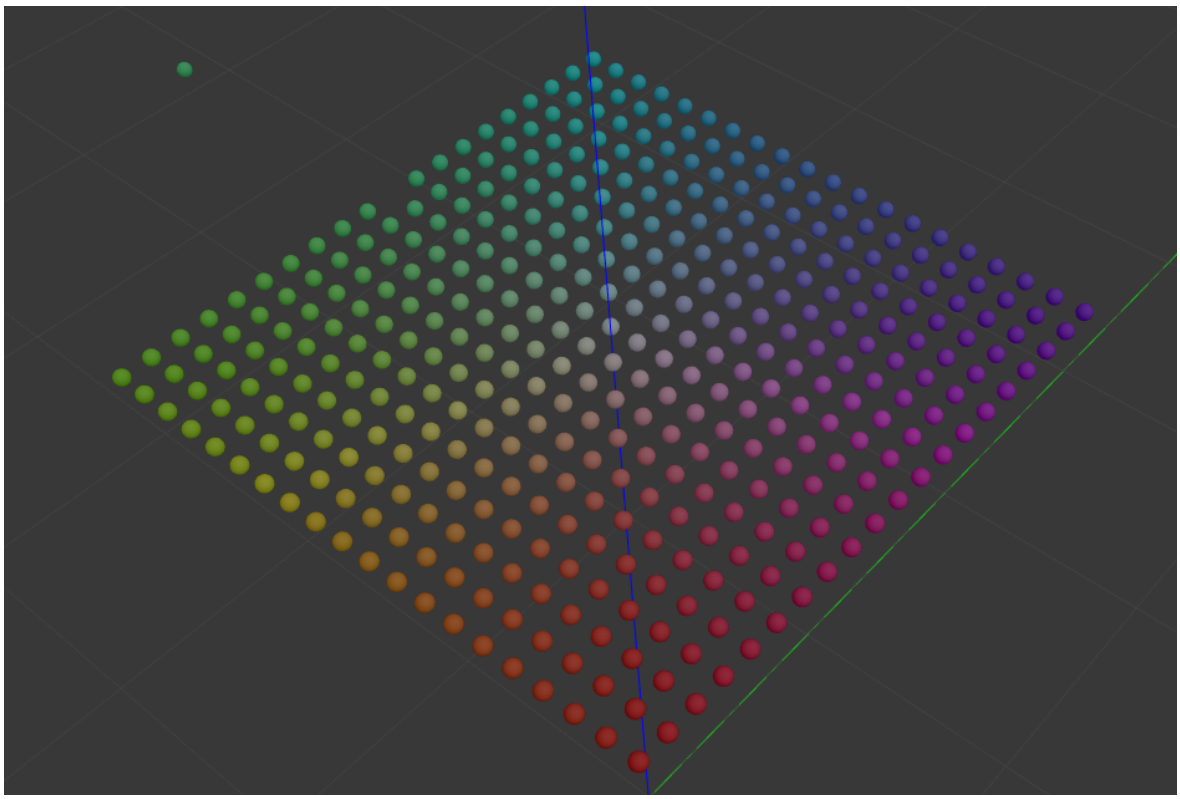


Figura B.9: Esfera 200 con una posición seleccionada manualmente en Gazebo.

B.3.7. *small_grid_manipulation*

Para ejecutar esta prueba, deberá utilizar **4 terminales**. En la primera, deberá introducir el siguiente comando: `./src/multiple_abb_irb120/launch/multiple_spawner_gazebo_script.bash 2 -s` La opción `-s` hará que la simulación cree un objeto con los parámetros en el fichero *small_grid.config*. Recuerde que es muy importante que haya al menos 2 robots para el funcionamiento correcto. Una vez inicializada la simulación en Gazebo, deberá observar una ventana como la de la Figura B.10, en la segunda y tercera terminales deberá lanzar MoveIt para los dos robots: `roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \ robot_name:=robot1`

`roslaunch multiple_abb_irb120 moveit_planning_execution_gazebo.launch \ robot_name:=robot2`

Finalmente, cuando las dos ventanas de RViz se hayan iniciado, deberá inicializar la prueba, en la cuarta terminal, con el siguiente comando: `rosrun multiple_abb_irb120 small_cloth_manipulation`
Ya puede observar la ventana de Gazebo para ver la simulación.

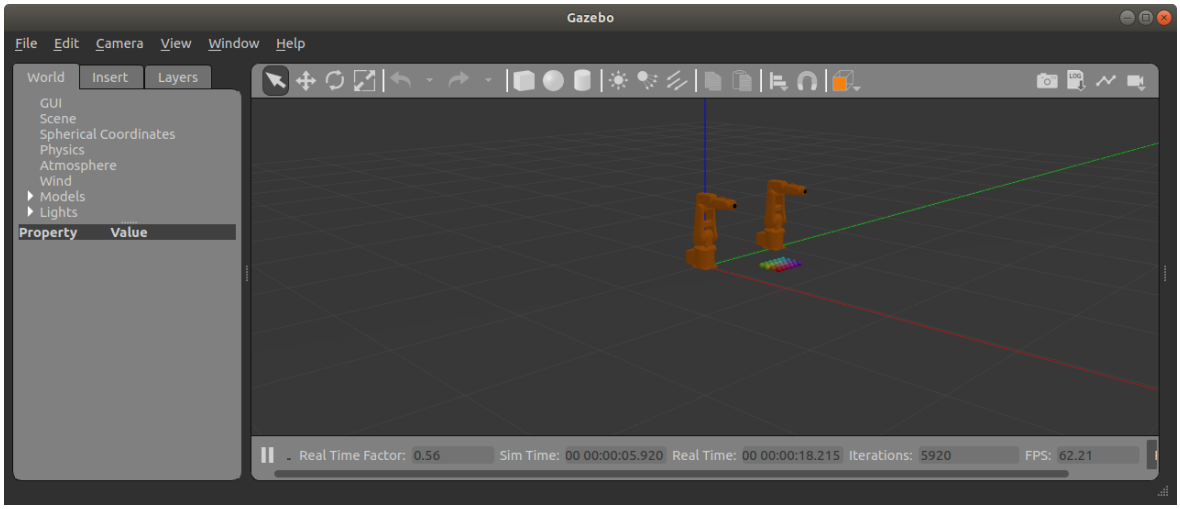


Figura B.10: Resultado de la ejecución de *multiple_spawner_gazebo_script.bash* con dos robots y con la opción `-s`.

B.4. Realización de pruebas con una mesa

Como se indica en la Sección 7.4, se ha incluido un archivo para realizar pruebas con una mesa por comodidad. Para ejecutarlo, deberemos introducir el siguiente comando en una terminal:

```
./src/multiple_abb_irb120/launch/multiple_spawner_gazebo_script.bash 2 -t
```

y veremos así una simulación con dos robots y con una mesa sobre la que la tela descansará (la mesa tiene unas medidas de $(0{,}5, 0{,}75, 0{,}5)$ aproximadamente), como se observa en la Figura B.11.

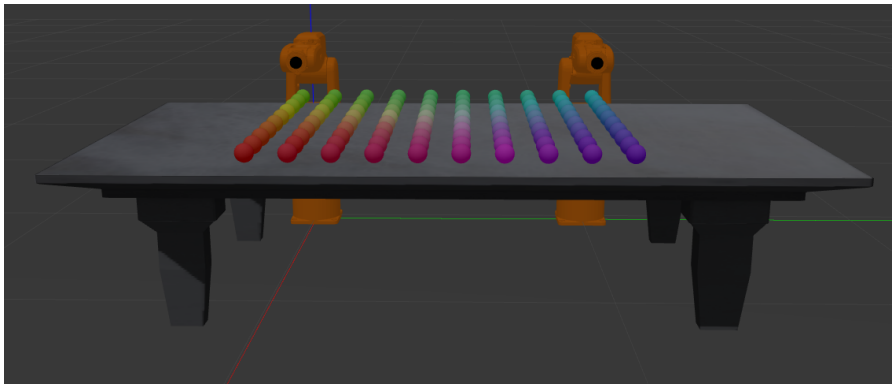


Figura B.11: Simulación con el mundo *grid_table.world*.

Cabe destacar, que si se desean utilizar mundos personalizados, se puede modificar el fichero *setup_gazebo.launch* y sustituir el nombre del mundo incluido al inicializarlo por el de *grid_table.world*. Si queremos volver a la normalidad, podremos eliminarla a través de la interfaz de Gazebo, o cambiar el nombre a *grid.world* de nuevo. Esto nos permite modificar el mundo a nuestro gusto, pero es importante que se mantenga la

cláusula que declara el *plugin* del mundo para poder generar el objeto deformable.

```
<launch>
  <arg name="must_start_world" default="true"/>
    <arg name="robots" default="2" doc="Number of robots to spawn" />

  <!-- IMPORTANT: topics must be remapped before using this launchfile
  <remap from="/$(arg base_name)$(arg robots)/arm_controller/follow_joint_trajectory" to="/$(arg base_name)$
  <remap from="/$(arg base_name)$(arg robots)/arm_controller/state" to="/$(arg base_name)$(arg robots)/feedb
  <remap from="/$(arg base_name)$(arg robots)/arm_controller/command" to="/$(arg base_name)$(arg robots)/joi
  -->

  <include file="$(find multiple_abb_irb120)/launch/recursive_spawn.launch" if="$(eval arg('robots') >= 1)">
    <arg name ="number" value="$(arg robots)"/>
  </include>

  <!-- startup simulated world -->
  <include if="$(arg must_start_world)" file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find multiple_abb_irb120)/worlds/grid.world"/>
    <arg name="gui" value="true"/>
  </include>

</launch>
```

Figura B.12: Cómo modificar el mundo de la simulación.
Subrayado en color naranja se encuentra el fragmento de código a cambiar si se desea cambiar el mundo de la simulación.

Anexos C

Algunos problemas encontrados

A lo largo del desarrollo de este Trabajo, ha habido muchos intentos de completar ciertas tareas, pero que se han tenido que descartar por los problemas que ha causado su implementación, o se han tenido que encontrar alternativas a su uso. En las siguientes secciones se detallan algunas de ellas.

C.1. Espacios de nombres

En varios momentos en la creación de los ficheros de configuración para el entorno multi-robot, se ha precisado de parámetros para especificar el nombre del robot, y así introducir todos los nodos y parámetros del servidor de ROS en el espacio de nombres del robot.

Para conseguir esto, es muy util la cláusula *<group>* del lenguaje XML de ROS [95], pero también puede hacerse manualmente modificando los nombres o añadiendo el parámetro *ns* en la declaración de los nodos o parámetros.

C.1.1. Problemas en los archivos de configuración de Gazebo

Al crear un entorno en Gazebo con varios robots ABB IRB120, la primera solución que se probó, fue la utilización de nombres únicos para cada uno de los nodos implicados en el control o creación de cada robot. Así, se introdujeron todos los elementos de los archivos en una cláusula *<group>*, que introduce los parámetros, nodos y servicios en un namespace con el nombre especificado [95]. A pesar de ello, algunos de los nodos daban errores de conflicto por lo que se optó por cambiarles manualmente el nombre para evitar dicho error. Sin embargo, esta solución inicializaba Gazebo sin ninguno de los robots presentes, por lo que se descartó.

También se realizaron pruebas cambiando los nombres de todas las articulaciones para que fueran únicas. Posteriormente se descubrió la opción de utilizar *tf_prefix*, un parámetro bajo cuyo namespace se colocarán todas las articulaciones y enlaces del

robot, facilitando mucho la tarea a realizar [96].

Por otro lado, para la correcta introducción de algunos topics en los espacios de nombres, fueron necesarios varias cláusulas del tipo *<remap>* (que se encargan de redirigir un topic a otro nombre distinto [93]), algunos de ellos ya presentes en los archivos proporcionados por ABB [94], para que los nombres coincidiesen. Sin embargo, tras varios intentos de depuración, se comprobó que estas redirecciones no se estaban produciendo si las cláusulas *<remap>* se encontraban en un fichero *launch* invocado desde otro, dejando sólo la opción de que estas se encontraran en el fichero invocado a través del comando *roslaunch*. No se ha llegado a ninguna conclusión sobre cuál pudiera ser la razón. Por esto, y para mantener el esquema recursivo, fue necesaria la creación de un fichero de ***bash*** cuya única función es crear un fichero temporal que contenga las cláusulas *<remap>* para cada robot (dándoles un nombre de $robot_i$, siendo i un número natural desde 1 hasta el número total de robots), e incluyendo el archivo *launch* ya existente. Si bien no es el mejor método, supone una solución temporal hasta que se descubran las causas de las redirecciones erróneas. También se incluye un fichero invocable por medio de *roslaunch* para la creación de sólo dos robots, sin que sea necesario el uso del fichero de bash.

C.1.2. Problemas en los archivos de configuración de MoveIt

A la hora de adecuar la configuración de MoveIt para que coincidiese con la de Gazebo, se introdujeron todos los elementos de los archivos en una cláusula *<group>*. Sin embargo, varios de los nodos daban errores de conflicto por lo que se optó por cambiarles manualmente el nombre para evitar dicho error.

Otro problema que se presentó durante las primeras ejecuciones fue que tanto RViz como la interfaz de MoveIt en C++ funcionaban correctamente para el primer robot, pero no para cualquier otro que se añadiese. En primer lugar, RViz no era capaz de enviar posiciones de objetivo a los robots a partir del segundo de ellos, mientras que la interfaz de C++ devolvía siempre la posición del primer robot al invocar funciones como *getCurrentPose* desde cualquier robot. El primero se solucionó redirigiendo el *topic /joint_states* a $/robot_i/joint\_states$ en el nodo *move_group* de un robot cuyo nombre es $robot_i$ (los nombres de los robots serán *robot1*, *robot2*...), tal y como indican los tutoriales de MoveIt [102].

El segundo fue más complicado, pues no existen tutoriales o documentación sobre este tema, por lo que fue necesario observar el código fuente de MoveIt. Se pudo observar, que la clase *MoveGroupInterface* hace uso de un *CurrentStateMonitor* para obtener la posición del robot [103] que a su vez hace uso del *topic* "*joint_states*" [104]. La creación del *CurrentStateMonitor* requiere del nombre del modelo del robot que se

va a monitorizar, y también un objeto del tipo *NodeHandle*, entre otros. Es importante ver que la interfaz *MoveGroupInterface* hace uso de la función *getSharedStateMonitor* [105] para crear el objeto, y se observa que esta función inserta en un mapa estos monitores como valor, y el nombre del modelo del robot como clave [106]. Por lo tanto, si se inserta una entrada en el mapa con una clave repetida, la función devolverá el monitor que ya se encuentra en el mapa con la misma clave y no se creará el nuevo [106] [107]. Es por ello que será necesario que **los nombres de los modelos de los robots sean siempre únicos** en los archivos de definición de *URDF* y *SRDF*, así se cambiaron los nombres en estos archivos (por medio de parámetros) para que fueran $abb\_irb120\_3\_58\_robot_i$, siendo $robot_i$ el nombre del robot, como se observa en la Figura C.1.

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

<robot name="abb_irb120_3_58_$(arg robotns)" xmlns:xacro="http://ros.org/wiki/xacro">

  <xacro:arg name="prefix" default="" />
  <xacro:arg name="robotns" default="robot1" />

  <xacro:include filename="$(find multiple_abb_irb120)/config/abb_irb120_3_58_macro.srdf.xacro"/>
  <xacro:abb_irb120_3_58_g prefix="$(arg prefix)" robotns="$(arg robotns)"/>
</robot>
```

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

<robot name="abb_irb120_3_58_$(arg robotns)" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:arg name="prefix" default="" />
  <xacro:arg name="robotns" default="robot1" />
  <xacro:include filename="$(find multiple_abb_irb120)/urdf/irb120_3_58_macro.xacro"/>
  <xacro:abb_irb120_3_58_g prefix="$(arg prefix)" robotns="/$(arg robotns)"/>
```

Figura C.1: Nombres de los modelos de los robots en los ficheros *abb_irb120_3_58.srdf.xacro* y *irb120_3_58_with_tool.xacro*

C.1.3. Utilización de parámetros en ficheros *yaml*

Los ficheros *yaml* no poseen ningún mecanismo para la inclusión de parámetros. Por tanto, se investigó y se encontró que existe la opción *subst_value=True* en la cláusula *rosparam* del lenguaje XML de ROS.

Por ello, se modificaron los archivos para que contuvieran controladores y articulaciones con nombres únicos, especialmente el fichero *irb120_3_58_arm_controller.yaml*, que contiene la definición del controlador y de sus articulaciones [86]. También se tuvo que modificar los archivos *urdf* para indicar el namespace del robot. Esto permitió una ejecución de MoveIt y RViz para cada robot, pero con la cual los robots no se movían y que hizo surgir varios mensajes de aviso, que alertaban de conflictos o ajustes que hacían que Gazebo y MoveIt no estuvieran de acuerdo en los nombres de los nodos y parámetros.

Al colocar un nombre de variable (indicadas con el carácter $) en el fichero *yaml*, si existe un parámetro con el mismo nombre en el fichero *launch* que lo carga, ROS se encargará de sustituir la variable en el *yaml* por el valor que tuviera el parámetro en el *launch*. En la Figura C.2 se observa un ejemplo, en el que el archivo *planning_context.launch* cargará el contenido del *kinematics.yaml* sustituyendo "*$(arg robot_name)*" por el valor del parámetro *robot_name* en el momento de la ejecución.

```
<launch>
  <!-- By default we do not overwrite the URDF. Change the following to true to change the default behavior -->
  <arg name="load_robot_description" default="false"/>

  <!-- The name of the parameter under which the URDF is loaded -->
  <arg name="robot_description" default="robot_description"/>

  <arg name="robot_name" default="robot1"/>
[...]
  <!-- Load default settings for kinematics; these settings are overridden by settings in a node's namespace -->
  <group ns="/$(arg robot_name)/$(arg robot_description)_kinematics">
    <rosparam command="load" file="$(find multiple_abb_irb120)/config/kinematics.yaml" subst_value="true"/>
  </group>
```

```
$(arg robot_name)_manipulator:
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
  kinematics_solver_search_resolution: 0.005
  kinematics_solver_timeout: 0.005
  kinematics_solver_attempts: 3
```

Figura C.2: Empleo del argumento *subst_value*. Los ficheros *yaml* tomarán los parámetros del fichero *launch* que lo cargue, en este caso, en la parte de arriba, el fichero *planning_context.launch* define el parámetro *robot_name* y carga el fichero *kinematics.yaml*, que hace uso del argumento *robot_name* en la parte inferior de la imagen.

C.2. API de Gazebo

Cuando se intentaba implementar la creación del objeto deformable, se intentó no crear un nuevo modelo para cada esfera, como ocurría en la Sección 4.1. La primera solución fue crear un modelo con un plugin del tipo *ModelPlugin* que sería el encargado de modificar el modelo para añadir todos los enlaces que forman la cuadrícula y posteriormente controlarla. Para esto se utilizaron las funciones de la clase ***Model*** en el namespace *gazebo::physics* [78]. Sin embargo, fue un intento fallido, pues el modelo no llegó a modificarse nunca sin ningún mensage de error, y puesto que la documentación de la API de Gazebo es más bien escasa (todas las funciones en la página oficial poseen exclusivamente una línea para explicar su objetivo y sus parámetros [78]), se buscó otra alternativa.

Anexos D

Fragmentos de código interesantes

A lo largo de esta memoria, se mencionan algunos de los ficheros desarrollados (recuerde que puede observar la estructura de ficheros en el Apéndice A). Se incluyen a continuación algunos de ellos. De todas formas, se recomienda visitar el repositorio de GitHub de este Trabajo si se desea tener una perspectiva mejor y actualizada de estos u otros ficheros [101].

D.1. *grid.config*

Este fichero posee los parámetros del tamaño de la tela en metros (*width*, *height*); el número de filas (*vertical_resolution*) y columnas (*horizontal_resolution*) de esferas; la posición del centro del objeto deformable *(offset_x, offset_y, offset_z)*; el radio de las esferas en metros (*sphere_radius*); la masa de cada esfera en kilogramos (*mass*) y las constantes de elasticidad (*stiffness*) y de amortiguamiento (*damping*) y si se desea que el objeto esté afectado por la gravedad, cuando el valor es 1, o no, cuando el valor es 0 (*gravity*).

```
width:1
height:1
vertical_resolution:10
horizontal_resolution:10
offset_x:0.5
offset_y:0.75
offset_z:0.5
sphere_radius:0.025
mass:0.08
stiffness:100
damping:10
gravity:0
```

Este archivo hará que se simule una tela de 1x1 metros, con 10x10 esferas de 25 cm de radio y una masa de 80 mg, alrededor de la posición (0.5, 0.75, 0.5), a la que no le afectará la gravedad y cuyas constantes de elasticidad y amortiguamiento serán 100 N/m y 10 N s/m, respectivamente.

D.2. *small_grid.config*

Este fichero posee la misma estructura que el fichero anterior *grid.config* y es utilizado para realizar algunas de las pruebas en este Trabajo.

```
width:0.2
height:0.3
vertical_resolution:6
horizontal_resolution:4
offset_x:0.5
offset_y:0.5
offset_z:0.5
sphere_radius:0.025
mass:0.04
stiffness:50
damping:3
gravity:1
```

Este archivo hará que se simule una tela de 20x30 centímetros, con 4x6 esferas de 25 cm de radio y una masa de 40 mg, alrededor de la posición (0.5, 0.5, 0.5), a la que le afectará la gravedad y cuyas constantes de elasticidad y amortiguamiento serán 50 N/m y 3 N s/m, respectivamente.

D.3. *GrabPetition.msg*

Este fichero define el mensaje utilizado por las diversas partes de la simulación para solicitar el "agarre" de una esfera.

```
int32 i
int32 j
bool grab
std_msgs/String robot_name
std_msgs/String link_name
```

D.4. *recursive_spawn.launch*

Se encarga de llamar al fichero *spawn_irb120.launch* que carga un robot IRB120 en la simulación, pero también a sí mismo, siempre y cuando el parámetro del número de robots sea mayor que 0. Este número se reduce en cada iteración, consiguiendo así un comportamiento recursivo.

```
<launch>
    <arg name="number" default="0"/>

  <include file="$(find multiple_abb_irb120)/launch/spawn_irb120.launch"
  if="$(eval arg('number') >= 1)">
    <arg name="robot_name" value="robot$(arg number)"/>
    <arg name="position_x" value="$(eval int(arg('number')) - 1)"/>
  </include>

```

```
  <include
  file="$(find multiple_abb_irb120)/launch/recursive_spawn.launch"
  if="$(eval arg('number') >= 1)">
    <arg name="number" value="$(eval int(arg('number')) - 1)"/>
  </include>
</launch>
```

D.5. *spawn_irb120.launch*

Se encarga de cargar un robot cuyo nombre será el que se introduzca por el parámetro *robot_name* y en la posición $position\_x, position\_y, position\_z$ en la simulación de Gazebo.

```
<launch>

  <arg name="robot_name" default="robot1"/>
  <arg name="position_x" default="0"/>
  <arg name="position_y" default="0"/>
  <arg name="position_z" default="0"/>

  <!-- setup tf_prefix-->
  <group ns="$(arg robot_name)">
  <param name="tf_prefix" value="$(arg robot_name)"/>
  </group>

  <!-- since the tf_prefix will change the name of the "world" frame,
  we need to publish the robot's world to "world" -->
  <node pkg="tf" type="static_transform_publisher"
    name="$(arg robot_name)_world_publisher"
    args="0 0 0 0 0 0 world $(arg robot_name)/world 100"/>

  <!-- urdf xml robot description loaded on the Parameter Server,
  converting the xacro into a proper urdf file-->
  <param name="/$(arg robot_name)/robot_description"
     command="$(find xacro)/xacro --inorder
     '$(find multiple_abb_irb120)/urdf/irb120_3_58_with_tool.xacro'
     robotns:='/$(arg robot_name)'" />

  <!-- push robot_description to factory and spawn robot in gazebo -->
  <node name="abb_irb120_$(arg robot_name)_spawn" pkg="gazebo_ros"
    type="spawn_model" output="screen"
    args="-urdf
        -param /$(arg robot_name)/robot_description
        -robot_namespace /$(arg robot_name)
        -model abb_irb120_3_58_$(arg robot_name)
        -x $(arg position_x)
        -y $(arg position_y)
        -z $(arg position_z)"
    ns="$(arg robot_name)"/>

  <!-- convert joint states to TF transforms for rviz, etc -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
  type="robot_state_publisher" output="screen" ns="$(arg robot_name)"/>

  <!-- init and start Gazebo ros_control interface -->
```

```
  <include
  file="$(find multiple_abb_irb120)/launch/irb120_3_58_control.launch">
  <arg name="robot_name" value="$(arg robot_name)"/>
  </include>

</launch>
```

D.6. *multiple_spawner_gazebo_script.bash*

Este fichero es un *script* de *bash*. Toma un argumento por línea de comandos (variable *robots*) y se utiliza para crear un fichero *launch* temporal en un directorio temporal con directivas *remap* y que incluirá el fichero *setup_gazebo.launch* con el número de robots introducido por la línea de comandos. Finalmente, lanza el fichero mediante *roslaunch* y una vez finalizado lo borra. Posee tres opciones:

- `-d` : Utilizada para lanzar la prueba *test_grid*. En este caso, el número de robots es ignorado.
- `-s` : Utilizada para lanzar un programa con una tela con los parámetros de *small_grid.cofig*.
- `-t` : Utilizada para cargar el modelo de una mesa en la simulación.

Si no se introduce el parámetro del número de robots, muestra un mensaje de aviso y usará un valor de 2.

```
#!/bin/bash
robots="$1"
option="$2"
dir="$(cd "$(dirname "$0")" && pwd)"
reg_ex="^[0-9]+$"
testing=0
small=0
table=0

if [[ $robots =~ $reg_ex ]] ; then
  echo "Spawning $robots robots..."
else
  echo -e "\033[33mWARNING: You have not introduced a valid number.
  Defaulting to 2...\033[0m"
  robots=2
  echo "Usage: $(basename $0) <number of robots> [-d | -s | -t]"
    echo "       -d : debug grid physics"
    echo "       -s : small grid"
    echo "       -t : use a table"
fi

if [ "$#" -gt "1" ] ; then
  if [ "$option" = "-t" ]; then
    testing=1
    echo "Testing... Robots will not be spawned, and a big grid will
```

```
    spawn."
  elif [ "$option" = "-s" ]; then
    small=1
    echo "Small grid applied."
  else
    echo -e "\033[33mWARNING: You have introduced an unknown option
    \"$option\". Ignored.\033[0m"
    echo "Usage: $(basename $0) <number of robots> [-d | -s | -t]"
    echo "       -d : debug grid physics"
    echo "       -s : small grid"
    echo "       -t : use a table"
  fi
fi

if [ "$#" -gt "3" ]; then
  echo -e "\033[33mWARNING: You have introduced too many options
  (more than 2).\033[0m"
  echo "Usage: $(basename $0) <number of robots> [-d | -s | -t]"
  echo "       -d : debug grid physics"
  echo "       -s : small grid"
  echo "     -t : use a table"
fi

if [ "$testing" -eq "1" ]; then
  roslaunch "$dir/text_grid.launch"
else

  source "$dir/../../../devel/setup.bash"
  temp_dir=$(mktemp
    -d "${TMPDIR:-/tmp/}multiple_robots_ros_package.XXXXXXXXXXXX")
  temp_file=$(mktemp
    "--tmpdir=$temp_dir" "gazebo_remapper_file_XXXXXX.launch")
      "gazebo_remapper_file_XXXXXX.launch")
  i=0

  echo "<launch>" > $temp_file

  while [ $i -lt $robots ]
  do
    i=$((i+1))
    echo "<remap from=\"/robot$i/arm_controller/follow_joint_trajectory\" \
             to=\"/robot$i/joint_trajectory_action\" />" \
             >> $temp_file
    echo "<remap from=\"/robot$i/arm_controller/state\" \
             to=\"/robot$i/feedback_states\" />" >> $temp_file
    echo "<remap from=\"/robot$i/arm_controller/command\" \
             to=\"/robot$i/joint_path_command\"/>" >> $temp_file
  done

  echo "<include \
       file=\"\$(find multiple_abb_irb120)/launch/setup_gazebo.launch\">" \
       >> $temp_file
  echo "<arg name=\"robots\" value=\"$robots\"/>" >> $temp_file \
  echo "<arg name=\"small\" value=\"$small\"/>" >> $temp_file
  echo "<arg name=\"table\" value=\"$table\"/>" >> $temp_file
  echo "</include>" >> $temp_file

  echo "</launch>" >> $temp_file
```

```
84
85    roslaunch $temp_file
86    rm $temp_file
87    rm -r $temp_dir
```

D.7. *setup_gazebo.launch*

Este fichero se encarga de llamar al fichero *recursive_spawn.launch* para que se cargen los robots en la simulación y si su parámetro *must_start_world* tiene valor verdadero, carga el mundo de Gazebo, que en este caso es *grid.world*.

```
1  <launch>
2    <arg name="must_start_world" default="true"/>
3    <arg name="robots" default="2" doc="Number of robots to spawn" />
4
5    <!-- IMPORTANT: topics must be remapped before using this launchfile
6    <remap from="/$(arg base_name)$(arg robots)/arm_controller/
7                 follow_joint_trajectory"
8             to="/$(arg base_name)$(arg robots)/joint_trajectory_action" />
9    <remap from="/$(arg base_name)$(arg robots)/arm_controller/state"
10            to="/$(arg base_name)$(arg robots)/feedback_states" />
11   <remap from="/$(arg base_name)$(arg robots)/arm_controller/command"
12            to="/$(arg base_name)$(arg robots)/joint_path_command"/>
13   -->
14
15   <include if="$(eval arg('robots') >= 1)">
16          file="$(find multiple_abb_irb120)/launch/recursive_spawn.launch"
17
18     <arg name ="number" value="$(arg robots)"/>
19   </include>
20
21   <!-- startup simulated world -->
22   <include if="$(arg must_start_world)"
23          file="$(find gazebo_ros)/launch/empty_world.launch">
24     <arg unless="$(eval arg('small') or arg('table'))"
25          name="world_name"
26          value="$(find multiple_abb_irb120)/worlds/grid.world"/>
27     <arg if="$(eval arg('small') and not arg('table'))"
28          name="world_name"
29          value="$(find multiple_abb_irb120)/worlds/small_grid.world"/>
30     <arg if="$(arg table)"
31          name="world_name"
32          value="$(find multiple_abb_irb120)/worlds/grid_table.world"/>
33     <arg name="gui" value="true"/>
34   </include>
35
36 </launch>
```

D.8. *moveit_planning_execution_gazebo.launch*

Este fichero se encarga de inicializar los nodos y parámetros para ejecutar MoveIt y RViz con un robot indicado por su parámetro *robot_name* que deberá ser modificado

según el robot que se desea controlar.

```
<launch>
  <!-- The planning and execution components of MoveIt! configured to run
       against a Gazebo based, ros_control compatible simulation of the
       IRB 120. This depends on the corresponding 'abb_irb120_gazebo' pkg
       to be installed first. This dependency is not expressed in the
       MoveIt config pkg manifest, as adding a run_depend there would
       cause Gazebo to be unconditionally installed, even if the user
       never intends to use the MoveIt config with it.
       Instead, installation is left to the user, as a kind of poor-mans
       optional dependency.
       Finally, this launch file assumes that gazebo is already running
       and that the IRB 120 and ros_controllers are loaded.
  -->


  <!-- By default, we do not start a database (it can be large) -->
  <arg name="db" default="false" />
  <!-- Allow user to specify database location -->
  <arg name="db_path"
 default="$(find abb_irb120_moveit_config)/default_warehouse_mongo_db"/>

  <arg name="robot_name" default="robot1"/>

  <remap from="/joint_trajectory_action"
           to="/$(arg robot_name)/joint_trajectory_action"/>

  <group ns="$(arg robot_name)">
    <rosparam command="load"
  file="$(find abb_irb120_support)/config/joint_names_irb120_3_58.yaml"/>

    <include
     file="$(find multiple_abb_irb120)/launch/planning_context.launch">
        <arg name="load_robot_description" value="false" />
        <arg name="robot_name" value="$(arg robot_name)"/>
      </include>

      <include
       file="$(find multiple_abb_irb120)/launch/move_group.launch">
        <arg name="publish_monitored_planning_scene" value="true" />
        <arg name="robot_name" value="$(arg robot_name)"/>
      </include>

      <include
       file="$(find multiple_abb_irb120)/launch/moveit_rviz.launch">
        <arg name="config" value="true"/>
        <arg name="robot_name" value="$(arg robot_name)"/>
      </include>
  </group>

  <!-- If database loading was enabled, start mongodb as well -->
  <include file="$(find
  abb_irb120_moveit_config)/launch/default_warehouse_db.launch"
   if="$(arg db)">
    <arg name="moveit_warehouse_database_path" value="$(arg db_path)"/>
  </include>
</launch>
```

D.9. *abb_irb120_3_58.srdf.xacro*

Este fichero posee la información semántica del robot ABB IRB120, en un formato xacro, que permite el uso de parámetros para definir el nombre del robot. Este utiliza una *macro*, que importa desde el fichero *abb_irb120_3_58.srdf_macro.xacro*.

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

<robot name="abb_irb120_3_58" xmlns:xacro="http://ros.org/wiki/xacro">

  <xacro:arg name="prefix" default="robot1_" />
  <xacro:arg name="robotns" default="robot1" />

  <xacro:include filename="$(find
  multiple_abb_irb120)/config/abb_irb120_3_58_macro.srdf.xacro"/>
  <xacro:abb_irb120_3_58_g prefix="$(arg prefix)"
                           robotns="$(arg robotns)"/>
</robot>
```

D.10. *abb_irb120_3_58.srdf_macro.xacro*

Este fichero posee una macro que contiene la información semántica del robot ABB IRB120, en un formato xacro, que permite el use de parámetros para definir el nombre del robot. Se pueden observar aquí el nombre del grupo *${robotns}_manipulator*, y también las dos poses predefinidas de los robots *all_zero* y *demo_pose*.

```
<?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 xmlns:xacro="http://ros.org/wiki/xacro">
<xacro:macro name="abb_irb120_3_58_g" params="prefix robotns">
    <!--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="${robotns}_manipulator">
        <chain base_link="base_link" tip_link="tool_link" />
    </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="all_zero" group="${robotns}_manipulator">
        <joint name="joint_1" value="0" />
        <joint name="joint_2" value="0" />
        <joint name="joint_3" value="0" />
        <joint name="joint_4" value="0" />
        <joint name="joint_5" value="0" />
        <joint name="joint_6" value="0" />
    </group_state>

    <group_state name="demo_pose" group="${robotns}_manipulator">
        <joint name="joint_1" value="0" />
        <joint name="joint_2" value="0.21" />
        <joint name="joint_3" value="-0.1" />
        <joint name="joint_4" value="0" />
        <joint name="joint_5" value="1.466" />
        <joint name="joint_6" value="0" />
    </group_state>
    <!--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="FixedBase" type="fixed" 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="link_1" reason="Adjacent"/>
    <disable_collisions link1="base_link" link2="link_2" reason="Never"/>
    <disable_collisions link1="link_1" link2="link_2" reason="Adjacent"/>
    <disable_collisions link1="link_2" link2="link_3" reason="Adjacent"/>
    <disable_collisions link1="link_2" link2="link_5" reason="Never"/>
    <disable_collisions link1="link_2" link2="link_6" reason="Never"/>
    <disable_collisions link1="link_3" link2="link_4" reason="Adjacent"/>
    <disable_collisions link1="link_3" link2="link_5" reason="Never"/>
    <disable_collisions link1="link_3" link2="link_6" reason="Never"/>
    <disable_collisions link1="link_4" link2="link_5" reason="Adjacent"/>
    <disable_collisions link1="link_4" link2="link_6" reason="Default"/>
    <disable_collisions link1="link_5" link2="link_6" reason="Adjacent"/>
  </xacro:macro>
</robot>
```

D.11. *small_cloth_manipulation.cpp*

Este fichero contiene el comportamiento de la simulación final. Se crean las clases para controlar los robots y conocer el estado de las esferas, y se ordenan movimientos a los robots para que cojan y agiten la tela, la estiren y la dejen en el suelo.

```
/*********************************************************************
 * Software License Agreement (BSD License)
 *
 *  Copyright (c) 2013, SRI International
 *  All rights reserved.
 *
```

```
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of SRI International nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *********************************************************************/

/* Author: Sachin Chitta, Dave Coleman, Mike Lautman */
// MODIFIED BY: Andrés Otero García, Gonzalo López Nicolás and María del
// Rosario Aragüés Muñoz


#include <moveit/robot_model_loader/robot_model_loader.h>
#include <moveit/planning_interface/planning_interface.h>
#include <moveit/planning_scene/planning_scene.h>
#include <moveit/planning_scene_monitor/planning_scene_monitor.h>
#include <moveit/kinematic_constraints/utils.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/PlanningScene.h>

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

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

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

#include "gazebo/common/common.hh"
#include "gazebo/gazebo.hh"

#include <iostream>     // std::streambuf, std::cout
#include <fstream>      // std::ofstream

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

```
65
66 #include "RobotInterface.hpp"
67 #include "GridState.hpp"
68 #include <chrono>
69
70 #include <signal.h>
71 #include <chrono>
72
73 #include "utils.hpp"
74
75 #include <multiple_abb_irb120/GrabPetition.h>
76
77 #include <ros/console.h>
78 #include <tf2/LinearMath/Quaternion.h>
79
80 #include "Demo.hpp"
81
82
83 const int NUM_ROBOTS = 2;
84
85 using MGIPtr =
86     std::shared_ptr <moveit::planning_interface::MoveGroupInterface>;
87 using GridStatePtr = GridState*;
88
89 class SmallClothDemo : public Demo {
90
91 private:
92
93   ros::NodeHandle n;
94   ros::Publisher grabPub;
95
96   virtual void doDemo(RobotInterface& robot, int robot_i, int params, ...)
97   override {
98     if(params != 3){
99       ROS_ERROR("Wrong number of parameters for grid demo");
100      return;
101    }
102    else{
103      va_list args;
104      va_start(args, params);
105
106      GridStatePtr gridState = va_arg(args, GridStatePtr);
107      int sphere_i = va_arg(args, int);
108      int sphere_j = va_arg(args, int);
109
110      tf2::Quaternion facing_down;
111      multiple_abb_irb120::GrabPetition grabMsg;
112      std_msgs::String robot_name, link_name;
113      std::vector<geometry_msgs::Pose> waypoints(1);
114      geometry_msgs::Pose initial, target, sphere_initial;
115      moveit_msgs::RobotTrajectory trajectory;
116      moveit::planning_interface::MoveGroupInterface::Plan my_plan;
117      MGIPtr move_group = robot.getMoveGroup();
118
119      facing_down.setRPY(0, M_PI, 0);
120      move_group->setPlanningTime(10.0);
121
122      initial = move_group->getCurrentPose().pose;
```

```
123       sphere_initial =
124       utils::getAdjustedSpherePose(gridState->getPose(sphere_i, sphere_j),
125       robot.getBasePosition(), facing_down);
126
127       std::cout << "Robot " << robot_i + 1 << ": " <<
128       "Approaching sphere " << sphere_i << " " << sphere_j << "..."<<
129       std::endl;
130
131       ////////////////////////////////////
132       // Approach sphere
133       ////////////////////////////////////
134       if(ros::ok()){
135         do {
136           target =
137             utils::getAdjustedSpherePose(gridState->getPose(sphere_i,
138             sphere_j),
139             robot.getBasePosition(), facing_down);
140
141           move_group->setPoseTarget(target);
142           move_group->move();
143         } while(ros::ok() &&
144                !utils::isNear(move_group->getCurrentPose().pose,
145                utils::getAdjustedSpherePose(gridState->getPose(sphere_i,
146                sphere_j), robot.getBasePosition(), facing_down), 0.03));
147       }
148
149       move_group->stop();
150       std::cout << "Robot " << robot_i + 1 << ": " <<
151       "Approached sphere " << sphere_i << " " << sphere_j << "." <<
152       std::endl;
153
154       ////////////////////////////////////
155       // Grab sphere
156       ////////////////////////////////////
157       link_name.data = LINK_NAME;
158       robot_name.data = ROBOT_PREFIX + std::to_string(robot_i + 1);
159       grabMsg.robot_name = robot_name;
160
161       grabMsg.i = sphere_i;
162       grabMsg.j = sphere_j;
163       grabMsg.link_name = link_name;
164       grabMsg.robot_name = robot_name;
165       grabMsg.grab = true;
166
167       // SYNC
168       if(!syncRobots(robot_i, 1)) {
169           va_end(args);
170           return;
171         }
172
173       grabPub.publish(grabMsg);
174       gridState->setGrabbed(robot_i, true);
175
176       std::cout << "Robot " << robot_i + 1 << ": " <<
177       "Grabbed sphere " << sphere_i << " " << sphere_j << "." <<
178       std::endl;
179
180       std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

```
181
182       ////////////////////////////////////
183       // Go up
184       ////////////////////////////////////
185       std::cout << "Robot " << robot_i << ": " <<
186       "Going up..." << std::endl;
187
188       //Get a position above the sphere
189       target = move_group->getCurrentPose().pose;
190       target.position.z = sphere_initial.position.z + 0.3;
191
192       move_group->setPoseTarget(target);
193       move_group->move();
194
195       waypoints[0] = move_group->getCurrentPose().pose;
196       waypoints[0].position.x = 0;
197       waypoints[0].position.z += 0.1;
198
199       move_group->computeCartesianPath(waypoints, EEF_STEP,
200       JUMP_THRESHOLD, trajectory);
201
202       my_plan.trajectory_ = trajectory;
203       move_group->execute(my_plan);
204
205
206       std::cout << "Robot " << robot_i + 1 << ": " <<
207       "Finished going up." << std::endl;
208
209       // SYNC
210       if(!syncRobots(robot_i, 2)) {
211         va_end(args);
212         return;
213       }
214
215       std::cout << "Robot " << robot_i + 1 << ": " <<
216       "Starting shaking..." << std::endl;
217       ////////////////////////////////////
218       // Shake object
219       ////////////////////////////////////
220       initial = move_group->getCurrentPose().pose;
221
222       waypoints[0].position.y = initial.position.y;
223       waypoints[0].position.z = initial.position.z;
224
225       waypoints[0].orientation.x = facing_down.x();
226       waypoints[0].orientation.y = facing_down.y();
227       waypoints[0].orientation.z = facing_down.z();
228       waypoints[0].orientation.w = facing_down.w();
229
230       for (int i = 0; i < 3; i++) {
231        waypoints[0].position.x = initial.position.x + (i % 2? 0.2 : -0.2);
232
233         move_group->computeCartesianPath(waypoints, EEF_STEP,
234         JUMP_THRESHOLD, trajectory);
235
236         my_plan.trajectory_ = trajectory;
237         move_group->execute(my_plan);
238
```

```
239         // SYNC
240         if(!syncRobots(robot_i, 3 + i)) {
241           va_end(args);
242           return;
243         }
244       }
245 
246 
247       std::cout << "Robot " << robot_i + 1 << ": " <<
248       "Finished shaking." << std::endl;
249 
250       std::this_thread::sleep_for(std::chrono::milliseconds(10));
251 
252       ////////////////////////////////
253       //////// Stretch object ////////
254       ////////////////////////////////
255       std::cout << "Robot " << robot_i + 1 << ": " <<
256       "Starting stretching..." << std::endl;
257 
258       waypoints[0] = move_group->getCurrentPose().pose;
259       waypoints[0].position.x = sphere_initial.position.x - 0.1;
260 
261       move_group->computeCartesianPath(waypoints, EEF_STEP,
262       JUMP_THRESHOLD, trajectory);
263 
264       my_plan.trajectory_ = trajectory;
265       move_group->execute(my_plan);
266 
267       target = move_group->getCurrentPose().pose;
268 
269       target.position.y += robot_i % 2 ? 0.2 : -0.2;
270       std::cout << robot_i << " " << target.position.y << std::endl;
271       move_group->setPoseTarget(target);
272       move_group->move();
273 
274       std::cout << "Robot " << robot_i + 1 << ": " <<
275       "Finished stretching." << std::endl;
276       //SYNC
277       if(!syncRobots(robot_i, 6)) {
278           va_end(args);
279           return;
280         }
281 
282       ////////////////////////////////
283       //////// Go drop object ////////
284       ////////////////////////////////
285       std::cout << "Robot " << robot_i + 1 << ": " <<
286       "Starting drop trajectory..." << std::endl;
287       waypoints[0].position.x = sphere_initial.position.x - 0.15;
288       waypoints[0].position.y = sphere_initial.position.y;
289       waypoints[0].position.z = sphere_initial.position.z + 0.2;
290 
291       target.position.x = sphere_initial.position.x + 0.15;
292       target.position.y = sphere_initial.position.y;
293       target.position.z = sphere_initial.position.z;
294 
295       waypoints.push_back(target);
296 
```

```
297       move_group->computeCartesianPath(waypoints, EEF_STEP,
298       JUMP_THRESHOLD, trajectory);
299
300       my_plan.trajectory_ = trajectory;
301       move_group->execute(my_plan);
302
303       //SYNC
304       syncRobots(robot_i, 7);
305
306       ////////////////////////////////
307       // Release sphere by both robots and finish demo
308       ////////////////////////////////
309
310       std::cout << "Robot " << robot_i + 1 << ": " <<
311       "Released sphere " << sphere_i << " " << sphere_j << std::endl;
312       grabMsg.grab = false;
313       grabPub.publish(grabMsg);
314       gridState->setGrabbed(robot_i, false);
315
316       move_group->setNamedTarget(ALL_ZERO_POSE_NAME);
317       move_group->move();
318
319       va_end(args);
320
321     }
322   }
323
324 public:
325   SmallClothDemo(int n_robots) : Demo(n_robots) {
326     grabPub =
327     n.advertise<multiple_abb_irb120::GrabPetition>("/grid/grab_petitions",
328     100);
329   }
330
331   void execute(RobotInterface& robot, int robot_i, GridState& gridState,
332    int sphere_i, int sphere_j){
333     doDemo(robot, robot_i, 3, &gridState, sphere_i, sphere_j);
334   }
335
336 };
337
338 int main(int argc, char** argv){
339
340   // Setup
341   // ^^^^^
342
343   std::string name_ = "robots_controller";
344   ros::init(argc, argv, name_);
345   ros::NodeHandle n;
346   ros::AsyncSpinner spinner(1);
347   spinner.start();
348
349
350
351   geometry_msgs::Point robot_bases[2];
352   for(int i = 0; i < 2; i++){
353     robot_bases[i].x = 0;
354     robot_bases[i].y = i;
```

```
355     robot_bases[i].z = 0;
356   }
357
358   RobotInterface robots[2] = {RobotInterface(robot_bases[0],
359   "manipulator", "robot1"), RobotInterface(robot_bases[1],
360   "manipulator", "robot2")};
361
362   std::vector<std::shared_ptr
363     <moveit::planning_interface::PlanningSceneInterface> >
364     planning_scene_interfaces;
365   planning_scene_interfaces.push_back(
366   std::make_shared<
367   moveit::planning_interface::PlanningSceneInterface>("robot1"));
368   planning_scene_interfaces.push_back(
369   std::make_shared<
370   moveit::planning_interface::PlanningSceneInterface>("robot2"));
371
372   ros::Rate loop_rate(10);
373
374   // Start the Grid
375   // ^^^^^^^^^^^^^^^^^^^^^^^^^
376   std::vector<double> size(2);
377   ros::param::get("/grid/width", size[0]);
378   ros::param::get("/grid/height", size[1]);
379   std::vector<int> resolution;
380   ros::param::get("/grid/resolution", resolution);
381   std::vector<double> offset;
382   ros::param::get("/grid/offset", offset);
383   float sphere_radius = 0.025;
384   ros::param::get("/grid/sphere_radius", sphere_radius);
385
386   const int sphere_i[2] = {0, resolution[0] - 1};
387   const int sphere_j[2] = {0, 0};
388
389   GridState gridState(size, resolution, offset, sphere_radius,
390   planning_scene_interfaces, robots, 2);
391
392   ros::Subscriber sub = n.subscribe("/gazebo/link_states", 1000,
393   &GridState::updateCallback, &gridState);
394
395   while(!gridState.isReady()){
396     std::this_thread::sleep_for(std::chrono::milliseconds(100));
397   }
398
399   SmallClothDemo demo(NUM_ROBOTS);
400
401   std::thread t1(&SmallClothDemo::execute, &demo, std::ref(robots[1]),
402   1, std::ref(gridState), sphere_i[1], sphere_j[1]);
403   demo.execute(robots[0], 0, gridState, sphere_i[0], sphere_j[0]);
404
405   t1.join();
406
407   if(ros::ok()){
408     ros::shutdown();
409   }
410
411   std::cout << "Shut down" << std::endl;
412
```

```
413     return 0;
414 }
```

Anexos E

Recomendaciones para la modificación del código

Este apéndice está dirigido a cualquiera que desee realizar modificaciones a los ficheros desarrollados para este Trabajo, con una serie de pautas que puedan ayudarle a cumplir su objetivo:

- No modifique directamente los ficheros de ABB. Esto puede provocar incompatibilidades en futuras actualizaciones o con el resto de paquetes de la empresa. Siempre que pueda, mantenga éstos intactos y duplíquelos si su propósito lo requiere.

- Si desea realizar modificaciones al mundo de la simulación, vea el ejemplo provisto en la Sección B.4 y no olvide mantener la definición del *plugin*.

- Tenga cuidado con los espacios de nombre, o *namespaces*, pues la estructura puede ser delicada por las limitaciones de ROS 1.

- Si desea crear mensajes o servicios nuevos, siga el tutorial [99] y preste mucha atención a la estructura del fichero *CMakeLists.txt*, pues es el que garantizará la correcta compilación.

- Si usted utiliza el entorno de desarrollo *Visual Studio Code*, podrá obtener una extensión para ROS [100]. En la vista general de la misma podrá observar distintos tutoriales, pero es particularmente interesante la forma para depurar nodos de ROS desarrollados en C++ o Python. Deberá seguir los tutoriales, pero es importante saber que es necesario realizar una compilación con símbolos de depuración si desea colocar *breakpoints* en su código. Para ello debe utilizar el siguiente comando para compilación con *catkin*:
  `catkin_make -DCMAKE_BUILD_TYPE=Debug`

o colocar la siguiente línea en su *CMakeLists.txt* para compilación con *rosbuild*: `set(ROS_BUILD_TYPE Debug)`

– Si desea utilizar otro modelo de robots, le recomiendo modificar los ficheros de la carpeta *urdf*, basándose en los ficheros provistos por el fabricante del robot. También deberá modificar los archivos *srdf* y deberá adaptar los ficheros *launch* para que utilicen los nuevos archivos.