Escuela Politécnica Superior

Sistema de rehabilitación de miembro superior asistido mediante un robot colaborativo UR3

Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:
Carlos Castellanos Ormeño

Tutor/es:
Carlos Alberto Jara Bravo

Gabriel J. García Gómez

Universitat d'Alacant
Universidad de Alicante

Junio 2018

UNIVERSIDAD DE ALICANTE

Trabajo Fin de Grado

Sistema de rehabilitación de miembro superior asistido mediante un robot colaborativo UR3

Autor: *Carlos Castellanos Ormeño*

Tutores: *Carlos Alberto Jara Bravo*

Gabriel J. García Gómez

4 de julio de 2021

Resumen

El presente proyecto se basará en la utilización de un robot colaborativo UR3 para la creación del diseño y control necesarios para el desarrollo de una aplicación centrada en la rehabilitación del miembro superior.

En primer lugar, será indispensable realizar un estudio referente al marco teórico del proyecto a través del análisis del estado del arte de las técnicas e investigaciones relacionadas con el presente trabajo, como pueden ser la aplicación de exoesqueletos vestibles de miembro superior o la incorporación de robots industriales a tareas de rehabilitación.

Después de presentar las bases del proyecto, se procederá a explicar las herramientas utilizadas y las bases teóricas del proyecto. Por lo tanto, se deben explicar los componentes de ROS que harán posible los controles de posición y fuerza del brazo robótico, así como la interconexión del equipo remoto y el robot. Por otro lado, se detallarán claramente las rutinas de los controladores planteados desde su base teórica.

Posteriormente, se explicará todos los procesos y rutinas creados en la solución práctica a raíz de lo mencionado en el anterior apartado, resultando en la implementación práctica del control de posición y fuerza.

Obtenida la solución propuesta, se explicarán las pruebas realizadas con el fin de evaluar el funcionamiento de la idea propuesta, en tanto que se obtienen una serie de resultados que se procederán a analizar, con el objetivo de validar la solución y aclarar una serie de cambios y mejoras que adecuen su funcionamiento al ámbito de la rehabilitación.

Motivación, justificación y objetivo general

Durante todo el trabajo desarrollado se pueden distinguir dos campos principales referidos a la implementación de diversas leyes de control y la rehabilitación, siempre enmarcados dentro de la robótica. Dichos ámbitos son de una relevancia sobresaliente en la actualidad de la robótica y su combinación da lugar a aplicaciones novedosas que favorecen una clara mejoría, entre otros campos, a las rutinas de rehabilitación de pacientes con dificultad motora. Es esta propiedad el principal motivo para la realización del presente proyecto, con el fin de diseñar y crear un prototipo de aplicación de rehabilitación.

Por otro lado, a lo largo de la carrera diversas asignaturas me han servido para adquirir conocimientos acerca de estas dos áreas de conocimiento, siempre desde un punto de vista más teórico. Por lo que tras contactar con mi tutor para establecer las principales lineas de trabajo, podré ampliar mi experiencia en ambos campos desde una perspectiva práctica y personal.

A raíz de estas ideas se establece como objetivo principal la creación de un prototipo que pueda establecer una base práctica de una aplicación de rehabilitación del miembro superior con la implementación de diferentes controladores, utilizando para ello una infraestructura bien desarrollada y documentada como es el robot UR3 y el driver de ROS que permite el control externo del mismo.

Agradecimientos

En primer lugar, me gustaría agradecer a mis tutores Carlos y Gabriel toda su ayuda indispensable a lo largo de este proyecto, guiándome sobre distintos factores del mismo, así como la confianza depositada en mi. Además, quisiera extender este agradecimiento al grupo de investigación *Human Robotics*, sin el que este trabajo no sería posible. Además, mencionar especialmente a Damián Mira Martínez por toda su ayuda en el diseño y posterior creación de la herramienta desarrollada.

En segundo lugar, agradecer todo el apoyo y palabras de aliento por parte de mis amigos, indispensable en los momentos complicados durante estos cuatro años de carrera. En especial, en estas líneas no puedo dejar pasar la oportunidad de recodar a los amigos que he hecho en esta carrera; Sheila, Carmen, Adrián, Víctor y Ramón. Ellos han hecho este camino mucho más agradable y no podría estar más agradecido por los compañeros que encontré.

Por último, no puedo acabar sin agradecer todo el apoyo que me llega siempre desde mi casa. Gracias Mama por apoyarme y animarme a esforzarme siempre, gracias Papa por toda la ayuda que siempre me das, gracias Pablo y Adrián por sacarme una sonrisa cuando lo necesitaba. Además, tengo que mencionar la ayuda de mi primo Jaime a la hora de realizar los vídeos de la experimentación.

Lo único que podemos decidir
es qué hacer con el tiempo
que se nos ha dado.
J.R.R. Tolkien

Índice

Apartados **Página**

Índice de figuras **3**

1. Introducción **5**

1.1. Estructura y metodología propuesta 7

2. Análisis del problema **8**

2.1. Objetivos del proyecto 9

3. Estado del arte **9**

3.1. Rehabilitación de miembro superior 9

3.2. Análisis de sistemas de rehabilitación robóticos 11

4. Bases teóricas y componentes del proyecto **19**

4.1. Robot UR3 20

4.2. Efector final implementado 23

4.3. ROS 28

4.4. Paquete Moveit 31

4.5. Driver para el UR3: Universal_Robots_ROS_Driver 33

4.6. Control de admitancia 37

4.7. Control cinemático cartesiano 39

4.8. Librerías urdf_parser_py y pykdl_utils 42

5. Diseño de la solución práctica **43**

5.1. Conexión con el UR3 43

5.2. Control de posición 45

5.3. Control de fuerza 55

5.3.1. Control de admitancia 58
5.3.2. Control de velocidad 62
5.3.3. Adecuación para el ámbito de rehabilitación 66

6. Experimentación **70**
6.1. Pruebas del control de posición 70
6.2. Pruebas del control de fuerza 72

7. Análisis de los resultados **73**
7.1. Rendimiento del control de posición 74
7.2. Rendimiento del control de fuerza 75

8. Conclusiones **92**
8.1. Trabajos futuros 94

Bibliografía **96**

Apéndice 1: Código prueba **98**

Apéndice 2: Código del control de posición **99**

Apéndice 3: Código del control de fuerza **110**

Índice de figuras

1. Representación de la rehabilitación robótica . . . 6
2. Partes que componen el brazo humano . . . 10
3. Planos principales . . . 10
4. Movimientos y sus respectivos rangos . . . 11
5. Implementaciones para rehabilitación de miembro superior . . . 12
6. Posiciones del exoesqueleto y el brazalete . . . 14
7. Implementación desarrollada . . . 15
8. a) Utilización de señales EMG, b) Utilización de señales EEG . . . 16
9. Sensor de fuerza desarrollado y su implementación . . . 16
10. Simulación creada con el robot UR3 . . . 17
11. Estructura del sistema elaborado . . . 18
12. Gráfica posición-fuerza ejercida . . . 19
13. Robot UR3 . . . 21
14. Detalle de las medidas del extremo del UR3 . . . 24
15. Pieza creada en Inventor . . . 25
16. Corrección del modelado 3D . . . 26
17. Modelo impreso . . . 27
18. Unión final con el robot . . . 28
19. Estructura básica de ROS . . . 29
20. Estructura de funcionamiento de Moveit . . . 31
21. Representación virtual del robot UR3 . . . 34
22. Ejemplo de interacción controladores-robot . . . 37
23. Representación de la formulación en los distintos dominios . . . 38
24. Esquema del controlador cinemático cartesiano . . . 40
25. Representación de la inversa de la matriz Jacobiana . . . 42

26. Programa creado para habilitar el control externo 44
27. Representación de la obtención de las coordenadas 51
28. Comparación entre sistemas de coordenadas 59
29. Gráfica fuerzas-velocidad de la primera componente 76
30. Gráfica fuerzas-velocidad de la segunda componente 77
31. Gráfica fuerzas-velocidad de la tercera componente 78
32. Gráfica posición-tiempo con velocidad 0.01m/s 79
33. Gráfica posición-tiempo con velocidad 0.02m/s 79
34. Gráfica posición-tiempo con velocidad 0.05m/s 79
35. Gráficas posición-tiempo con avance en el eje X 80
36. Gráficas posición-tiempo con avance en el eje Y 81
37. Gráficas posición-tiempo con avance en el eje Z 81
38. Gráficas determinante - factor amortiguamiento 82
39. Gráfica fuerza-velocidades con una influencia 1/20 84
40. Gráfica fuerza-velocidades con una influencia 1/25 85
41. Gráfica fuerza-velocidades con una influencia 1/30 86
42. Gráfica fuerza-velocidades con una influencia 1/35 87
43. Gráfica fuerza-velocidades con una influencia 1/40 88
44. Gráfica fuerza-velocidades con una influencia 1/32.5 89
45. Gráfica fuerza-velocidades en el eje X 90
46. Gráfica fuerza-velocidades en el eje Y 91
47. Gráfica fuerza-velocidades en el eje Z 91

1. Introducción

La rehabilitación es uno de los procesos más importantes a la hora recuperación de diversas disfunciones físicas y sus consiguientes procesos neuromotores. En este campo, el estudio de las implicaciones relacionadas con la inclusión de mecanismos robóticos en el proceso de rehabilitación del paciente ha obtenido más relevancia en los últimos años.

Este proceso permite una mejora de recuperación del sistema nervioso, mediante la neuroplasticidad, que describe la habilidad de los nervios para formar nuevas conexiones con el objetivo de compensar heridas o cambios producidos.

Como se nos indica en [1], existen numerosos factores que alteran la neuroplasticidad, entre los que se encuentran la intensidad de la terapia y diseño de la misma o el tiempo transcurrido desde la lesión. Estos elementos se conocen como principios del aprendizaje motor y, actualmente, son objeto de estudio con el objeto de optimizar los programas de rehabilitación.

En lo referido a la robótica de rehabilitación, se ha estudiado en qué grado afecta la inclusión de diferentes tipos de sistemas robóticos en la terapia de rehabilitación y en sus principios del aprendizaje motor. En concreto, permiten ciertas mejoras respecto a la rehabilitación convencional, como el incremento de la intensidad de la terapia o la capacidad de aumentar el número de repeticiones, al mismo tiempo que se consigue que la precisión y la repetitividad del ejercicio no se vean repercutidas.

Otro de los puntos importantes que ofrece es la posibilidad de proporcionar *feedback* cognitivo, lo que supone un progreso a la hora de analizar la respuesta del paciente a la terapia y, de este modo, adaptarla en función de las necesidades dinámicas del paciente durante los periodos de la rehabilitación. De esta forma, se consigue una optimización del proceso, lo que conlleva una mejora en la recuperación del sistema nervioso. Por último, podemos destacar la capacidad de incluir juegos en las terapias, de forma que se favorezca la inclusión del paciente.

Figura 1: Representación de la rehabilitación robótica. Fuente: https://n9.cl/53nsb

Todos estas cuestiones que se han mencionado disponen una base de estudio que motiva este proyecto que buscará la inclusión de un robot colaborativo tipo UR3 a las terapias referidas a la rehabilitación de pacientes con discapacidad motora del miembro superior, de manera que el robot deberá realizar una serie de ejercicios activos y pasivos adaptados a las necesidades de cualquier paciente.

1.1. Estructura y metodología propuesta

Una vez visto este primer punto introductorio que sirve para contextualizar el proyecto desarrollado, en tanto que, se presentan la composición de la misma y los objetivos que se pretenden alcanzar; el siguiente apartado se centrará en el problema planteado de forma que se pueda deducir los requisitos y objetivos necesarios y, por consiguiente, proponer una solución.

Posteriormente, será necesario un estudio centrado en el análisis del estado del arte de las técnicas e investigaciones relacionadas con el presente trabajo. Después de presentar las bases, se detallará la solución planteada, entrando en detalle en las herramientas utilizadas y en la estrategia implementada.

En el siguiente punto, se explicarán las pruebas realizadas con el fin de evaluar el funcionamiento de la idea propuesta, en tanto que se obtienen una serie de resultados que se procederán a analizar en el apartado siguiente, de modo que verifiquemos la efectividad del mecanismo y la estrategia desarrollados.

Gracias a los datos obtenidos se podrá concluir con una serie de inferencias y, por otra parte, especificar los posibles trabajos o investigaciones futuras que se podrán realizar para incluir nuevas funcionalidades o solventar posibles errores.

En cuanto a la metodología que se seguirá en este proyecto, cabe destacar que es necesario, en primer lugar, establecer los objetivos y límites que definirán el alcance del trabajo. Una vez realizado este análisis, se procederá a la evaluación de las herramientas y conocimientos indispensables para llevar a cabo la experimentación. De esta manera, se desarrollaran las pruebas pertinentes y sus posteriores observaciones con las que obtener las conclusiones finales.

2. Análisis del problema

Antes de realizar la apreciación de técnicas desarrolladas en el ámbito de la robótica de rehabilitación, en este apartado nos centraremos en evaluar los diferentes aspectos del problema, de manera que se puedan definir claramente los objetivos a alcanzar.

Como se ha mencionado anteriormente, el robot empleado será el UR3, que deberá realizar el apoyo en los ejercicios que simulen una correcta terapia de rehabilitación. Para ello, este robot colaborativo tendrá que realizar y ejecutar movimientos adaptados a los movimientos de cualquier paciente para su rehabilitación del miembro superior.

El sistema de control deberá adecuarse a cada uno de los ejercicios y pacientes, realizando la lectura de los datos/sensores externos y aplicando en consecuencia el movimiento idóneo. Principalmente dos estrategias a llevar a cabo, respecto a la rehabilitación pasiva, el robot ejercita la articulación moviéndola completamente, mientras que el paciente no produce ninguna fuerza. Por lo tanto, los movimientos deben ser precisos y suaves, estableciendo una serie de límites que aseguren un rango de acción sin peligro.

Con respecto a la rehabilitación activa, el paciente debe participar en el proceso de rehabilitación, por lo que los movimientos del robot deberán obedecer a un control de fuerza para poder ofrecer el movimiento adecuado al paciente, teniendo en cuenta su fuerza ejercida. De esta manera, el paciente será el que gestiona la creación de los movimientos, mientras que el robot se centrará en seguir la trayectoria marcada, al mismo tiempo que brinda un apoyo de fuerza.

2.1. Objetivos del proyecto

En primer lugar, será necesario establecer una conexión entre el robot y la plataforma que se utilizará para crear y gestionar el control de los diferentes métodos mencionados. Por lo tanto, se explicarán los diversos componentes, así como su cohesión completa antes de entrar en detalle de los procesos de rehabilitación.

Seguidamente, el proyecto se centrará en el proceso de implementación de las dos estrategias mencionadas, de forma que en el comportamiento pasivo, el control decaiga en la posición del extremo del robot. Mientras, en la ejecución de la asistencia del robot al paciente activo, se realizará un control alimentado por la fuerza ejercida del paciente.

3. Estado del arte

Para establecer una base clara del trabajo será necesario empezar aclarando aspectos fundamentales presentes en las terapias de rehabilitación de miembro superior. A continuación, se indaga en soluciones propuestas que permiten integrar de una forma efectiva a los robots en las terapias de rehabilitación.

3.1. Rehabilitación de miembro superior

En primer lugar, es importante analizar y detallar las características que definen el sistema que compone el brazo, a fin de encontrar unas guías y límites para establecer la terapia de recuperación. Como definen en [2], el miembro superior se puede considerar como un mecanismo de cadena abierta formado por 4 eslabones, en relación con el brazo, antebrazo, mano y tórax, donde el último representaría la base:

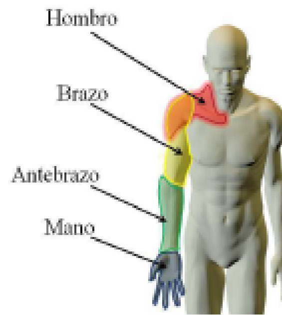


Figura 2: Partes que componen el brazo humano. Fuente:
http://dspace.ucuenca.edu.ec/handle/123456789/25820

El miembro superior permite un gran rango de movimientos, gracias a su número de articulaciones como puede ser el codo, uniendo el brazo y el antebrazo. Por tanto, es importante describir un rango de acción dentro de la movilidad articular del brazo, de modo que se pueda actuar sin entrar en una zona límite y peligrosa para el paciente.

Todos estos movimientos se comprenden dentro de diferentes planos imaginarios que dividen al cuerpo humano, en concreto, son el plano sagital, frontal y transversal [3]. Estos se distribuyen de la siguiente manera:

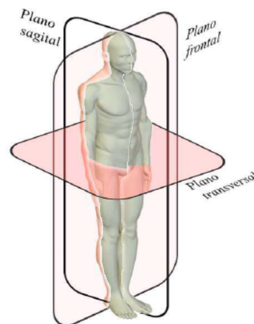
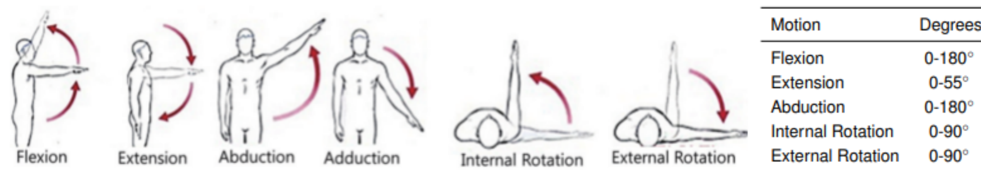


Figura 3: Planos principales. Fuente:
http://dspace.ucuenca.edu.ec/handle/123456789/25820

Los movimientos de flexión y extensión de hombro se engloban en el plano sagital, mientras que la abducción y aducción de hombro se da en el plano frontal. Asimismo, podemos establecer la rotación horizontal del hombro en el plano transversal. Esto se puede complementar con la siguiente imagen de [4], que recogen datos como los rangos de movimientos de cada movimiento.



Motion	Degrees
Flexion	0-180°
Extension	0-55°
Abduction	0-180°
Internal Rotation	0-90°
External Rotation	0-90°

Figura 4: Movimientos y sus respectivos rangos. Fuente: https://n9.cl/o3sxs

Gracias a esto, se pueden establecer una serie de límites a la hora de crear los movimientos que componen las terapias de rehabilitación. Para ello, se puede optar a múltiples soluciones, desde la realización de rutinas con la ayuda y supervisión de un profesional hasta que se puedan completar por uno mismo cuando se haya fortalecido la zona. En el siguiente apartado se contemplarán algunas de las estrategias seguidas que permiten la inclusión de robots para mejorar en algunos aspectos la rehabilitación del paciente, referidas al miembro superior.

3.2. Análisis de sistemas de rehabilitación robóticos

En primer lugar, es importante destacar las principales estrategias que definen tanto la forma de actuación como la elaboración del robot. Entre ellas encontramos dos grandes grupos, las soluciones que implementan exoesqueletos y las que utilizan dispositivos anclados de efector final [5]. En el primer grupo podemos encontrar que el exoesqueleto es completamente vestible o con un anclaje, mientras que en el segundo se puede diseñar un robot específico para la tarea de rehabilitación o acondicionar un robot ya creado como pueden ser los robots industriales.

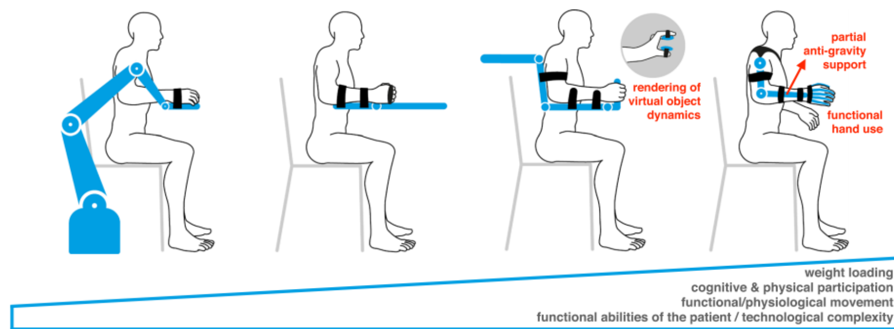


Figura 5: Implementaciones para rehabilitación de miembro superior. Fuente: `https://n9.cl/3gnmj`

Los sistemas de efector final se basan en generar movimiento desde un punto en el extremo del brazo del paciente y no existe un alineamiento entre las articulaciones del robot y el brazo [6]. Esto permite un ajuste más eficiente a los parámetros del paciente, al mismo tiempo que se brinda un control más simple. Asimismo, se puede acondicionar robots de otros ámbitos, facilitando la implementación al tener una plataforma ya definida. Sin embargo, al solo tener un apoyo en el brazo del paciente no se puede controlar la posición completa del brazo, aumentando el riesgo de lesión, por lo que se debe tener especial cuidado. Además, la adquisición de datos del paciente vuelve a estar limitada a un solo punto de conexión.

Mientras, los exoesqueletos permiten adaptarse completamente al brazo de la persona, coincidiendo sus respectivas articulaciones [6]. En cuanto a la programación, es necesario especificar el rango de movimiento por separado de cada articulación, con el fin de no forzar al paciente. Por otra parte, la inclusión de sensores en el exoesqueleto permite obtener un análisis más completo del paciente sin necesidad de aumentar el equipo que debe llevar el usuario. La gran capacidad de adaptabilidad mencionada al mismo tiempo puede suponer un inconveniente, ya que se ha de crear el exoesqueleto teniendo en cuenta un rango determinado de características de pacientes.

Estos dispositivos se pueden organizar también según en su objetivo respecto a la asistencia dad al usuario [6]. Así, los sistemas activos permiten una asistencia activa del paciente en el movimiento, mientras que los pasivos aseguran un apoyo al paciente que no aplicará potencia en la extremidad. Por otro lado, los esquemas de asistencia-activa deben completar el movimiento después de que el usuario lo inicia, en tanto que los mecanismos resistivos ofrecen una oposición a la ejecución del movimiento y, por último, los dispositivos interactivos, a través de unas trayectorias definidas, permiten corregir el recorrido de la extremidad.

Dentro de las alternativas se pueden encontrar múltiples proyectos que proponen sistemas de rehabilitación del miembro superior. En todos, sin embargo, es importante conocer la intención del usuario a la hora de moverse, de manera que se puedan adaptar a sus necesidades y proporcionarle un apoyo en la rehabilitación. Con este fin, encontramos que principalmente se suelen utilizar métodos de medición de fuerzas que permiten detectar el movimiento del brazo cuando este comienza el movimiento o, en cambio, se analizan los parámetros de la electriomigrafía dada por unos sensores específicos. Asimismo, se pueden encontrar proyectos que deciden estudiar el comportamiento de los motores de las articulaciones a fin de detectar cuando el usuario ha ejercido una fuerza. En los siguientes trabajos se ven más en detalla cómo hacen uso de estas tecnologías.

En primer lugar, en el trabajo presentado en [2] se propone la creación de un exoesqueleto de dos grados de libertad centrado en la rehabilitación de codo que permite realizar movimientos de flexión/extensión y supinación/pronación.

Para ello, se utiliza el brazalete Myo capaz de leer las actividades eléctricas de los músculos para traducirlas en poses y gestos del brazo. Este irá colocado en el brazo del terapeuta, de modo que el exoesqueleto colocado en el usuario replique el movimiento.

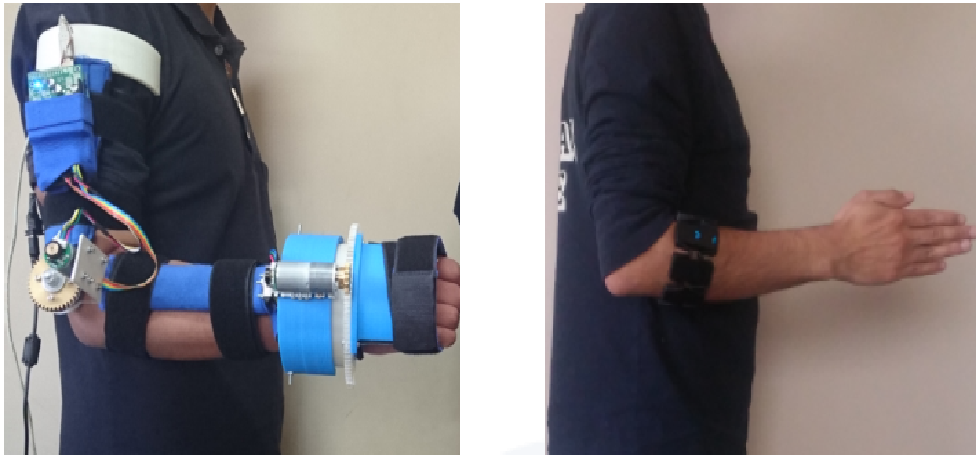


Figura 6: Posiciones del exoesqueleto y el brazalete. Fuente: http://dspace.ucuenca.edu.ec/handle/123456789/25820

A partir de los datos de cambio de pose que transmitirá el brazalete, el programa creado es capaz de transmitir órdenes de movimiento a los motores del exoesqueleto consiguiendo que el tiempo de respuesta con respecto al movimiento del brazo de referencia sea, aproximadamente, de 1,16 segundos. Además, durante las pruebas realizadas se obtuvo un error de posición de aproximadamente ±6 grados.

En los trabajos referidos a la utilización de robots industriales de efector final, podemos destacar la tesis [4], donde se configura el KUKA Lightweight Robot (LWR) junto con un sensor JR3 Force-Torque y una implementación personalizada de un terminal donde se acopla el antebrazo del paciente como se puede en la siguiente imagen:

Figura 7: Implementación desarrollada. Fuente: https://n9.cl/o3sxs

El control desarrollado se basa, en primer lugar, en definir un espacio de actuación resultado de la combinación del espacio de trabajo del robot y el del paciente. A partir de este punto, el fisioterapeuta puede mover el brazo robótico a fin de que este almacene las posiciones, velocidades y fuerzas que se deberán en los ciclos de entrenamiento. En todo momento, se analizará la respuesta del sensor de fuerza de modo que el brazo del paciente siga la trayectoria sin forzarlo, al mismo tiempo que el fisioterapeuta puede interferir en el recorrido para modificar algún punto.

Una aplicación con el mismo robot KUKA se puede ver en el recopilatorio [7], donde se propone un control basado en el procesado de la relación entre las fuerzas aplicadas en el efector final mediante un sensor de fuerza y la actividad muscular del paciente mediante sensores electromiográficos (EMG). Incluso en el mismo trabajo se hace mención a un desarrollo que pretende utilizar las las señales electroencefalográficas (EEG), permitiendo establecer una interacción entre el robot y pacientes con disfunción de movilidad severa.

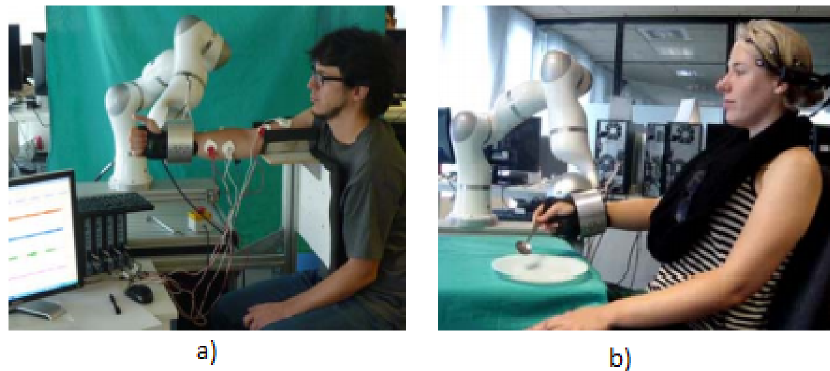


Figura 8: a) Utilización de señales EMG, b) Utilización de señales EEG. Fuente: https://n9.cl/lxnof

Existen otros trabajos centrados en la investigación de los sensores utilizados en el proceso de rehabilitación, como es [8]. El objetivo de este proyecto consiste en analizar las fuerzas producidas en la interacción humano/robot, de modo que se diseña un sensor basado en galgas extensiométricas que pueda sustituir a la propia percepción de fuerza del robot, apoyada en las lecturas de potencial de sus motores y que pueden verse afectadas por la configuración del robot, retrasos en las comunicaciones o histéresis. Cabe mencionar, que para la experimentación se utilizó únicamente movimientos de flexión/extensión del codo, donde el usuario hacía completamente el recorrido y, por otro lado, recibía la asistencia del robot.

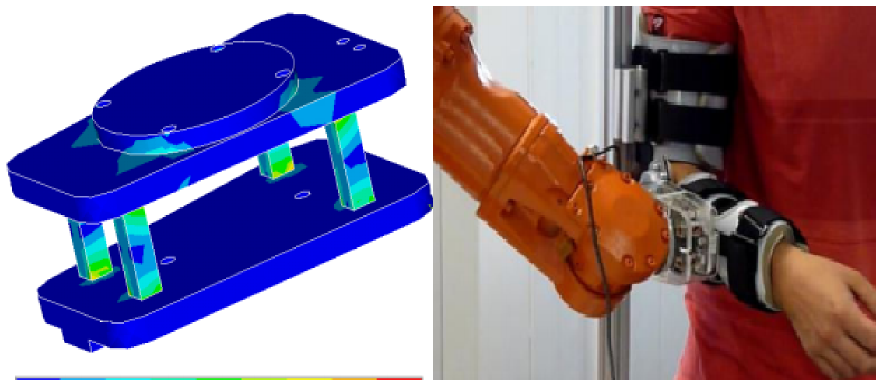


Figura 9: Sensor de fuerza desarrollado y su implementación. Fuente: https://ieeexplore.ieee.org/abstract/document/6290777

Centrándonos en trabajos que hayan tenido como base el mismo robot que se utilizará en el presente trabajo, es decir, el robot UR3 se puede encontrar los trabajos desarrollados en [9, 10]. La principal diferencia a destacar a priori de ellos es el entorno de trabajo, ya que en el primero se desarrollo una aplicación virtual, mientras en el segundo se estudia con un robot real.

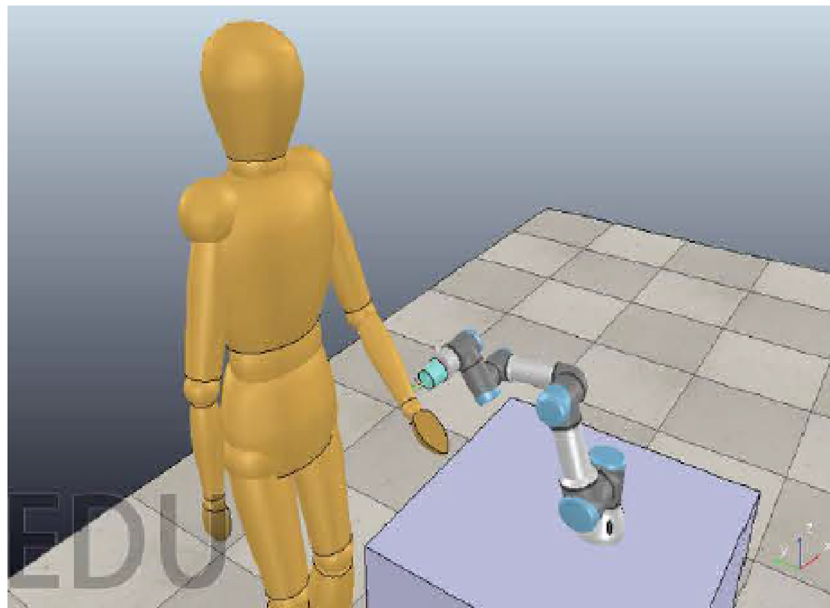


Figura 10: Simulación creada con el robot UR3. Fuente: https://link.springer.com/chapter/10.1007/978-3-030-36150-1_35

En ambos se presenta una interacción humano/robot donde la primera implementación es un acercamiento simulado para establecer una base que verifique el correcto funcionamiento de la idea. A partir de esta experimentación en un espacio virtual, en el segundo trabajo se elabora un sistema compuesto por el robot mencionado y un sensor de fuerzas junto a un efector final impreso en 3D que permita su agarre por parte del usuario.

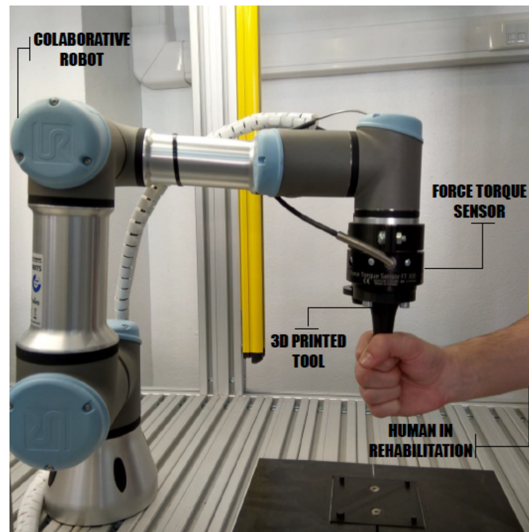


Figura 11: Estructura del sistema elaborado. Fuente: https://n9.cl/r7p7t

En cuanto al control, se pretende instaurar una estrategia resistiva al movimiento de usuario, de manera que cuando este empiece la trayectoria, el sensor de fuerza empieza a disponer datos al sistema de control. El paciente es capaz de mover el efector final en los tres ejes cartesianos sin que existan recorridos preprogramados, sin tener en cuenta las indicaciones del terapeuta encargado. De esta manera se pretende que el usuario ejercite el brazo mediante una resistencia creada por el robot que se adapta a la fuerza que ejerce el usuario.

El algoritmo encargado de manejar este comportamiento esta basado en una estrategia de aprendizaje por refuerzo, llamado 'SARSA learning' [10] que, dependiendo del movimiento generado por el paciente, actuará para producir una resistencia. Se cita que este algoritmo recuerda en su modo de trabajo a los algoritmos utilizados para resolver procesos de decisión de Markov, donde en los primeros estados realizará el método de prueba y error para generar una base de conocimiento.

Para demostrar el funcionamiento se nos presenta una gráfica que relaciona la posición del robot durante la ejecución de los episodios y la fuerza de resistencia generada. Cabe decir, que la fuerza está cuantificada en un intervalo entre 20N y 40N, de manera que el 0 corresponde con 25N:

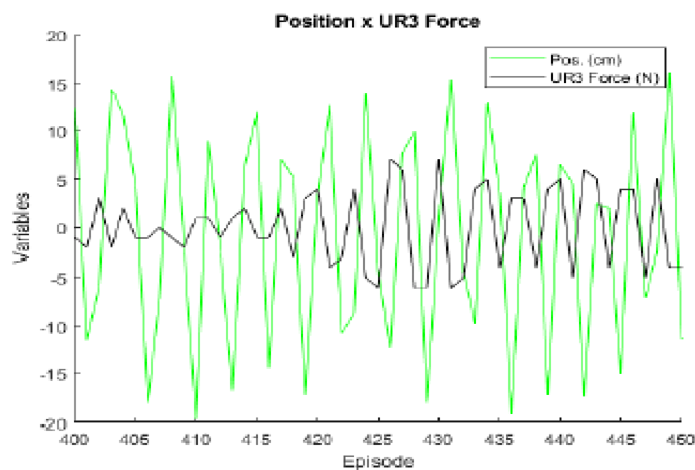


Figura 12: Gráfica posición-fuerza ejercida. Fuente: `https://n9.cl/r7p7t`

Lo más importante que se puede observar es que la fuerza generada, tras un pequeño retardo, se posiciona en contra del movimiento ejercido, sin embargo, se menciona que no siempre se puede mantener la fuerza en el rango seleccionado, aunque en términos generales prevalece.

4. Bases teóricas y componentes del proyecto

En esta apartado, se detallarán los aspectos que componen la base teórica y las herramientas utilizadas para alcanzar una solución funcional. Por lo que, se analizará los elementos físicos del sistema y las teorías que definirán las técnicas de control, con el fin de, posteriormente, presentar el control final empleado y aclarar los procesos que se deben llevar a cabo para el correcto funcionamiento.

A la hora de desarrollar el trabajo, existirán una serie de elementos físicos, siendo el principal elemento, el propio robot que ayudará el paciente, sin embargo, el funcionamiento del mismo se gestionará de manera externa, es decir, el control del robot no reside en él, sino que seguirá órdenes recibidas por paso de mensajes desde la plataforma donde se ejecutarán los diferentes archivos de control.

4.1. Robot UR3

El robot UR3, como se ha mencionado a lo largo del trabajo, ha sido el elegido para ejecutar la aplicación explicada. Este es un robot colaborativo (cobot) de sobremesa pequeño y ligero, llegando a pesar 10.9 kg y contando con una capacidad de carga de 3Kg. Asimismo, este robot cuenta con 6 grados de libertad rotacionales y una de las más importantes características a destacar es el rango de movimiento de sus articulaciones, siendo de ±360 grados, excepto la última que es de rotación infinita [11]. Esta capacidad de movimiento nos permite contar con un robot muy flexible y capaz de posicionarse correctamente en las posiciones que ayuden en la rehabilitación.

Otra de las propiedades útiles en el proceso de rehabilitación que cabe mencionar es su radio de trabajo de 500 mm, lo que nos permitirá establecer movimientos amplios que faciliten el movimiento del usuario. Además, cuenta con una gran precisión a la hora de posicionarse, como se puede ver en su datasheet en el apartado de la repetitividad que es de ±0.1 mm.

Figura 13: Robot UR3. Fuente: https://www.universal-robots.com/media/240787/ur3_us.pdf

Por último, destacar la comunicación que se utilizará para conectar el robot con el ordenador central que controlará el paso de mensajes. Esta se basará en utilizar el controlador del robot de modo similar a un servidor Modbus TCP. El ordenador donde se desarrollará el programa actuará como cliente, estableciendo la conexión a través de de mensajes de petición que siguen el estándar MODBUS.

Para hacer una primera comprobación de la conexión entre los dos elementos principales, se decidió realizar una prueba sencilla a través del programa MATLAB, que utilizará esta misma conexión explicada. A nivel físico, necesitaremos un router para establecer una conexión red local, utilizando dos cables Ethernet conectados al robot y el ordenador, respectivamente.

En el programa de prueba desarrollado en MATLAB, deberemos establecer la dirección IP del robot de manera que cuando se lance el programa se conecte al robot. Una vez conectado, se podrán mandar las peticiones de movimientos registradas en el programa o solicitar información de los sensores del robot.

A la hora de realizar movimientos, es necesario crear un vector que guarde las 6 variables articulares del robot o las coordenadas cartesianas del extremo, junto a los valores de orientación en los tres ejes. De esta forma, se tendrá dos vectores con 6 variables y pueden ser utilizados en cualquiera de los dos movimientos:

- **MoveJ**: realizará movimientos calculados en el espacio articular del brazo robótico, de manera que lleguen al mismo tiempo a la ubicación final deseada. En este caso, la posición de la herramienta no se mantiene fija. Además, de pasar el vector con la configuración deseada de cada articulación, es necesario pasar la velocidad máxima, aceleración máxima y el tiempo que definirá el movimiento. Igualmente, existe un último parámetro referido al radio respecto al punto final que se considera que el robot ha llegado al destino, por lo que si queremos un movimiento preciso lo dejaremos a 0. Este movimiento se usa, sobretodo, cuando se necesitan movimientos rápidos sin tener en cuenta la trayectoria de la herramienta.

- **MoveL**: utilizado para que la herramienta se mueva linealmente entre los puntos de paso. Esto significa que cada junta realiza un movimiento mas complicado para mantener la herramienta en una trayectoria recta. Nuevamente, se deben especificar los parámetros mencionados anteriormente, como la velocidad máxima. Este tipo de movimientos es especialmente útil para hacer movimientos precisos que necesiten mantener el extremo del robot fijo y transcurra por un recorrido lineal.

Cabe mencionar que existen más tipos de movimientos que se pueden consultar en [12], sin embargo, para la realización de una prueba simple, los movimiento explicados serán suficientes.

Por otra parte, cuando se especifique el tiempo que conlleve realizar el movimiento al robot, se debe tener en cuenta que es necesario insertar en el código un delay de espera antes de mandar otra petición de movimiento, ya que sino el que esta en ejecución no se finalizará.

Además de realizar movimientos, podemos acceder a la información del robot para imprimirla por pantalla. En nuestro caso, un dato trascendental sería el estudio de las fuerzas detectadas por el extremo, por lo que se pedirán al robot a lo largo de la ejecución para comprobar su estado en diferentes posiciones.

Todo lo anterior se puede ver en detalle en el apéndice 1 que recoge el código utilizado en este primer acercamiento. Igualmente, se puede acceder al vídeo, donde se puede ver la grabación del programa ejecutado. En él, se puede ver el éxito de realizar una primera conexión, además de comprobar que se ejecutan las órdenes en los tiempos establecidos y con una disposición de las poses muy precisas.

Enlace vídeo

4.2. Efector final implementado

Otro de los elementos indispensables en este proyecto se evalúa a la hora de incorporar al paciente en los movimientos del brazo robótico. Por este motivo, se decidió diseñar e implementar un agarre personalizado de manera que el usuario pueda interaccionar eficientemente con el UR3.

En primer lugar, se debe realizar el diseño del agarre a través de la aplicación Inventor 3D, que permite modelar diferentes estructuras en un entorno 3D desde formas básicas como cilindros hasta la incorporación de perfiles estandarizados de rosca para un tornillo.

Principalmente, la pieza resultante se compondrá de dos partes, la base creada para unirla con el extremo del robot y el extremo que estará en contacto con el usuario. Para el primer componente debemos partir de los planos del extremo del robot:

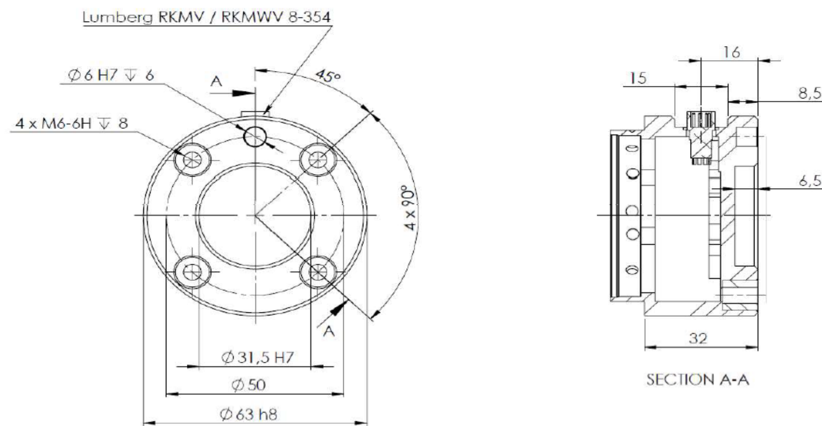


Figura 14: Detalle de las medidas del extremo del UR3. Fuente: documentación aportada por el tutor

Observando estos detalles, debemos tener en cuenta, por una parte, las medidas de rosca de los 4 incorporaciones que tiene el extremo, de manera que situaremos en nuestra base perforaciones de las mismas medidas, en concreto, se utilizarán perfiles roscados de M6xH6. De esta manera, se incorporarán cuatro tornillos M6 de 20cm de longitud, puesto que los huecos del extremo del robot son de 8cm, mientras que la base de la pieza será de 12cm de longitud. Por otro lado, se adecuará el radio exterior de la base para que coincida con el del extremo y así no sobresalga del mismo. Con estos datos, se realizará el diseño de la primera pieza.

Por último, se añadirá un cilindro desde la base creada con la altura suficiente para que se pueda realizar un agarre efectivo con la mano del paciente, en concreto, esta parte tendrá una longitud de 9.8 cm, por lo que en total a longitud de la pieza del efector final creada será de 11cm.

Para añadir estabilidad y un aguante mejor frente a las fuerzas ejercidas por el usuario se decidió añadir un hueco transversal a la pieza con el objetivo de introducir posteriormente una varilla de metal. Concretamente, el diámetro de este espacio será de 8.55 mm, permitiendo la operación descrita cuando se tenga la pieza impresa en 3D.

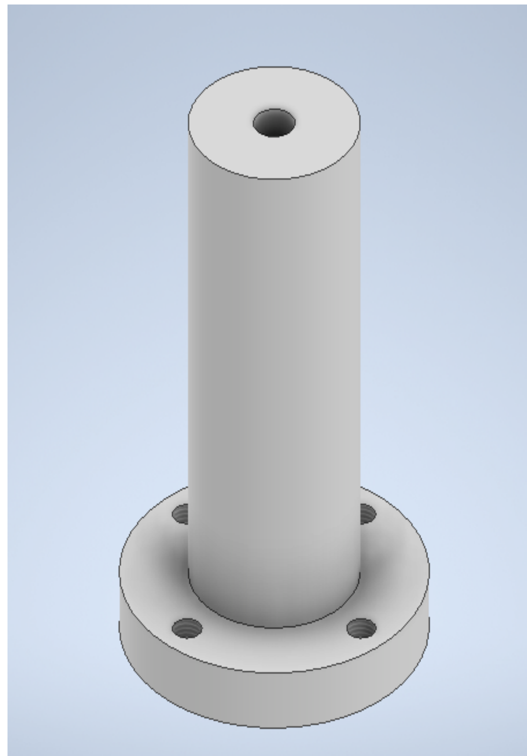


Figura 15: Pieza creada en Inventor. Fuente: elaboración propia

Gracias a este primer modelo creado a través de la aplicación de inventor, se podrá exportar dicho archivo a una formato compatible con la impresora 3D. Sin embargo, antes de realizar la impresión final de la pieza, se decidió hacer una serie de pruebas, que ayudarán a determinar el mejor diámetro interior de la pieza, a fin de que se ajuste óptimamente a la pieza de metal. Con este fin, se imprimieron diferentes cilindros con diámetros interiores diferentes y como resultado, el diseño final tendrá un diámetro interior de 8.55 mm.

Además, otra comprobación importante a la hora de confirmar el modelo creado es referida a la impresión de los perfiles roscados de los 4 agujeros donde irán los tornillos. Cuando se crearon se especificó el tipo de perfil roscado, sin embargo, el programa Inventor por defecto solo aplica un sombreado al hueco, por lo que para realizar el modelo 3D es necesario instalar el complemento llamado 'thread modeler', utilizado principalmente para crear modelos 3D a partir de los perfiles roscados de Inventor. Es imprescindible que para realizar un cilindro de prueba para verificar la impresión, primeramente se utilice dicho complemento obtener un interior roscado como se puede ver en la siguiente imagen.

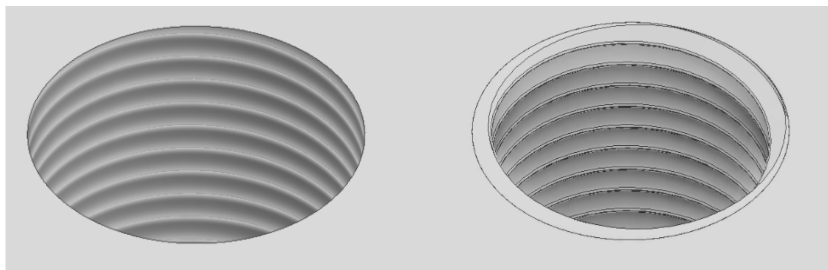


Figura 16: Corrección del modelado 3D. Fuente: elaboración propia

Esta propiedad es crítica, ya que se utilizará a la hora de anclar la base de la herramienta al extremo del robot y, por lo tanto, su modelado tiene que ser suficientemente preciso para que concuerde con el tornillo. Sin embargo, a la hora de realizar la impresión, se comprobó que la impresora utilizada no puede reproducir perfiles roscados tan pequeños, por lo que la pared del hueco estará lista. A pesar de este desperfecto, enroscando el tornillo a la pieza con la suficiente fuerza se podrá deformar esta pared, obteniendo el anclaje necesario. Por lo tanto, en los modelos desarrollados se mantendrá los cambios a fin de que puedan ser reproducidos en trabajos futuros y, por otra parte, con la solución aportada, se verifica esta parte del modelo para la impresión final.

Con este procedimiento realizado, se podrá obtener finalmente el efector final de la aplicación de rehabilitación desarrollada en este proyecto. Como muestra del éxito de esta parte del proyecto se muestra el componente final creado impreso en 3D:

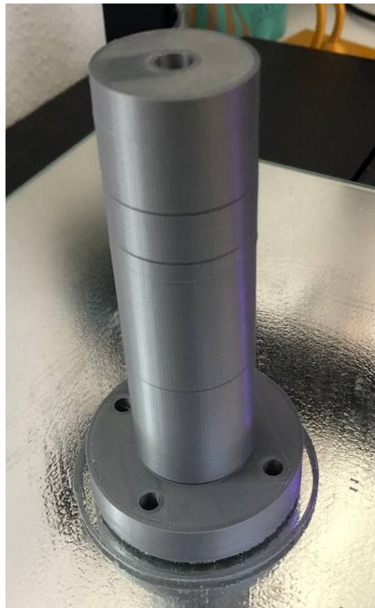


Figura 17: Modelo impreso. Fuente: elaboración propia

Finalmente, esta pieza se unirá al extremo del robot atornillando la base de este componente como se mencionado anteriormente, además de utilizar la varilla de metal para añadir una resistencia extra a la pieza creada. El resultado de las operaciones mencionados y, por tanto, la unión final que servirá para la inclusión del paciente se puede observar en la siguiente imagen:

Figura 18: Unión final con el robot. Fuente: elaboración propia

4.3. ROS

Por parte de software, se utilizará la plataforma ROS (Robot Operating System) de código libre que proporciona funciones flexibles para el manejo y control de robots. Este sistema provee una colección amplia de herramientas, librerías y convenciones que pretenden simplificar la tarea de crear comportamientos que rijan distintas plataformas robóticas [13].

El funcionamiento de ROS se puede comprender como un modelo gráfico de procesos, donde estos se generan en los nodos y están conectados por *topics*. Estos se representan como líneas que conectan nodos y permiten el paso de mensajes con información de manera que interaccionen entre ellos. El correcto funcionamiento depende de un nodo maestro que se encarga de registrar los nodos activos, roscore. Este maestro establece comunicación "peer-to-peer" [14] entre los nodos, por lo que los mensajes o llamadas no necesitan pasar por el nodo maestro.

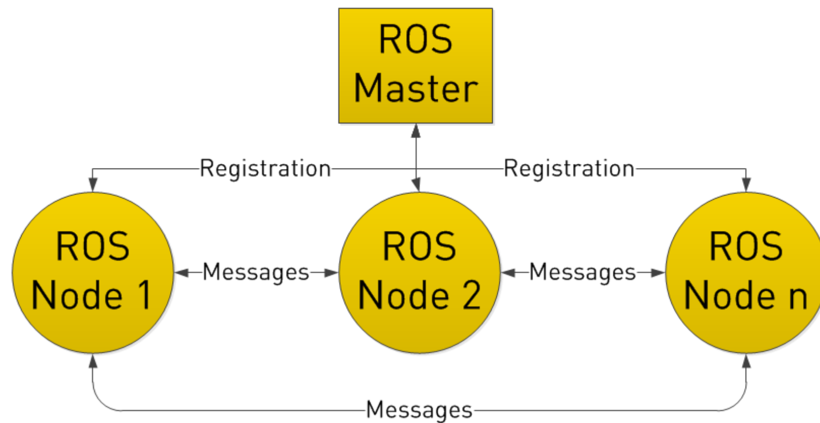


Figura 19: Estructura básica de ROS. Fuente:
https://www.ros.org/about-ros/

Además, de los *topics* existen los servicios que permiten crear un código específico para una funcionalidad del robot. Estos están estructurados en dos partes, en primer lugar, tenemos el 'Service Server' que provee la funcionalidad para que cualquiera pueda usarlo y, por otro lado, tenemos el 'Server Client' que es quien requiere el servicio. Sin embargo, una vez se hace la petición el flujo del programa se detendrá hasta obtener un resultado.

Otro de los elementos más utilizados en ROS son las *actions*, que permiten dotar al programa de funcionalidades externas sin detener el flujo de ejecución. Su funcionamiento se basa en crear, primeramente, una conexión al servidor de *Action* desde el nodo cliente, con el fin de enviarle un objetivo que debe alcanzar. Mientras, este servidor nos informará del estado a través de un *feedback* para actuar en consecuencia. Dentro del programa implementado se utilizará las *actions* que provee el paquete 'trajectory_msgs' que proveen capacidades para ejecutar órdenes de movimiento en el espacio articular.

Entre las ventajas que proporciona este tipo de control respecto a la solución de Moveit, que se explicará posteriormente, es la posibilidad de implementar un tiempo de ejecución de una trayectoria. Sin embargo, la definición de los puntos dentro del movimiento tienen que ser estipulados en el espacio articular. Asimismo, se pueden definir las velocidades articulares finales de los puntos a fin de que los puntos intermedios en las trayectorias no supongan un parón en la ejecución del recorrido.

Una muestra de la introducción de estas propiedades se puede ver a continuación, donde se define en el objetivo 'g' mediante la posición a alcanzar junto con las velocidades de las articulaciones finales y el tiempo de duración:

```
g.trajectory.points = [JointTrajectoryPoint(positions =
    pos_art, velocities = [0]*6, time_from_start = rospy.
    Duration(duracion))]
```

Por otra parte, ROS implementa una serie de herramientas que serán útiles a la hora realizar la simulación de robots en entornos virtuales, como pueden ser Gazebo y Rviz. El primero permite la representación de robots y la creación de escenarios 3D y, mediante un conjunto de paquetes, puede funcionar junto con ROS.

Por otra parte, Rviz nos permite obtener los detalles del robot, como sus mediciones, para poder modificarlo al mismo tiempo. Estos dos simuladores se utilizarán en lo referente al paquete Moveit explicado en el siguiente apartado. Asimismo, otra herramienta que se usará en los posteriores apartados es roslaunch que habilita el lanzamiento de los nodos establecidos en un programa junto con las configuraciones deseadas.

4.4. Paquete Moveit

Este software de código abierto [15] permite, entre otras cosas, realizar una planificación de la trayectoria del robot para, posteriormente, poder ejecutarlas tanto en un entorno simulado como en un robot real.

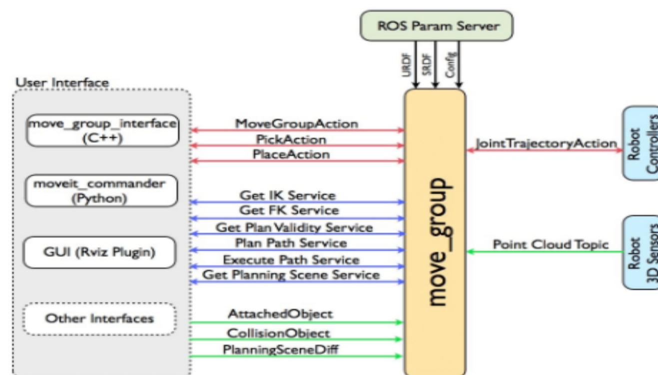


Figura 20: Estructura de funcionamiento de Moveit. Fuente: https://moveit.ros.org/

Como se puede ver en la imagen existen diversas interfaces de usuario para controlar el sistema total, por lo que podremos mandar las señales desde la ventana apropiada de Rviz para probar posiciones concretas y, posteriormente, crear una pauta de movimientos desde un archivo Python con el uso de la biblioteca 'moveit_commander'. Por lo tanto, se usará el lenguaje Python junto la biblioteca mencionada para ejecutar una serie de movimientos que se asemejan a los mencionados en apartados anteriores, de manera que podemos hacerlos en el espacio articular y en el cartesiano.

Existen diversas funciones que implementa la biblioteca mencionada con el objetivo de ejecutar los movimientos. Principalmente, se diferencian según el objetivo establecido, es decir, si se quiere introducir una posición final definida con los valores articulares, cartesianos o una pose predefinida:

- **<move_group>.set_named_target("name pose")**: esta función permite situar la posición objetivo con una ya preestablecida moviéndose en el espacio articular.

- **<move_group>.set_pose_target(pose target)**: en este caso se establece la posición según las coordenadas del efector final, por lo que se moverá en el espacio cartesiano.

- **<move_group>.set_joint_value_target(joint value target)**: por último, se definirá un conjunto de valores para las articulaciones de forma que se moverá en el espacio articular.

Una vez especificado el objetivo de cualquiera de las tres maneras posibles, se iniciará el movimiento del brazo robótico con la función <move_group>.go() ultimando el proceso de movimiento del robot.

Por otra parte, es posible acceder a la posición objetivo guardada anteriormente, de manera que se puede obtener la pose del robot en el espacio articular. Gracias a esto, se utilizarán las *actions* de 'trayectory_msgs' junto con una posición del espacio cartesiano, posibilitando la creación de una función que permita realizar el movimiento hasta un punto en concreto, cumpliendo con una serie de especificaciones de tiempo y velocidad.

Todos estos conceptos, se utilizarán a lo largo del programa desarrollado, de manera que se detallarán posteriormente y se pueden observar en el Apéndice 2 para la creación de comandos de movimiento desde el paquete de Moveit del driver que se explica a continuación.

4.5. Driver para el UR3: Universal_Robots_ROS_Driver

Dentro de toda la plataforma de ROS, se utilizará este driver a la hora de controlar y comunicarse con el robot UR3, ya que implementa una serie de herramientas que permitirán tener una base desde la que partir, como puede ser el archivo de conexión principal para conectar ambos elementos. En la página principal [16] se pueden ver toda la documentación referente al driver, en nuestro caso, el primer paso será descargarlo siguiendo el procedimiento detallado.

Al instalarlo, se deberán seguir una serie de pasos para poder habilitar la conexión entre el ordenador y el robot que se explicarán posteriormente en la puesta en marcha del proceso. Por otra parte, podemos encontrar archivos .launch que nos permiten realizar la conexión con los diferentes modelos de Universal Robots, en nuestro caso, deberemos utilizar los referentes al modelo UR3. Asimismo, mencionar que este proceso de conexión depende de otros archivos depositados en otras carpetas que permiten entre otras lanzar los controladores de los motores, por ejemplo.

Igualmente, se proporciona una carpeta suplementaria en la instalación con el trabajo realizado en anteriores drivers que han servido para manejar los robots de Universal Robots. En esta carpeta podemos encontrar la estructura creada para el funcionamiento del paquete Moveit del UR3, compuesta de los archivos que definen una estructura virtual y sus controladores entre otros. Gracias a esto, podemos lanzar también una versión virtual del robot en el espacio Gazebo y controlarlo al mismo tiempo.

Sin embargo, existen ciertos fallos debido a la utilización de distintos *topics* entre drivers, por lo que es necesario realizar cambios en el código para que se establezca la conexión, porque al no hacerlos, las órdenes de movimiento no se ejecutarán. Esto son debidos, principalmente, al cambio de nombre de los *topics* utilizados en la estructura del driver, por lo que actualizando los archivos se solucionará el error.

Es importante destacar que gracias a la simulación del UR3 se podrán realizar las estrategias planteadas y ajustarlas sin la necesidad de utilizar del robot real. A la hora de ponerla en marcha la aplicación de Gazebo junto con la versión virtual del robot se utilizaría el siguiente comando:

```
$ roslaunch ur_gazebo ur3.launch
```

El resultado obtenido se puede ver en la siguiente imagen:

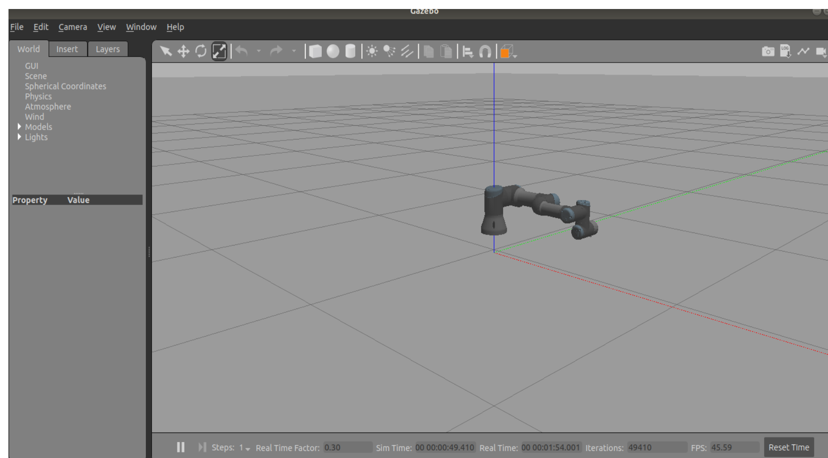


Figura 21: Representación virtual del robot UR3. Fuente: elaboración propia

A la hora de realizar la conexión con el robot real, es importante crear nuestro archivo previo de las calibraciones establecidas dentro de nuestro UR3, a fin de facilitar la gestión y control del mismo, ya que se tendrán en cuenta los datos exactos para calcular las cinemáticas inversa y directa.

```
$ roslaunch ur_calibration calibration_correction.launch \
  robot_ip:=<robot_ip> target_filename:="${HOME}/
   my_robot_calibration.yaml"
```

Este archivo debe ser cargado a la hora de realizar la conexión con el robot con el fin de utilizar esta calibración en el cálculo de las trayectorias de los ejercicios de rehabilitación.

Otro de los elementos importantes que establece este driver es a nivel del control establecido, ya que implementa una serie de controladores que se utilizarán a la hora de mover el brazo robótico. Dentro de este drive podemos encontrar el archivo 'ur3_controllers.yaml' en el que se definen los posibles controladores a elegir para el robot.

El primero de ellos y uno de los que se utiliza por defecto en el lanzamiento de la conexión con el robot del driver llamado 'scaled_pos_joint_traj_controller'. Este se utiliza sobretodo en tareas de manipulación y permite implementar la interfaz para acceder desde la estructura de Moveit o utilizando comando desde un servidor *Actions* de tipo 'control_msgs/FollowJointTrajectory', con el objetivo de realizar movimientos de posición basados en la creación de un elemento objetivo en el que se debe especificar esencialmente la posición articular del robot, además de otros parámetros como la velocidad, la aceleración y el tiempo de ejecución del movimiento.

Por lo tanto, para la realización de la aplicación basada en el control de posición, este controlador será el utilizado para la comunicación del robot real y el ordenador donde se ejecutará el control.

Por otra parte, dentro de las definiciones de los controladores podemos encontrar distintos tipos de ellos para ampliar las especificaciones del control realizado para la aplicación. A continuación, entraremos en detalle en los controladores que se deben utilizar a la hora de realizar un control de fuerza del robot. El primero de ellos nos permitirá acceder a las variables captadas en el extremo del robot referidas a la detección de fuerzas, de forma que se utilizarán para la realimentación del sistema. El *topic* utilizado por el programa será '/wrench' y es tipo 'WrenchStamped', lo cual nos permitirá obtener las fuerzas ejercidas en el extremo en los ejes cartesianos, tanto como los momentos resultantes.

El siguiente es referido a la introducción de órdenes al robot y, debido a sus propias características, se decidió interactuar con él a través comandos de velocidad. Por lo tanto, cuando realicemos este tipo de control, primeramente, se debe desactivar el control de posición y activar 'joint_group_vel_controller'. El funcionamiento de este controlador nos permite lanzar un nodo que se ocupará de introducir la velocidad articular a cada una de las articulaciones gracias a un vector de tamaño 6, coincidiendo con cada uno de los motores del robot. Además, al tratarse de un mensaje de tipo 'Float64MultiArray' podremos configurar parámetros para la correcta identificación del mismo en caso de necesitarlo, como la tarjeta con la que identificar el grupo de valores que definamos o la dimensionalidad de este.

Una forma de ejemplificar la podremos encontrar en [17], donde se nos explica la variedad de controladores que se pueden dar en ROS Control, además de un supuesto gráfico de la interacción entre estos y el hardware del robot.

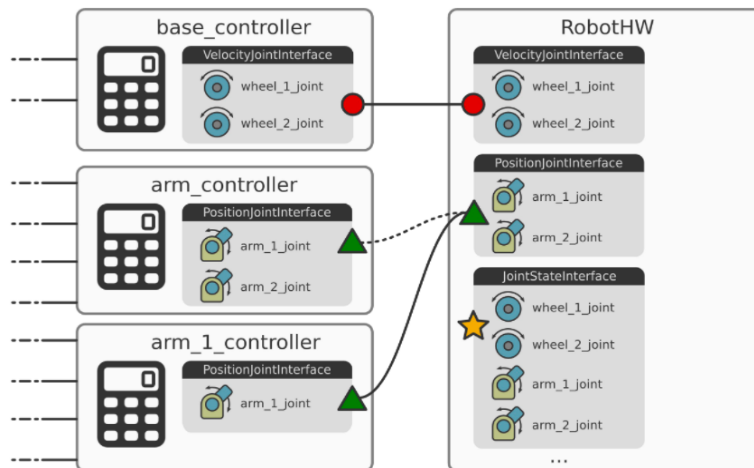


Figura 22: Ejemplo de interacción controladores-robot. Fuente: `https://fjp.at/posts/ros/ros-control/`

Como se puede observar, se pueden crear varios de ellos para que trabajen conjuntamente, siempre que interaccionen con distintos componentes del robot. Sin embargo, si estos necesitan acceder a un mismo elemento, no podrán funcionar al mismo tiempo y uno deberá desactivarse.

4.6. Control de admitancia

Una vez explicado la estructura y el control requeridos a la hora de manejar el robot, entraremos en detalle de la base teórica referente al control de fuerza. Para ello, en primer lugar, es necesario explicar cómo se obtendrán las velocidades cartesianas de referencia a partir de los datos de fuerza obtenidos por el UR3.

Este comportamiento reactivo deberá seguir la formulación especificada en el control de admitancia, que define tres parámetros principales, es decir, masa, amortiguación y rigidez. Como se menciona en [18], el efecto de la rigidez es lo suficientemente insignificante para ignorarlo del modelo, especificado en la siguiente fórmula:

$$F = m * \ddot{p} + c * \dot{p} \quad (1)$$

Donde la variable F representa la fuerza detectada, m es el parámetro referente a la masa virtual, c define el amortiguamiento de la función y $\ddot{p}, \dot{p}$ representan, respectivamente, los vectores de aceleración y velocidad del extremo.

Sin embargo, para hacer uso de esta función es necesario utilizar el dominio de la frecuencia para facilitar su procesamiento en tiempo de ejecución. Dentro de este sistema la entrada sería la fuerza detectada, mientras que la velocidad actuaría como salida.

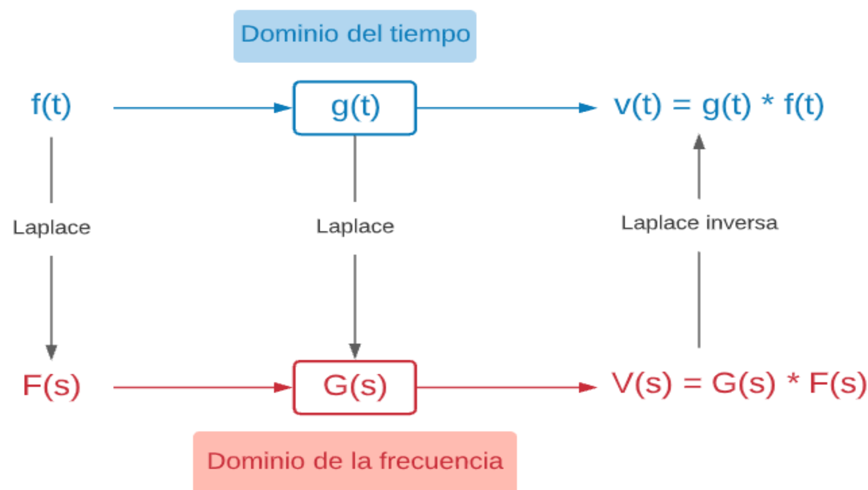


Figura 23: Representación de la formulación en los distintos dominios. Fuente: elaboración propia

Por otro lado, cabe destacar que al tratarse de valores de fuerza puntuales gestionados cada iteración del programa, solo será necesario multiplicar estos por la función de transferencia para obtener la velocidad deseada. En concreto, la función de transferencia de primer orden obtenida con la representación de Laplace sería la siguiente:

$$G(s) = \frac{\frac{1}{c}}{\frac{m}{c} * s + 1} \quad (2)$$

A partir de esta ecuación, se puede observar como la salida de velocidad máxima en estado estacionario depende del parámetro de amortiguación. Mientras, la dinámica de la respuesta es una relación acoplada entre los coeficientes de masa y amortiguamiento. Por lo tanto, para establecer una correcta relación entre un humano y un robot, estos parámetros deben ser determinados con el fin de que el robot adquiera un comportamiento que imite el movimiento del usuario.

Además, podemos deducir que el parámetro que representa la masa virtual será fijo en toda la ejecución del código, sin embargo, será el control del parámetro de amortiguación lo que nos permita manejar de cierta manera el movimiento del extremo del robot. Un valor bajo de esta variable permitirá al robot realizar grandes aceleraciones, a costa de limitar la capacidad de frenado, por el contrario, un valor alto evitará impulsos acelerados excesivos, limitando la aceleración del movimiento. Asimismo se podrá explotar dicho parámetro para crear funcionalidades referidas al control de la posición del brazo robótica en su espacio de trabajo.

Finalmente, este controlador nos devolverá un conjunto de velocidades cartesianas que se deberán gestionar, a fin de que puedan ser comprendidas por el robot.

4.7. Control cinemático cartesiano

La base teórica que nos permitirá controlar la velocidad del brazo robótico, se basa en la implementación de un controlador cinemático cartesiano. Un primer acercamiento se muestra en la siguiente imagen:

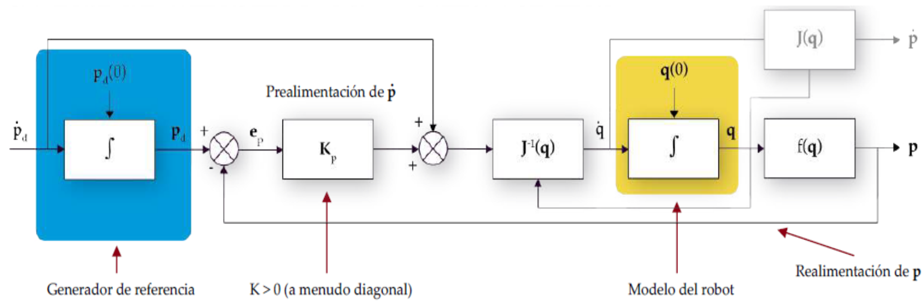


Figura 24: Esquema del controlador cinemático cartesiano. Fuente: documentación aportada por el tutor

La entrada de este controlador, como se puede deducir, será la velocidad en los ejes cartesianos del extremo del robot que se calculará dentro del control de admitancia. Por el contrario, la salida que se debe mandar al robot será un vector de seis velocidades articulares que, a través del controlador expuesto en el driver llamado 'joint_group_vel_controller', podrá manejar la velocidad de cada articulación.

La primera parte de este controlador es el generador de referencia, donde se deberá integrar la velocidad cartesiana deseada para obtener la posición deseada en ese ciclo del controlador. Con este objetivo se debe obtener el tiempo total de ejecución que especificará el límite de la integral de la posición y se deberá ir incrementando a medida que avance la ejecución. Para ejemplificar se muestra la ecuación que rige el cálculo de la posición deseada común para los tres ejes de coordenadas:

$$pos_deseada = pos_inicial + \int_0^t v \, \mathrm{d}t \tag{3}$$

Para obtener el error se debe guardar, también, el valor de la posición alcanzada por el extremo del robot en el ciclo anterior, de manera que se puedan restar ambos valores. Es importante destacar que en la primera iteración este valor de realimentación será la posición inicial del robot.

Una vez que hemos calculado el valor del error de posición, este se multiplicará por matriz diagonal de ganancia Kp que afectará a todas las componentes del vector de error de posición. Es importante destacar que no todos los valores dentro del matriz tienen que ser iguales, sino que se condicionarán para dar más importancia en el posterior vector de velocidad de control a las componentes cartesianas del vector de velocidades deseadas, cuyo valor sea superior o destacable frente al resto. Por ejemplo, cuando solo detectemos fuerza en el eje Z, la velocidad deseada calculada tendrá un valor significativo en esta componente, por lo que el trabajo de esta matriz de ganancia es dar más importancia al error en este eje, frente a los errores en las componentes X e Y.

El resultado de la multiplicación por la matriz de ganancia Kp se le sumará las velocidades deseadas cartesianas, obteniendo la velocidad final cartesiana de control (u).

La siguiente ecuación muestra el desarrollo matemático realizado para los valores de velocidades lineales, sin embargo, cabe destacar que realmente los vectores y matrices serían de dimensión 6, para controlar la orientación del extremo, aunque al no realizarse cambios en ellas se han omitido de la siguiente fórmula:

$$\begin{bmatrix} ux \\ uy \\ uz \end{bmatrix} = \begin{bmatrix} \dot{p}_d x \\ \dot{p}_d y \\ \dot{p}_d z \end{bmatrix} + \begin{bmatrix} K_p x & 0 & 0 \\ 0 & K_p y & 0 \\ 0 & 0 & K_p z \end{bmatrix} * \begin{bmatrix} error_x \\ error_y \\ error_z \end{bmatrix} \tag{4}$$

Uno de los componentes más importantes es el que nos permite obtener, a partir de la velocidad de control calculada anteriormente, la velocidad articular que se podrá mandar directamente al robot.

Con este fin, se requiere implementar el cálculo on-line de la inversa del Jacobiano del UR3, la cual nos permite obtener una relación directa entre las componentes cartesianas de la velocidad del extremo y las velocidades articulares del robot completo. Este desarrollo en el programa puede ser complicado, provocando problemas de singularidad, por lo que habrá que estudiar de cerca su funcionamiento, realizando un control que tenga como objetivo evitar estas singularidades.

$$\begin{bmatrix} \dot{q}_1 \\ \vdots \\ \vdots \\ \vdots \\ \dot{q}_n \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \dot{x} \\ \vdots \\ \vdots \\ \vdots \\ \dot{\gamma} \end{bmatrix} \quad \Rightarrow \quad \mathbf{J}^{-1} = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \cdots & \cdots & \cdots & \frac{\partial f_1}{\partial \gamma} \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \frac{\partial f_n}{\partial x} & \cdots & \cdots & \cdots & \frac{\partial f_n}{\partial \gamma} \end{bmatrix}$$

Figura 25: Representación de la inversa de la matriz Jacobiana- Fuente: elaboración propia

Posteriormente, en el apartado donde se explica completamente el diseño del controlador de fuerza, se entrará en detalle de las funciones creadas para el desarrollo de este controlador cinemático cartesiano y los valores utilizado para ello.

4.8. Librerías urdf_parser_py y pykdl_utils

Ambas librerías se utilizarán en el controlador en velocidad con el fin de calcular la inversa del Jacobiano del UR3. La primera de ellas se puede encontrar [19] y nos permitirá cargar el archivo URDF del robot en nuestro programa, por tanto, obtener los parámetros cinemáticos del mismo. Esta librería supone una mejora al no tener que definir manualmente todos los datos necesarios para realizar el control cinemático y, además, podremos basar los cálculos posteriores con la carga en el programa del archivo URDF del UR3 dado por el driver explicado.

En lo referente a la librería 'pykdl_utils' [20], partirá de las referencias obtenidas del archivo URDF cargado anteriormente con el objetivo de obtener valores cinemáticos del robot. Esta librería nos permite modelar el brazo robot a nivel cinemático y dinámico, por lo que de esta forma podremos desarrollar el cálculo on-line de la inversa de la matriz Jacobiana en el controlador cinemático cartesiano. Las dos librerías serán utilizadas en el controlador mencionado como se verá posteriormente en el apartado del diseño de la solución desarrollada.

5. Diseño de la solución práctica

Una vez explicadas las bases que influyen el presente trabajo, a continuación, se detallará los pasos seguidos para el desarrollo del sistema propuesto. Con este objetivo, se dividirá el proyecto en tres subapartados que expliquen la conexión, el control de posición y el control de fuerza.

5.1. Conexión con el UR3

El proceso para realizar una primera conexión con el UR3 se explica en [15], donde se nos comenta que se deben preparar tanto el robot como el PC externo. En el primer elemento se tiene que instalar un tipo de aplicación llamada 'externalcontrol-1.0.4.urcap', que como su nombre indica, habilita el control del robot externo. Una de las condiciones para poder implementar este tipo de control en el robot es que debe tener una versión de software superior a la 3.7, por lo tanto, previamente se actualizó el robot a la última versión disponible, la 3.14. De esta manera, la instalación del control externo no presentará mayores problemas y se podrá continuar con la conexión.

Al tener la aplicación activada, la configuraremos con la dirección IP de la máquina virtual 192.168.2.129, de manera que pueda obtener las órdenes de movimiento correctamente. Por consiguiente, solo quedaría implementar esta aplicación a modo de nodo dentro de un programa que debe ejecutar el robot. De manera ilustrativa, en la siguiente imagen se visualiza el programa creado en la página del driver [16], donde se comprueba los detalles de la IP utilizada en su caso.

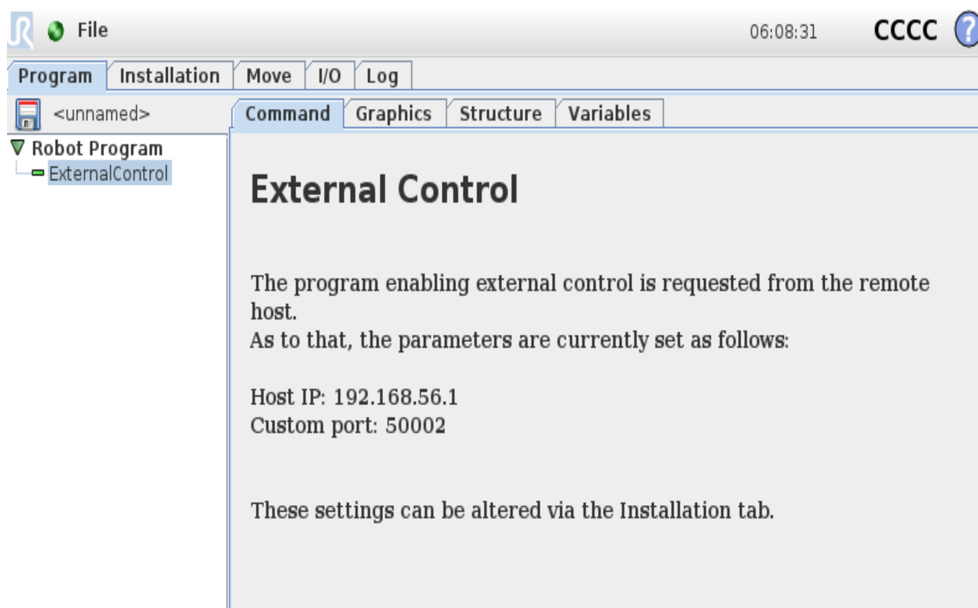


Figura 26: Programa creado para habilitar el control externo. Fuente: https://github.com/UniversalRobots/Universal_Robots_ROS_Driver

Este programa se guardará y se lanzará posteriormente, al mismo tiempo que se lancen los ejecutables de conexión desde el ordenador que se explicarán a continuación.

En la configuración del PC, la primera fase será extraer la información de calibración que el robot tiene almacenada de fabrica y, aunque no sea un paso obligatorio, si que es recomendable para posicionar el efector final con precisión.

Realizado el paso anterior, solo quedará lanzar el ejecutable que nos permite establecer la conexión, siempre que se esté ejecutando en el robot el programa creado antes. Para ello se debe ejecutar en la consola el siguiente comando:

```
$ roslaunch ur_robot_driver ur3_bringup.launch robot_ip
   :=192.168.2.2 kinematics_config:="${HOME}/
   my_robot_calibration.yaml"
```

Estas dos ejecuciones conectadas por la misma red local habilitarán la conexión compartida. Se puede confirmar que se ha realizado correctamente si en la ventana de la consola nos aparece el mensaje de 'Robot ready to receive control commands'. A partir de aquí, tendríamos una conexión estable que podremos probar con alguno de los programas de prueba que añade el driver a modo de comprobación con el siguiente comando:

```
$ roslaunch ur_driver test_move.py
```

Al ejecutarlo, el robot repite una misma trayectoria de tres puntos durante 10 ciclos, realizando movimientos precisos y veloces demostrando que este procedimiento consigue una manera estable de mandar las órdenes desde la máquina virtual al robot, en otras palabras, se obtendría un control externo que gobierne al robot y que servirá para la implementación de los comportamientos siguientes.

5.2. Control de posición

La base para realizar el movimiento del robot serán los comandos que ofrece Moveit dentro del paquete instalado del driver explicado anteriormente, junto con las *actions* que ofrece la librería 'trayectory_msgs. Por lo tanto, se podrá instaurar una serie de movimientos dentro de un archivo para que se ejecuten secuencialmente.

La ejecución de este comportamiento se encuentra dentro del archivo llamado 'cont_posicion.py', que corresponde con el Apéndice 2. Este archivo se encontrará dentro paquete llamado 'control_mov' que nos permitirá lanzar la ejecución con el comando rosrun, como se observará posteriormente. Nuevamente, dentro de programa deberemos de cargar una serie de bibliotecas que nos permite realizar movimientos con el sistema iniciado con Moveit o la capacidad de mandar mensajes de control de trayectoria, entre otras posibilidades.

Dentro del programa, en primer lugar se explicarán una serie de funciones creadas con el fin de implementar las funcionalidades mencionadas en el apartado anterior, es decir, la realización de los distintos tipos de movimientos. El objetivo de estas funciones es simplificar el código en la flujo del programa principal. Dichas funciones se explican a continuación y, por otra parte, se podrán ver en detalle en el Apéndice 2:

- **Función eu_to_q**

Como los comandos de posición, que se deben pasar a las órdenes que manejan Moveit, funcionan utilizando los cuaternios como datos de orientación, esta función nos permitirá expresar la orientación del efector final de una manera más intuitiva con los grados de Euler y transformarlos a la estructura que necesita el programa en las funciones posteriores que precisan de datos de posición y orientación. La transformación realizada se basa en las siguientes fórmulas:

$$\begin{aligned} qx &= \sin(R/2) * \cos(P/2) * \cos(Y/2) - \cos(R/2) * \sin(P/2) * \sin(Y/2) \\ qy &= \cos(R/2) * \sin(P/2) * \cos(Y/2) + \sin(R/2) * \cos(P/2) * \sin(Y/2) \\ qz &= \cos(R/2) * \cos(P/2) * \sin(Y/2) - \sin(R/2) * \sin(P/2) * \cos(Y/2) \\ qw &= \cos(R/2) * \cos(P/2) * \cos(Y/2) + \sin(R/2) * \sin(P/2) * \sin(Y/2) \end{aligned} \tag{5}$$

Siendo R, P e Y referencias a los valores angulares de las tres coordenadas que componen el sistema de ángulos de Euler (*Roll, Pitch, Yaw*), mientras que el vector formado por qx, qy, qz y qw hace referencia al nuevo sistema obtenido, en este caso regido por la orientación por cuaternios.

- **Función home**

Esta función nos permitirá mandar un comando de posición utilizando una de las poses predefinidas en el paquete Moveit del robot, utilizando una de las tres funciones descritas en el apartado referente a Moveit. Esta pose en concreto servirá como posición de inicio del programa y, como resultado, evitamos movimientos indeseados del robot, ya que siempre empezará en con la misma colocación.

- **Función mover_articular**

En este caso se requiere el paso de 6 valores como argumento, los cuales deben definir el posicionamiento del robot, es otras palabras, serán valores articulares para las 6 articulaciones en su pose deseada y, utilizando el control de movimiento del paquete Moveit, se realizará el recorrido en el mínimo tiempo posible, ya que no se puede influir de una manera realmente efectiva en la velocidad de los actuadores y, por tanto, ajustar el movimiento a las necesidades de un paciente.

- **Función mover_cartesiano**

En esta función se esperarán como argumentos los datos de posición del efector final respecto a un objetivo a alcanzar. La rotación se pasará a cuaternios con la función mencionada, para poder crear la variable pose con la que trabaja la función 'set_pose_target' del paquete Moveit. Nuevamente, este movimiento en el espacio cartesiano se realizará en el mínimo tiempo calculado.

- **Función move_controlado**

Es la primera función que transmitirá sus órdenes de movimiento con el servidor*Actions* mencionado en otros apartados. En concreto, tras recibir unas coordenadas de posición y orientación del extremo del robot, junto con el tiempo establecido; la fijaremos en la variable objetivo de Moveit para obtener su configuración articular.

Este paso es realmente importante, puesto que nos permite obtener la definición articular de la posición deseada y, por tanto, la posibilidad de utilizar este tipo de función que no esta basada en Moveit.

Los valores se podrán establecer en el 'goal' de la action, añadiendo también el tiempo para realizar el movimiento exitosamente. Es importante recordar que el movimiento realizado por esta función se define en el espacio articular, por lo que todavía no es acto para implementarlo en los ejercicios de rehabilitación. Con este objetivo, en la siguiente función se pretende crear una segmentación de la trayectoria a seguir en diferentes puntos de paso para forzar una trayectoria lineal para el efector final.

- **Función movimiento_lineal**

Esta función trabajará con la base desarrollada en el punto anterior de manera que a la hora de realizar la ejecución de movimientos dentro de los ejercicios, se debe ejecutar de una manera controlada facilitando la inclusión del paciente. Por consiguiente, esta función permite introducir el tiempo que debe tardar el recorrido completo. Este tiempo se dividirá según el número de iteraciones del bucle de control que fragmentará la trayectoria, definiendo un subtiempo idéntico en todos los subtramos del bucle.

Asimismo, se deben introducir la posición inicial y final deseadas, definidas en el código. Una vez obtenidas, el programa creará una serie de puntos intermedios, ya que nos permitirán crear una trayectoria lineal a seguir por el extremo del robot. Estos puntos de paso, no deben definir un movimiento completo como en la anterior función, sino que es necesario que la velocidad en el punto final no sea nula. Por lo tanto, se deben disponer una serie de velocidades intermedias para que el robot no frene en ellos.

Para el cálculo de las velocidades, un primer acercamiento es utilizando la fórmula para establecer la velocidad intermedia como velocidad de estos puntos. Una vez que se obtienen las referencias articulares de ambas posiciones, se creará un bucle que accederá a cada articulación, de manera que se obtendrá la velocidad media al dividir la resta de la posición final e inicial de la iteración en cuestión, con el tiempo del ciclo de manera que:

$$velocidad(x) = \frac{pos_final(x) - pos_inicial(x)}{tiempo_dispuesto/iteraciones} \tag{6}$$

En este caso, la x representaría los índices de cada una de las articulaciones, yendo desde la articulación 0 a la 5. Un importante detalle es que en la última ejecución se deben establecer las velocidades de las articulaciones a 0, es decir, se debe crear una condición en el código para el último tramo.

En este punto del programa se gestionan la evolución de la posición y la orientación del extremo de maneras diferentes. Para la posición siempre se regirá por el mismo procedimiento utilizando una fórmula lineal que asegurará una transición idéntica en todos los ciclos de trabajo:

$$qx_punto_de_paso = qx_inicial + (qx_final - qx_inicial)/iter * i \tag{7}$$

Siendo *iteraciones* el valor total de las iteraciones del bucle realizadas e *i* el número de la iteración actual. Esta evolución incremental se llevará a cabo en los cuatro valores de la orientación. Asimismo, al necesitar una trayectoria lineal, está fórmula también seria utilizada para el procesamiento de las coordenadas cartesianas del extremo.

Una vez calculado completamente el punto de paso del recorrido, se procederá al procedimiento de obtener los valores articulares necesarios para realizar el movimiento con el servidor de *Actions*. Es en este instante donde se calculan las velocidades de paso del punto para incluirlas en la definición del objetivo que constituirá las trayectoria completa. Por tanto, en cada iteración del bucle se añadirá un punto de paso del recorrido hasta llegar al punto final.

Con la trayectoria completamente configurada, se enviará al servidor *Actions* y se esperará a que esta se realiza. En definitiva, esta función será esencial en la ejecución de los ejercicios de rehabilitación, puesto que capacita al robot para seguir una trayectoria reconfigurable según las necesidades de la operación al poder introducir el tiempo de ejecución y las posiciones de inicio y fin del movimiento para establecer un recorrido lineal entre ellas.

- **Función movimiento_circular**

Una de las mejoras pensadas durante el desarrollo de la aplicación fue la introducción de otro tipo de trayectoria para el efector final del robot. Es por ello, que esta función trabajará con los componentes de la anterior, cambiando los aspectos necesarios para especificar un nuevo recorrido circular.

La primera idea introducir los mismos parámetros, es decir, con el conocimiento de un punto de inicio y final, sin embargo, el resultado sería la obligación de introducir dos puntos dentro de la misma circunferencia y a su vez se compatible con el rango de movimiento del robot. Por eso se decidió cambiar los parámetros iniciales y, por tanto, permitir que la llamada de la función introduzca el punto inicial, el tiempo de ejecución, el radio de la circunferencia y el ángulo de diferencia.

Gracias a esta implementación y colocando el punto inicial en un lugar centrado y bajo del espacio de trabajo del robot, conseguiremos la introducción de otros tipos de ejercicios de rehabilitación. El primer paso, será definir la posición final con los datos de entrada siguiendo esta estructura:

$$\begin{gathered} pos_final(x) = pos_inicial(x) + \sin(angulo) * radio \\ pos_final(y) = pos_inicial(y) \\ pos_final(z) = pos_inicial(z) + radio*[1 - \cos(angulo)]\ si\ angulo > 0 \\ pos_final(z) = pos_inicial(z) - radio*[1 - \cos(angulo)]\ si\ angulo < 0 \end{gathered} \tag{8}$$

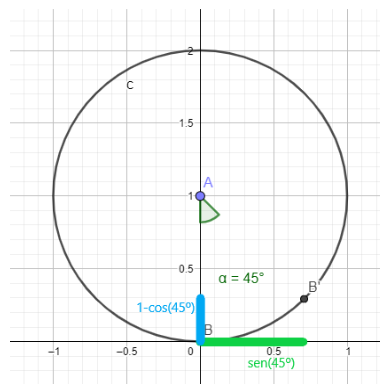


Figura 27: Representación de la obtención de las coordenadas. Fuente: elabotación propia

Como se puede deducir, lo movimientos circulares se realizarán en el plano x-z, coincidiendo con el plano frontal del robot, además que se puede distintas definiciones para el ángulo z, ya que el coseno tiene el mismo valor para ángulos positivos y negativos. Estos cambios hacen posible la regresión del movimiento al introducir el ángulo de movimiento negativo. Por otra parte, la orientación del extremo del robot será la misma que el punto inicial y, por consiguiente, en todo el recorrido.

El posterior procedimiento funcionará de la misma manera que en el anterior código, cambiando la definición de la trayectoria al cambiar las fórmulas que especifican las coordenadas de los puntos de paso. En nuestro caso, la coordenada y no tendrá cambios, al mantener su valor en todo el recorrido. Para las coordenadas x z se usarán las siguientes definiciones:

$$\begin{gathered}
\textit{si el angulo es mayor que } 0: \\
x = pos_inicial(x) + radio * \sin(angulo/iteraciones * i) \\
z = pos_inicial(z) + radio * [1 - \cos(angulo/iteraciones * i)] \\
\textit{si el angulo es menor que } 0: \\
x = pos_inicial(x) + radio * [\sin(-ang + ang/iter * i) - \sin(-ang)] \\
z = pos_inicial(z) + radio * [[1 - \cos(-ang + ang/iter * i)] - [1 - \cos(-ang)]]
\end{gathered} \tag{9}$$

La fórmulas necesarias para el ángulo menor que 0 se deben realizar de la forma descrita, ya que si utilizáramos más parecidas a las ecuaciones para ángulos positivos, el robot no seguiría la misma trayectoria utilizando para su movimiento circular un centro distinto.

Por último, destacar que esta función devuelve la posición final facilitando la siguiente llamadas a la función para que haga un movimiento de regreso. En ese caso, la posición final obtenida pasaría a ser la inicial y el ángulo dado sería el inverso.

- **Función ejercicio_1**

Una vez establecida la función que nos permite obtener una trayectoria idónea para la consecución de los ejercicios de rehabilitación, en la presente función se dispondrá las llamadas necesarias para implementar un modelo práctico de un ejercicio de rehabilitación apoyado por el control de posición del robot, es decir, que el esfuerzo de movimiento recae completamente en el brazo robótico.

En concreto, esta primera tarea de rehabilitación dispondrá de un movimiento de subida y bajada del extremo del robot en el que se apoyará la muñeca del paciente para realizar un ejercicio de flexión/extensión de la articulación del hombro. En principio, las posiciones que establezcan los límites del movimiento se fijarán en el programa al tratarse de un primer acercamiento.

La llamada de la función no contendrá ningún argumento, ya que será dentro de esta donde se definan las posiciones límite del ejercicio de rehabilitación. Para asegurar el correcto funcionamiento del robot, primeramente, se mandará al robot a la posición de inicio con la función descrita 'mover_cartesiano'.

Una vez colocado, se pedirá por pantalla el número de repeticiones del ejercicio, así como el tiempo de ejecución de los recorridos para llamar a la función 'movimiento_lineal'. Este paro en la ejecución de la aplicación puede servir también, como el momento en el que el paciente entra en contacto con el extremo del robot para empezar la terapia.

Con los datos pertinentes introducidos, se realizará un bucle que ejecute el ejercicio de rehabilitación, en otras palabras, en un mismo ciclo del bucle se realizará un movimiento de subida y otro de bajada, completando una repetición del ejercicio.

- **Función ejercicio_2**

Al igual que la anterior, esta función nos permite mostrar un ejemplo práctico del control de robot en su participación en las rutinas de rehabilitación. En este caso, se pretende que el paciente realice movimientos de abducción y aducción en un rango de movimiento variable a las especificaciones introducidas.

Nuevamente, no se recibirán parámetros en la llamada de la función, sino que se definirán las características necesarias dentro, ya sea como la disposición directa de la posición de inicio en el código como la petición de pantalla de las características del movimiento, es decir, el tiempo o el ángulo recorrido que se deberá pasar a radiantes. Asimismo, con el número de repeticiones se creará el bucle de ejecución que llamará dos veces a la función 'movimiento_circular' completando los dos movimientos mencionados. Gracias a la devolución de la posición final por la función, no será necesario especificarla en el código de esta, pudiendo utilizarla para la segunda llamada de retorno, junto con el ángulo inverso.

- **Función main**

En ella se deben definir las estructuras encargadas de la comunicación con el resto de la estructura de nodos predispuesta por el driver utilizado. Por lo tanto, se iniciará el nodo de control, además de realizar la conexión con el servidor *Actions* que manejará el paso de mensajes al nodo en cuestión que moverá al robot. Es necesario destacar, que este cambiará dependiendo de si se realiza junto con la simulación del ur3 o con el brazo robótico real. En este punto del programa, ya se podrán implementar las órdenes de movimiento.

A la hora de ejecutar la aplicación del control de posición, en primer lugar, es importante establecer una posición de inicio de manera que las siguientes trayectorias se garantice la seguridad del movimiento desde una postura conocida. Después, se decidió introducir una posición de paso intermedia antes de la realización de ejercicios, con el objetivo de condicionar las siguientes posiciones articulares del robot, ya que existían ocasiones que la cinemática inversa del propio robot resultaba en la posición articular complementaria que conllevaba un choque con la mesa en la que se sitúa el robot.

Por último, solo quedaría estructurar los patrones de movimiento para realizar una rutina práctica de ejercicios de rehabilitación. Posteriormente, se podrá evaluar la ejecución de toda la distribución comentada y verificar el grado de adaptación.

5.3. Control de fuerza

Como ya se ha mencionado anteriormente, este control nos permitirá analizar la intención del usuario de manera que el robot sea capaz de actuar en consecuencia y acompañar el movimiento, dependiendo de la fuerza ejercida por el paciente. Para ello, el control completo se dividirá en dos submódulos.

El primero de ellos, será el propiamente dicho como control de admitancia capaz de analizar las fuerzas presentes en el extremo del robot para obtener comandos de velocidad cartesiana que definan el siguiente movimiento del extremo del robot, a fin de compensar las fuerzas analizadas con el desplazamiento.

Por otro lado, deberemos establecer un controlador cinemático cartesiano, que permita transmitir las órdenes de velocidad cartesiana obtenidas, traduciéndolas a comandos articulares que puedan gobernar los motores que componen el robot.

Estas dos partes se definirán dentro de dos funciones principales que definen el comportamiento completo de la aplicación, sin embargo, antes es importante mencionar las definiciones y los procesos de preparación y acondicionamiento para obtener todas las variables necesarias.

El primero es la llamada al servicio de ROS control que permite cambiar entre los controladores lanzados, llamado '/controller_manager/switch_controller'. De esta forma podremos realizar movimientos para posicionar el robot en una posición controlada dentro de su espacio de trabajo, donde se presenten menos singularidades y luego pasar al controlador que permite los comandos en velocidad. Para variar entre los controladores solo tenemos que usar esta orden:

```
ret = switch_controller([controlador1], [controlador2],
   strictness, start_asap, timeout)
```

Donde se especifica que el controlador 1 será el que pasará a estar activo, mientras que el controlador 2 se parará. Por otra parte, el valor *strictness* define cómo de rigurosa será la conexión, puesto que si damos el valor 1, el servicio intentará lanzar y parar los controladores elegidos a pesar de existir algún fallo. Si elegimos el valor 2, el intercambio se finalizará dando un mensaje de error. El valor booleano *start_asap*, indicará si el intercambiador debe hacer el intercambio esperando o no a que todas las estructuras necesarias estén disponibles. Para un correcto funcionamiento, este valor se implementará como True siempre. Por último, el *timeout* define cuando dará por terminado el intento de conexión.

Por otro lado, será necesario la creación de un publicador al *topic* que transmitirá las velocidades articulares al robot y, por otro, la subscripción al *topic* donde el robot publicará los datos de fuerza percibidos en su extremo.

Antes de entrar en el bucle de realización de la aplicación y como ya se ha mencionado, mandaremos al robot a un posición articular más estable, a través de la función 'mover_articular' que utilizará el servicio de control de posición por paso de mensajes al servidor *Actions*, al igual que en el control de posición, por lo que no se entrará en detalle de esta función.

Otro de los parámetros que necesitamos establecer es la frecuencia de realización del bucle de control que influirá en la publicación de los mensajes de velocidad y la modificación de los datos que definen las fuerzas ejercidas en el extremo. Por lo tanto, se estableció una frecuencia de trabajo igual 60Hz, puesto que fue la frecuencia máxima conseguida al ejecutar por primera vez el programa. Al mismo tiempo, se debe crear una variable que guarde el tiempo del bucle, inverso a la frecuencia, ya que será de utilidad para realizar diversos cálculos posteriores.

Por último, obtendremos una serie de datos necesarios a la hora de ejecutar el controlador cinemático cartesiano como son los valores cinemáticos del robot guardados en un objeto KDL (Kinematics and Dynamics Library), además de los valores actuales de posición guardados en la variable de retroalimentación.

Una vez obtenidos los datos de partida, será necesario entrar en detalles de ambas partes para comprobar el proceso completo que se hará en esta parte de la solución. Ambos controles estarán representados por dos funciones principales, cuyos procesos serán explicados, así como, las llamadas a funciones auxiliares. Además, se hablará del control final que se ejecutará en bucle, utilizando ambas funciones principales y realizando un control global para adecuar estas funciones al ámbito de la rehabilitación.

5.3.1. Control de admitancia

Esta primera parte nos permitirá detectar el valor de las fuerzas aplicadas al extremo del robot para poder actuar en consecuencia, definiendo las velocidades cartesianas a seguir. Por lo que será necesario utilizar la subscripción definida anteriormente al *topic* donde el robot publicará los datos de fuerza.

La primera parte de este control se debe definir en la función **callback**, la cual se invocará al recibir un mensaje nuevo en el *topic* de fuerzas. En ella, se definirá una variable global llamada ’fuerza’ que guardará las tres componentes cartesianas de las fuerza lineal detectadas y transmitidas en el mensaje. Sin embargo, en la realización de pruebas para comprobar los valores dados por el *topic* , se constató que los datos obtenidos están definidos en otro sistema de coordenadas.

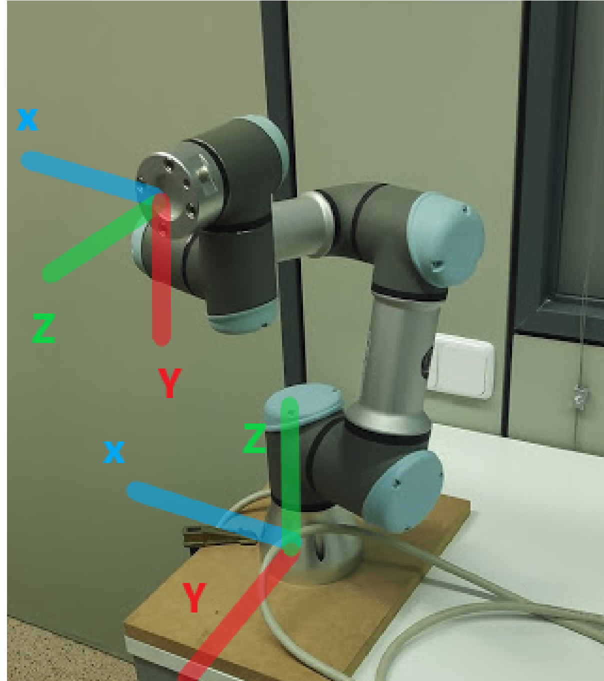


Figura 28: Comparación entre sistemas de coordenadas. Fuente: elaboración propia

Por consiguiente, será indispensable transformar los datos obtenidos en el sistema del extremo para expresarlos en el sistema de la base que definirá el movimiento cartesiano del extremo en el control de velocidad. La transformación se puede abordar fácilmente con una matriz de rotación de -90 grados respecto al eje X:

$$\begin{bmatrix} fx_{base} \\ fy_{base} \\ fz_{base} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(-90) & -sin(-90) \\ 0 & sin(-90) & cos(-90) \end{bmatrix} * \begin{bmatrix} fx_{extremo} \\ fy_{extremo} \\ fz_{extremo} \end{bmatrix} \tag{10}$$

De esta manera, se consigue expresar las fuerzas percibidas en el mismo sistema coordenadas que se utilizará para controlar la velocidad del robot. Este valor de fuerza será accesible en el resto de las funciones para realizar los cálculos convenientes.

La función principal del control de admitancia se llamará **control_admitancia** y será utilizada al principio del ciclo de ejecución de la aplicación para obtener las velocidades cartesianas deseadas en dicha iteración. Dentro de ella, primeramente, se debe crear un parámetro inicial antes del bucle que guarde una referencia de los valores de fuerza detectados en la posición de trabajo. Este se pasará a la función a fin de calcular los incrementos de cada eje, respecto a las fuerzas detectadas en primera instancia.

Este vector con el valor de los incrementos de los tres ejes cartesianos se utilizará para calcular las velocidades deseadas. Sin embargo, es importante destacar que en las ejecuciones iniciales del algoritmo se producían variaciones en estas variables sin que hubiera una fuerza externa. Es por ello que se decidió que para los procedimientos posteriores se tuvieran en cuenta los incrementos que detectarán un cambio superior respecto a un umbral de +/- 40. En caso contrario, este valor se considerará nulo y por consiguiente la velocidad de ese ciclo será 0.

Del mismo modo, se comprobó que realizar movimientos con el robot se pueden alcanzar incrementos con valores muy superiores, que pueden ser debidos a la realización del movimiento por parte del robot, mientras aún se sigue aplicando fuerza. Estos valores provocaban velocidades peligrosas y el robot además entraba en bucle oscilatorio, donde el robot detectaba fuerzas desproporcionadas y contrarias. Por esta razón, se decidió crear un límite superior para el incremento de la fuerza en los ejes, que se situará en +/- 200.

Por otro lado, se debe especificar los parámetros de la función de transferencia, definida con la siguiente ecuación:

$$G(s) = \frac{\frac{1}{c}}{\frac{m}{c} * s + 1} \tag{11}$$

El primero de ellos será la masa virtual del robot, que se definirá por las propias características del mismo, es decir, se utilizará una masa de 10.9 Kg. El segundo de estos parámetros es la coeficiente de amortiguación que debe ser capaz de conseguir que un control adecuado por parte del paciente, sin embargo, como se mencionó en el apartado de explicación del control de admitancia se pretende manejar esta variable para dificultar el movimiento a posiciones cercana a las singularidades del robot.

Con este fin el valor de este coeficiente variará según el determinante de la matriz Jacobiana obtenida en el ciclo anterior del bucle de trabajo. Este determinante nos dará una idea de la cercanía de una posición singular, por lo tanto a menor valor de esta variable, se debe aumentar el coeficiente de amortiguamiento dificultando el movimiento del extremo del robot al paciente. Es por ello que sistema favorece las trayectorias dentro de un campo estable del espacio de trabajo del robot, evitando entrar en singularidades. Por consiguiente, la formulación que definirá esta relación será la siguiente:

$$c = c + \frac{1}{determinante} \tag{12}$$

La razón por la que se incluye el factor de coeficiente dentro de la ecuación es debido a que en la primera iteración cuando no se tiene un valor real del determinante, este valor de amortiguamiento no puede quedar sin valor, ya que la ganancia debe estar operativa siempre. Por lo que se inicializará con un valor de 1 que solo tendrá efecto en la primera iteración y, realmente, no afectará al funcionamiento de la aplicación.

Esta formulación del coeficiente de amortiguación funcionará adecuadamente, siempre y cuando el robot permanezca en una región estable de su espacio de trabajo. No obstante, se consideró que su valor no es suficiente en regiones cercanas a la singularidad. Es por esta razón que se decidió añadir una condición extra cuando el determinante este por debajo del valor 0.005, de manera que el coeficiente sea el resultado de la siguiente ecuación:

$$c = c + \frac{10}{determinante} \tag{13}$$

De esta forma, la amortiguación en estas regiones más sensibles será un orden de magnitud mayor por defecto, favoreciendo una rigidez que evite acercarse a una posición singular.

Una vez definida la función de transferencia y los incrementos de las fuerza del extremo, se creará un bucle para obtener la velocidad de los tres ejes, a través de la multiplicación si el valor del incremento se encuentra dentro de los límites establecidos. Sino, el valor de la velocidad será 0. El resultado de este bucle serán las velocidades cartesianas que se deberán pasar al control cartesiano en velocidad.

5.3.2. Control de velocidad

A continuación, explicaremos el desarrollo del programa referido al controlador cinemático cartesiano partiendo de los datos de velocidad obtenidos anteriormente. La función principal para el manejo de esta parte de la aplicación se llamará **control_vel_cart** a la que se le pasará como parámetros la velocidad cartesiana deseada, el tiempo de ciclo, la posición de referencia del robot correspondiente al ciclo anterior y el objeto KDL.

Para su procedimiento partiremos de la explicación detallada en el apartado 4.6, por lo que si tenemos como entrada las velocidades lineales del extremo del robot, se realizará el generador de referencia para obtener la posición deseada en ese ciclo del controlador. Este procedimiento seguirá la ecuación 3.

Sin embargo, a la hora de implementar esta función en el controlador se observó que no es lo suficientemente robusta a la hora de presentar múltiples cambios de dirección de la velocidad, ya que siempre partirá de una posición inicial sin tener en cuenta la posición en la que se realiza el cambio de dirección. Esto resultaba en un fallo a la hora de mover el extremo del robot, por lo tanto se procedió a realizar el siguiente cambio:

$$pos_deseada = pos_anterior + \int_0^{t_int} v \, \mathrm{d}t \tag{14}$$

En esta función, se parte de la última posición alcanzada por el robot y el límite superior de la función integral será el tiempo de un ciclo del programa, obteniendo la posición deseada de la iteración. De esta manera, el robot reacciona correctamente a los cambios de dirección típicos que puede realizar el paciente.

Todo el procedimiento anterior será realizado dentro de la función **calculo_pos_d**, que recibirá el vector con las velocidades cartesianas, el tiempo de ciclo del controlador y la posición de referencia previamente obtenida. Cabe mencionar que para el cálculo de la integral que se realiza en esta parte se utiliza la librería scipy, que incorpora a python este tipo de cálculos. En concreto, utilizaremos la parte de la librería llamada 'integrate', que nos permitirá realizar el cálculo de la siguiente manera:

```
integrate.quad(lambda x:velocidades_cart[i], 0, t_int)
```

Como se puede observar, lo que se obtendrá es una progresión lineal, donde se multiplica la velocidad cartesiana del ciclo por el tiempo total. Gracias a la aplicación de la función 'quad', se podrán realizar cambios fácilmente en la definición de la integral para calcularla en el intervalo previsto. Sin embargo, es importante destacar que el valor devuelto se encontrará en una tupla por lo que a la hora de sumar la posición inicial, se deberá incluir esta en otra tupla y sumar ambas.

Por lo tanto, se realizará este procedimiento para las tres coordenadas cartesianas y, solo, devolver los primeros valores de las tuplas que corresponde a la posición deseada en las coordenadas XYZ.

Una vez obtenidas las posiciones deseadas, se realizará el cálculo del error entre estos valores y la posición de referencia obtenida en el ciclo anterior y, por lo tanto, se podrá realizar la estimación de la velocidad de control necesaria. Este vector de valores será la suma de las velocidades obtenidas en el control de admitancia y la influencia de los errores de posición.

$$\begin{bmatrix} ux \\ uy \\ uz \end{bmatrix} = \begin{bmatrix} \dot{p}_d x \\ \dot{p}_d y \\ \dot{p}_d z \end{bmatrix} + \begin{bmatrix} K_p x & 0 & 0 \\ 0 & K_p y & 0 \\ 0 & 0 & K_p z \end{bmatrix} * \begin{bmatrix} error_x \\ error_y \\ error_z \end{bmatrix} \tag{15}$$

Dicha influencia es la que se debe gestionar a través de la matriz de ganancia diagonal Kp y, por tanto, variaremos sus componentes. Para ello, en primer lugar obtendremos

Obtenida la velocidad de control final que se precisa para manejar el robot en la presente iteración, se procederá a calcular el valor de la inversa de la matriz Jacobiana en este instante. El primer paso para ello, es obtener la configuración articular del robot actual, llamando a una función auxiliar similar a la utilizada para obtener la posición en determinados instantes. Con esta información de base, se realizarán los procedimientos que requieren las funciones de la librería 'pykdl_utils'. La primera de ellas será para calcular la matriz Jacobiana de la base del robot, utilizando el objeto KDL que contendrá la función 'jacobian' y como parámetros de entrada, solo necesitará la posición articular del robot.

De esta matriz se obtendrán dos elementos importantes para el control posterior, por una parte se guardará el determinante de la matriz, para medir objetivamente la cercanía de la posición actual del robot a una singularidad que provoque un error en la ejecución de la aplicación. Este valor se mandará como resultado de la presente función a fin de que en el bucle de control se tomen medidas en función del valor del determinante. Concretamente, se podrá modificar la resistencia de movimiento del extremo del robot cuando el valor de este determinante alcance un cierto umbral. De esta forma, se pretende evitar que el usuario entre en espacios de trabajo del robot comprometidos.

Por otro lado siguiendo la ejecución de la función, se procederá a realizar la inversión de la matriz Jacobiana, por lo que solo quedaría multiplicar esta matriz por el vector de velocidades de control calculadas anteriormente. Con este último paso, se obtendrá un vector de seis velocidades correspondientes a cada articulación del robot y se mandarán como segundo resultado de la función con el fin de mandar estas velocidades al robot en el bucle de control.

5.3.3. Adecuación para el ámbito de rehabilitación

Como se ha mencionado anteriormente, el control del robot se realizará dentro del bucle de control, en el cual estarán la función que permite gestionar las fuerzas detectadas y la función que implementa el control cartesiano articular.

La primera opción que se debe implementar imperativamente es la gestión de tiempos y paradas dentro del sistema, de modo que la interacción con el robot sea seguro y no comience el movimiento hasta que el usuario se encuentre en la posición adecuada. Esta medida es fácil de implementar dentro del control de posición, puesto que se necesitará primeramente los datos del movimiento a realizar, como el tiempo y número de repeticiones. Esta parada en la ejecución puede ser aprovechada por el paciente para agarrar correctamente el robot, ya que hasta no introducir los datos el robot no se moverá.

Por otro lado, en el control de fuerza al ser un control reactivo al paciente el establecimiento de estas interrupciones será más complicado, sobretodo a la hora de parar la aplicación. En el comienzo, cuando el robot este en una posición adecuada para empezar la ejecución, se pedirá por pantalla una confirmación con el fin de establecer una espera que permita al paciente agarrar el extremo correctamente. Con esta confirmación el robot empezará a ejecutar el programa, por lo que no se podrá volver a pedir una confirmación por teclado con el mismo procedimiento, ya que la ejecución no debe ser interrumpida sin ocasionar un error.

Para solucionar esta problemática, se decidió utilizar la opción de detectar la pulsación de una tecla durante la ejecución con la biblioteca que ofrece Python, llamada **pynput**.

Dentro de esta referencia [21] se nos presentan diversos modos que nos permiten detectar la pulsación del teclado, diferenciándose principalmente en la interrupción o no de la ejecución. En el caso de la aplicación, se buscará comprobar durante el bucle si se cumple la condición de parada, por lo que seleccionaremos un método que nos permita continuar con la ejecución.

Para ello se debe definir, dos funciones que comprobarán cuando la tecla se ha pulsado y soltado. A pesar de que a priori una función que comprobará si se ha pulsado la tecla correspondiente, en la práctica se comprobó como en ocasiones no detectaba el evento correctamente. Por lo tanto, se especificó dos funciones para asociarlas al mismo evento.

La primera función **pulsar** solo nos mencionará por pantalla la tecla pulsada, mientras que la función **suelta** será la que ejecute la condición de parada. Por lo que comprobará la tecla pulsada y si es la tecla 'q', realizará la parada. Para este procedimiento, se utilizará una variable booleana global para informar de la parada a la función principal y, después, se impondrá una parada de 5 segundos, de forma que el control de velocidad establezca las últimas velocidades a 0. Con este paso, se establece un periodo de tiempo para que el robot este parado en la última posición y el paciente pueda alejarse. Pasados los 5 segundos, el control de parada pasará al control en posición de manera que sea fácil posicionarlo en la posición home, finalizando la aplicación.

Por otro lado, en el código de la función principal se debe implementar el objeto que nos permite detectar si se ha pulsado una tecla y asociarlo a las funciones que gestionarán los procedimientos a realizar cuando se pulse con los siguientes comandos:

```
escuchador = kb.Listener(pulsa, suelta)
escuchador.start()
```

Por lo tanto, al realizar el bucle de control se añadirá una nueva condición asociada al objeto creado, que permitirá ejecutar el programa mientras este tenga un valor True. Así, cuando esto suceda, la ejecución desarrollará otro hilo relacionado con la función de parada permitiendo, por un lado, la finalización del control de velocidad con valores 0 para todas las articulaciones y, por otra parte, la ejecución completa de la finalización del programa.

Una de las características que se comprobaron en las primeras puestas en marcha de la aplicación, es que durante la aplicación de un esfuerzo por parte de usuario, los valores de fuerza no son siempre constantes, sino que en ciertas iteraciones cambia bruscamente, lo que producía una perturbación semejante en la velocidad calculada. Esta situación provocaba que el extremo del robot se parará durante el movimiento en ciertos momentos, provocando un movimiento discontinuo que no favorece al control. Esta propiedad se volverá a ver en detalle en el apartado de análisis de resultados.

Con el fin de solucionar dicha problemático, se planteo un sistema de realimentación de la velocidad calculada en el control de admitancia con el fin de mantener, lo más precisamente posible, un valor constante que no provoque discontinuidades. Por lo tanto, se creará una variable que nos permita guardar la velocidad utilizada en la iteración anterior. Esta se utilizará para calcular el error con la velocidad actual resultante. Con este error se calculará una velocidad que se debe pasar como parámetro al controlador cinemático cartesiano. Esta se calculará siguiendo la siguiente fórmula:

$$velocidad_final = velocidad_anterior + \frac{error}{20} \quad (16)$$

De esta forma se consigue mantener un cierto valor constante debido a que la influencia del valor actual es menor, evitando las inestabilidades a costa de no alcanzar automáticamente los valores descritos por el control de admitancia. Es decir, habrá un periodo inicial y final en los que la velocidad debe aumentar acorde a la ecuación, con el objetivo de adecuarse a los valores obtenidos. Por lo tanto, se alcanzarán estos valores después de un periodo de tiempo, convirtiendo al sistema de manera que sea capaz de mantener valores constantes que no se podrán alcanzar inmediatamente. Esta situación será objeto de estudio en el apartado de pruebas.

Otro de las propiedades necesarias en una aplicación de rehabilitación es la inclusión de ciertos límites en el espacio de trabajo del robot, de manera que el usuario sea incapaz de entrar dentro de las singularidades propias del robot. Para este fin, se comprueba en cada iteración del bucle de control el determinante de la Jacobiana, ya que cuando este valor se hace 0 significa que el robot posee una configuración articular que provoca una singularidad. Por lo tanto, a la hora de establecer las velocidades a publicar en el *topic* se comprobará el determinante y, mientras su valor sea mayor a 0.001, el programa podrá ejecutarse con normalidad.

Cabe recordar que esta última medida se ejecutará como última opción posible para asegurar la seguridad del paciente, puesto que existe la característica dentro del control de admitancia que permite el control de velocidades cartesianas resultantes con el fin de disminuirlas en función del valor del determinante de la jacobiana.

6. Experimentación

En este apartado, tras explicar todo lo referente a la implementación de la aplicación creada, se procederá a registrar y comentar las diferentes pruebas realizadas a los dos controles principales que usará el brazo robótico, es decir, el control en posición y el control de fuerza.

6.1. Pruebas del control de posición

Al tratarse de un control en el que se especifica posiciones concretas para que el robot se dirija a ellas, la comprobación del correcto funcionamiento de esta parte de la aplicación se debe realizar verificando cómo la primera implementación virtual se materializa en una rutina en un entorno real.

Por consiguiente, se realizará una serie de rutinas que cambiarán los diferentes aspectos configurables de las mismas, a fin de comprobar la adaptabilidad a ellos, aunque todas ellas tendrán en común que existirá un ejercicio lineal y otro circular.

En primer lugar, se desea probar la capacidad del programa para realizar los mismos movimientos con tiempos de realización diferentes, es decir, cambiarán la velocidad a la que se dirigirán al punto destino. Por lo tanto, se realizarán las siguientes rutinas que recogen un mismo movimiento lineal en el eje Z y un movimiento circular de 90 grados:

Ejercicio lineal		**Ejercicio circular**			
Repeticiones	Tiempo(s)	Repeticiones	Tiempo(s)	Radio(m)	Ángulo(º)
2	2	2	2	0.3	90
2	4	2	4	0.3	90
2	6	2	6	0.3	90

El siguiente procedimiento estará destinado a evaluar cambios en las trayectorias de los movimientos preestablecidos. En el caso del movimiento linear, se probarán la introducción de otros puntos que definan la trayectoria a fin de probar movimientos en otros ejes cartesianos.

Por otra parte, el movimiento circular se cambiarán las variables de radio y ángulo a fin de establecer unos límites de movimiento. Por consiguiente, las nuevas rutinas del movimiento lineal serían:

Ejercicio lineal				
Ejes	Punto 1 (m)	Punto 2 (m)	Repeticiones	Tiempo (s)
X	[0.03, 0.45, 0.4]	[-0.23, 0.45, 0.4]	2	2
Y	[-0.11, 0.2, 0.3]	[-0.11, 0.5, 0.3]	2	2
X-Y	[0.2, 0.4, 0.4]	[-0.2, 0.25, 0.4]	2	2
X-Z	[0.1, 0.45, 0.4]	[-0.2, 0.45, 0.1]	2	2
Y-Z	[-0.11, 0.2, 0.4]	[-0.11, 0.4, 0.2]	2	2

Mientras, los cambios elegidos para las pruebas referidas a los ejercicios que engloben los movimientos circulares se recogerán en la tabla que se muestra a continuación:

Ejercicio circular			
Radio (m)	Ángulo (º)	Repeticiones	Tiempo (s)
0.40	30	2	2
0.35	60	2	2
0.25	120	2	2
0.20	150	2	2
0.20	180	2	2

Estas serán las pruebas que se realizarán para el control de posición y cuyos resultados se pueden ver en el apartado siguiente referido al análisis de los mismos.

6.2. Pruebas del control de fuerza

En lo referente al control de fuerza, se decidió que antes de comprobar la aplicación completa del mismo, se verificaran sus componentes. Por tanto, el control de admitancia se procederá a verificar la coincidencia de la dirección entre los incrementos de fuerza y la velocidad calculada. Es importante destacar que no comprobamos si los valores dados son correctos o no, ya que para ello tendremos que tener en cuenta la aplicación completa, es decir, añadiendo el control cinemático cartesiano.

Con esta prueba se buscará que el signo de la velocidad sea acorde al incremento de fuerza y, al mismo tiempo, describir una cierta proporcionalidad entre los resultados.

En lo referente al control en velocidad se estudiará que sea capaz de establecer las velocidades deseadas impuestas. Para ello, la primera parte de la experimentación del control de velocidad tratará de visualizar la influencia del cambio del valor de la velocidad en uno de los ejes cartesianos del extremo del robot. De esta manera se pretende comprobar si la imposición se realiza de manera efectiva, diferenciándose claramente la utilización de distintos valores de velocidad, al mismo tiempo que su valor permanece constante a lo largo de la ejecución. Por lo tanto, se utilizará un movimiento en el eje Y con distintos estimaciones de la velocidad, todos ellos positivos.

Por otro lado, es conveniente verificar que al establecer valores fijos en las tres componentes de la velocidad, estableciendo dos a 0 y el restante con un valor positivo, el funcionamiento del controlador no se ve afectado, puesto que debe ser capaz de mantener una dirección fija en un eje sin presentar desviaciones graves.

Cabe destacar que en el eje X se ha utilizado una velocidad negativa, puesto que debido a la posición de inicio de las pruebas las velocidades positivas en el eje X hacen que el robot entrara dentro de una singularidad. A pesar de este cambio, la idea principal de la prueba no cambia, ya que dentro del programa las componentes mayores obtendrán mayor influencia en la dirección del extremo del robot.

Al ser probadas por separado las dos partes principales del programa, a continuación debe realizarse pruebas al conjunto total con el fin de comprobar la coordinación y correcto funcionamiento del control de fuerza. Además, en estas pruebas entrarán en servicio aquellas partes del código referentes al bucle que gestiona la aplicación y adecúan el funcionamiento de ambas partes.

En esta prueba se aplicará distintas fuerzas para ver la adecuación del sistema, cambiando la intensidad de la fuerza aplicada o la dirección. De esta manera se conseguirán una serie de velocidades para definir la respuesta actual del control de admitancia y la velocidad final que se mandarán al robot. Estas velocidades aunque no sean las articulares que se aplican al robot, se utilizan en el análisis al darnos una visión más clara de la trayectoria del extremo.

7. Análisis de los resultados

Partiendo de las pruebas explicadas anteriormente, en este apartado se recogerán los diferentes resultados de las mismas a fin de valorar el rendimiento de los controles implementados y, por consiguiente, la validez de la prueba así como sus posibles limitaciones actuales.

7.1. Rendimiento del control de posición

En este caso, se decidió recoger las pruebas realizadas en un vídeo, donde se comparará la ejecución de la versión virtual del brazo robótico y la implementación real. De esta forma, se podrá visualizar la precisión y la adecuación de la rutina en ambos medios, por lo que a continuación se presenta un enlace al vídeo comparativo realizado:

Enlace vídeo

Es importante destacar, antes de proceder al análisis de las partes del vídeo, que los vídeos grabados dentro del entorno virtual muestran una demora en la ejecución de los movimientos. Esto es debido a la carga de trabajo que supone para la máquina virtual el proceso de realización, junto con la grabación del mismo. Sin embargo, en la aplicación real no se produce este efecto comentado.

Como se puede observar, el vídeo que recoge las pruebas está dividido en los tres principales grupos de prueba. El primero de ellos, referente a la realización de una rutina básica con diferentes tiempos de ejecución, se puede verificar claramente como las trayectorias calculadas no cambian a pesar del tiempo, sin embargo, se puede destacar que debido al rendimiento del ordenador, en el entorno virtual los tiempos se pueden ver afectados a modo de retrasos respecto al entorno real.

En la segunda parte de la experimentación se comprueba cómo el robot es capaz de moverse dentro de su espacio de trabajo con trayectorias distintas. En concreto, se comprueba que se pueden utilizar comandos para mover el extremo en los tres ejes cartesianos y, además, se pueden crear trayectorias que representen un cambio de posición en varios ejes a la vez, incrementando las posibilidades de nuevas rutinas de movimiento para los ejercicios de rehabilitación.

Por último, los experimentos del tercer conjunto son aplicados a los movimientos circulares que varían sus radios y ángulos. Es importante destacar que estos recorridos dependen del espacio de trabajo, por lo que en estas pruebas al fijarse en una posición concreta del eje Y, el ángulo de las trayectorias creadas será mayor cuando más pequeño sea el radio, para permanecer dentro del espacio de trabajo.

Otros datos importantes dentro de los vídeos, es que se puede ver como en el primer apartados pruebas se puede utilizar la rutina creada para hacer ejercicios de rehabilitación con movimientos de flexión/extensión o abducción/aducción. Asimismo, en la última parte se pueden observar una serie de errores que se suceden dentro del entorno virtual, es decir, sin causar errores permanentes en el robot real. En ellos se puede observar las consecuencias de salirse del espacio de trabajo durante la ejecución. También puede suceder que el robot no realice el movimiento al implementar un punto objetivo inalcanzable.

Como conclusión de estas pruebas realizadas se puede verificar que el funcionamiento trasladado desde el entorno virtual al robot real desempeña las trayectorias de la misma manera y, por tanto, se podrían planificar las trayectorias de las rutinas de ejercicios dentro de un entorno seguro y aplicarlos en una situación real.

7.2. Rendimiento del control de fuerza

Para el control de fuerza se han realizado diferentes pruebas de las partes del programa, por lo que se empezará debatiendo los resultados del control de admitancia.

En ellas se han obtenido una serie de gráficas que muestran el valor de la velocidad obtenidas, gracias a la aplicación de esfuerzo sobre el extremo, mostrado desde el punto de vista del eje de coordenadas del extremo y el de la base.

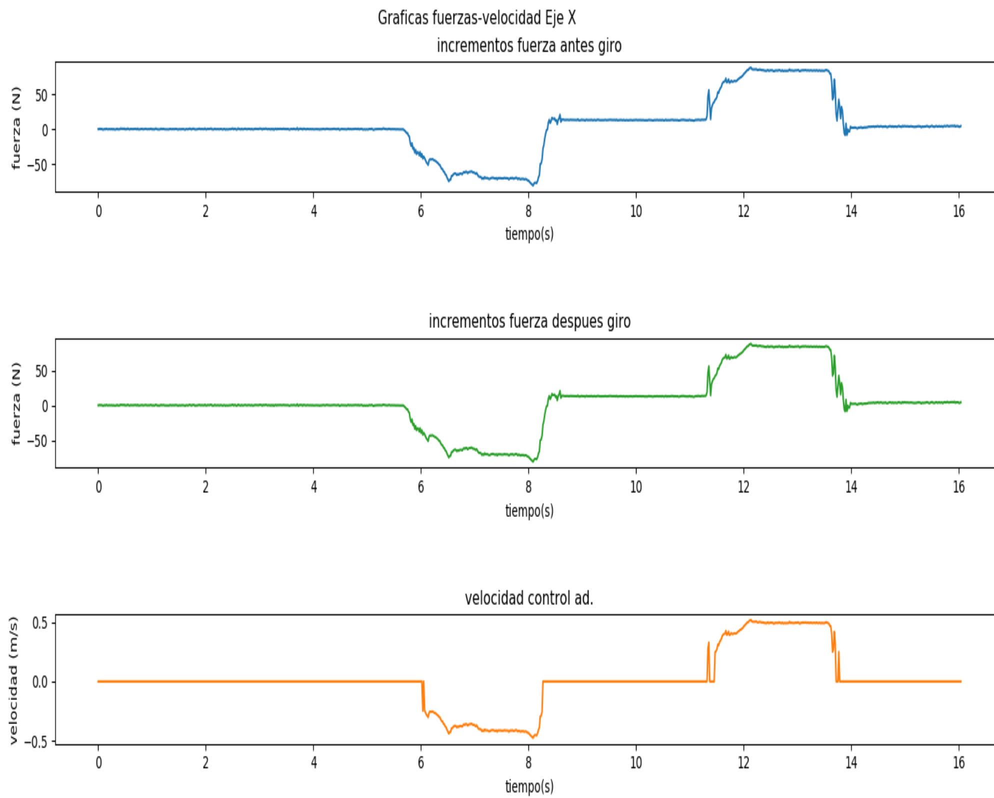


Figura 29: Gráfica fuerzas-velocidad de la primera componente. Fuente: elaboración propia

En este caso en ambos sistemas los valores se representan en el eje X, por lo que serán idénticos. Se puede observar como al no haber movimiento del extremo el valor fuerza permanece cercano a un mismo valor, mientras se aplique fuerza. Este suceso no debería producirse en el sistema completo puesto que el movimiento del extremo compensará el esfuerzo externo. Por otra parte, si nos centramos en la segunda y tercera gráficas, las verdaderamente utilizadas en el algoritmo, podemos destacar una similitud en sus evoluciones, lo que significará que las velocidades calculadas podrán responder correctamente al esfuerzo del paciente. En la siguiente gráfica se muestra la respuesta del Eje Y, en referencia al sistema de coordenadas de la base:

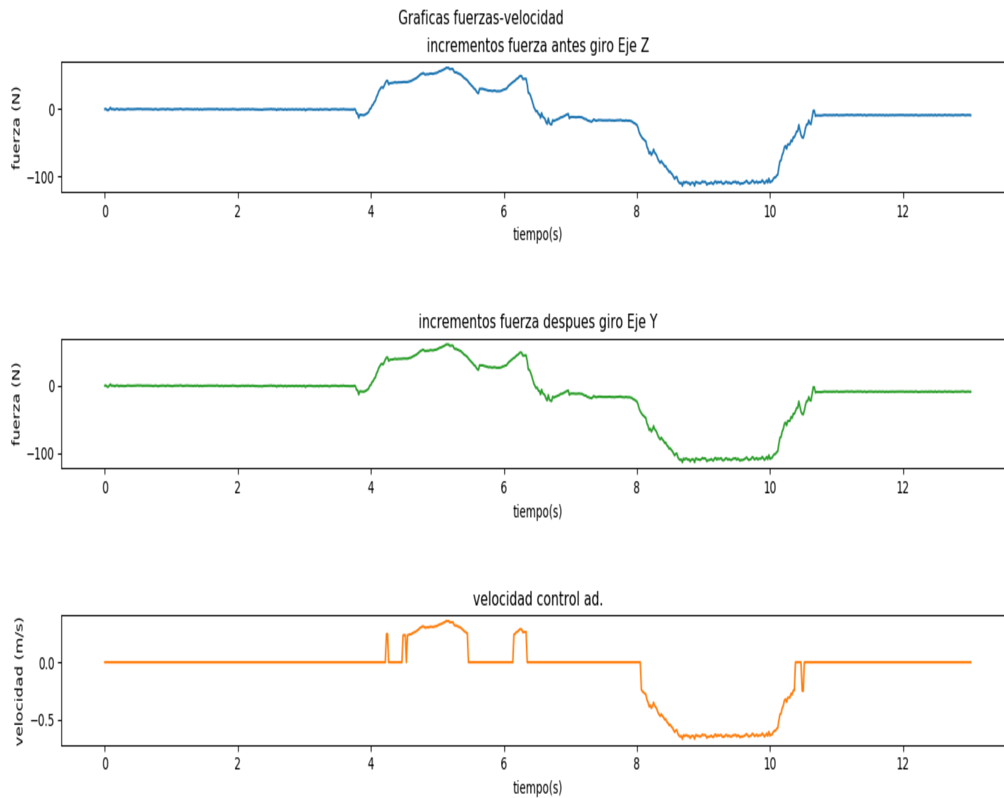


Figura 30: Gráfica fuerzas-velocidad de la segunda componente. Fuente: elaboración propia

Como se mencionó anteriormente, el giro realizado entre los sistemas de coordenadas hace coincidir el eje Z del sistema del extremo con el eje Y de la base. Por lo tanto, el giro será necesario para obtener una respuesta efectiva en el sistema que gobierna el movimiento del robot, es decir el de la base. La respuesta, como se puede verificar, es consecuente al valor del incremento de fuerza, por consiguiente, en la primera curva se visualiza un valle en mitad de la misma con valor 0, puesto que el valor de incremento no supera el límite inferior impuesto a 40 N.

Por último, se muestra la evolución obtenida en el Eje Z, respecto al sistema de coordenadas de la base:

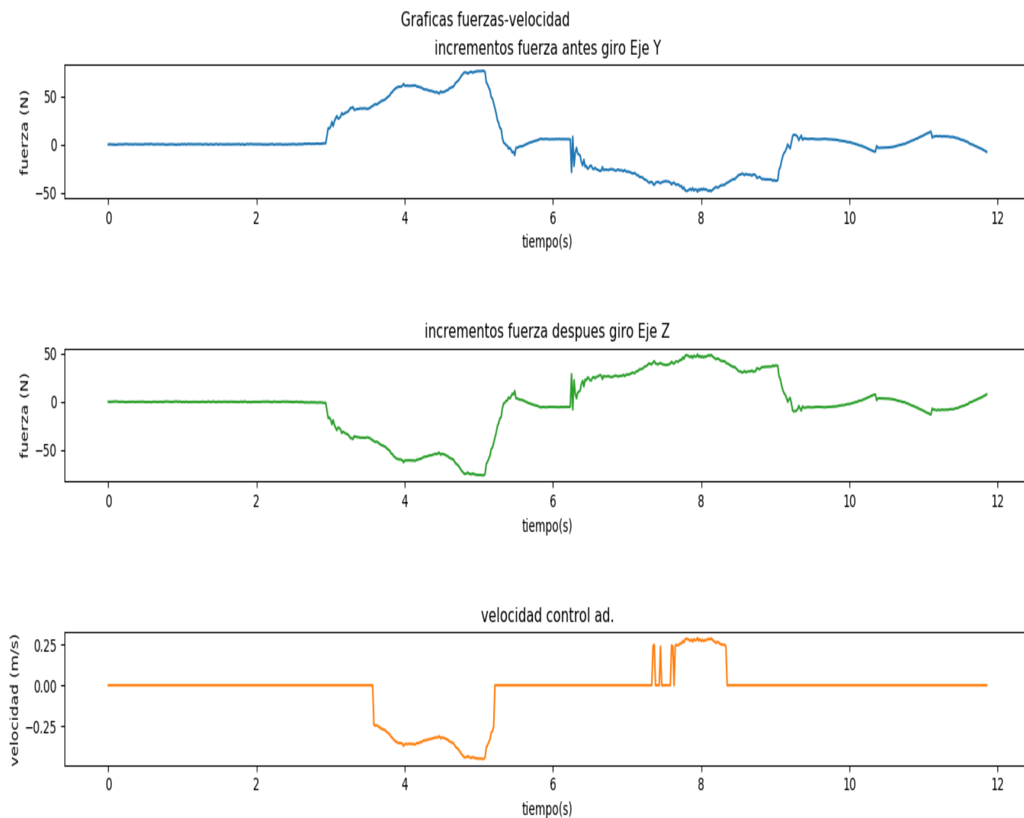


Figura 31: Gráfica fuerzas-velocidad de la tercera componente. Fuente: elaboración propia

En esta tercera componente es donde se puede ver la consecuencia del giro realizado, ya que el valor dado en el Eje Y del sistema del extremo, resultará en el contrario respecto al Eje Z de la base. Nuevamente, con el giro efectuado se puede obtener valores de velocidad válidos para mover el brazo robótico. Además, los incrementos pequeños no provocan cambios en el movimiento, previniendo al sistema de incrementos que no representan una intención real por parte del paciente.

En segundo lugar, se presentan la pruebas realizadas al control cinemático cartesiano. Por lo tanto, la primera experimentación recoge la evolución de la posición del extremo en el eje Y, siguiendo una trayectoria lineal con velocidades distintas.

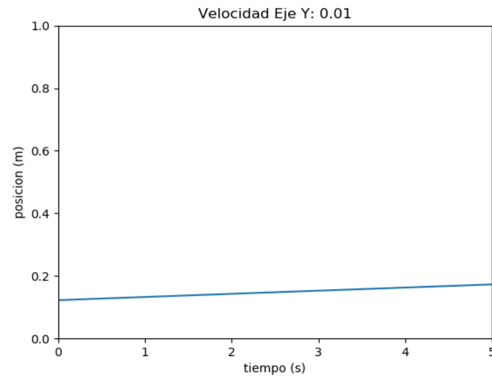


Figura 32: Gráfica posición-tiempo con velocidad 0.01m/s. Fuente: elaboración propia

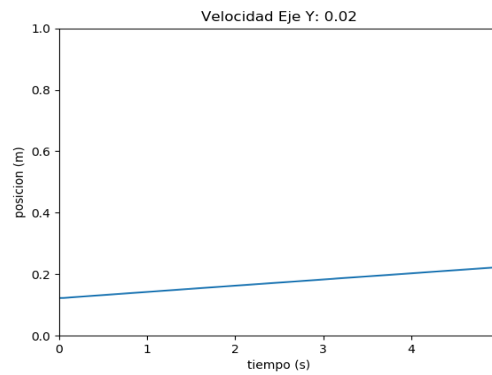


Figura 33: Gráfica posición-tiempo con velocidad 0.02m/s. Fuente: elaboración propia

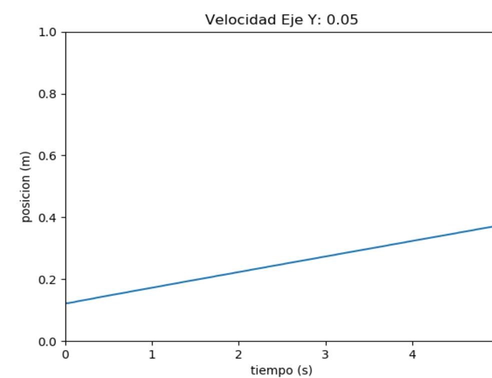


Figura 34: Gráfica posición-tiempo con velocidad 0.05m/s. Fuente: elaboración propia

Como se puede comprobar, se establece claramente una relación lineal entre ambas variables, indicando la correcta implementación de la velocidad que permanece constante, siempre que se tenga en consideración que el robot debe operar en su espacio de trabajo. Asimismo, a medida que se establece un valor más alto de la velocidad, se puede observar cómo la pendiente de la gráfica aumenta en consecuencia.

La siguiente prueba realizada tiene el objetivo de ver la influencia que tiene el movimiento ejecutado en un solo eje para el resto con velocidades nulas. Por consiguiente y a modo de visualización, se muestran las gráficas que recogen dicha experimentación.

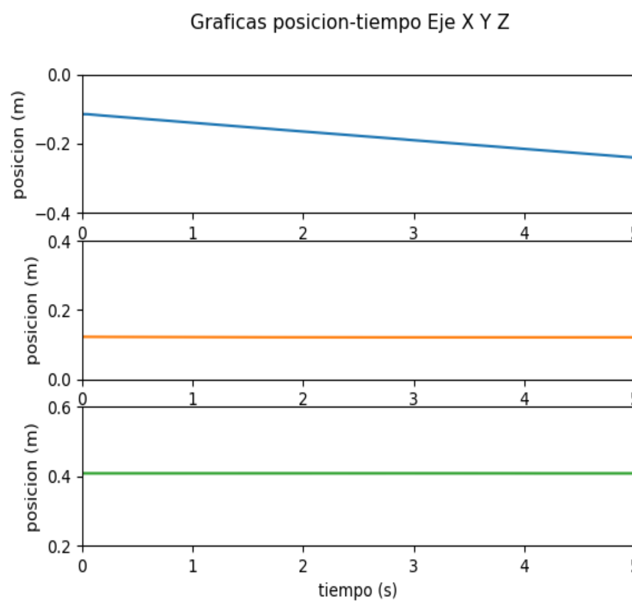


Figura 35: Gráficas posición-tiempo con avance en el eje X. Fuente: elaboración propia

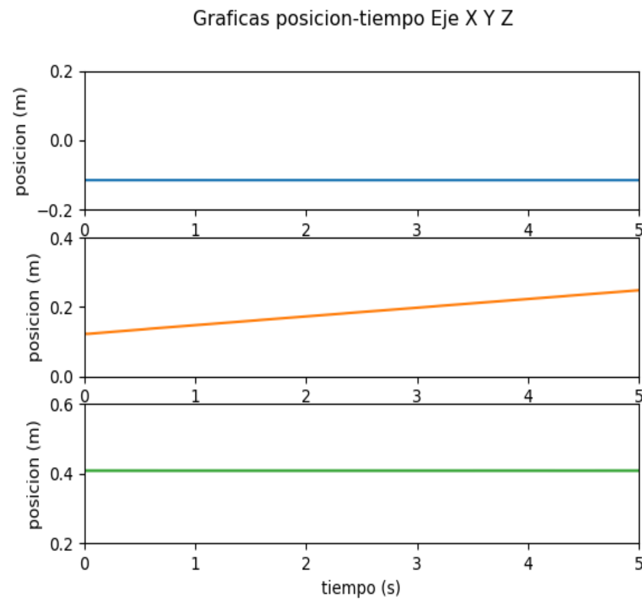


Figura 36: Gráficas posición-tiempo con avance en el eje Y. Fuente: elaboración propia

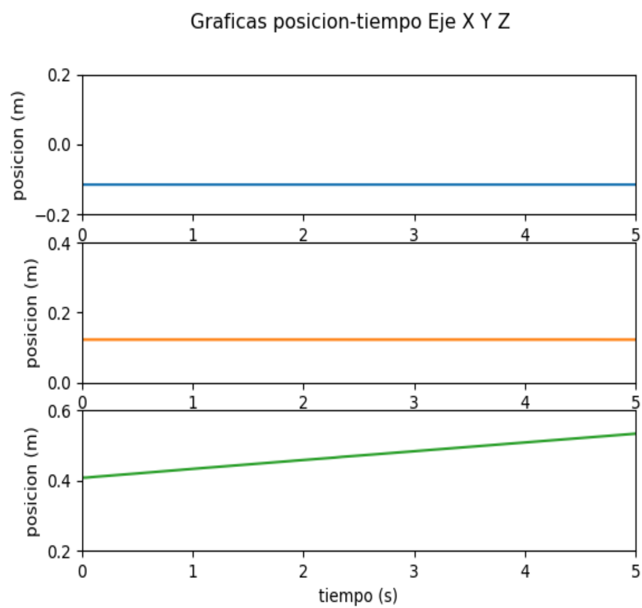


Figura 37: Gráficas posición-tiempo con avance en el eje Z. Fuente: elaboración propia

Con la ayuda de las gráficas se puede observar que la imposición de velocidad en uno de los ejes funciona correctamente, puesto que el cambio de la posición solo se produce en el eje en el que previamente se ha impuesto la velocidad. Mientras, los demás ejes, con valores de velocidad 0, no se ha producido cambios notables de la posición y es por estas razones por lo que podemos determinar una implementación funcional y correcta del control cinemático cartesiano, parte esencial en esta parte del proyecto. El siguiente paso es comprobar el funcionamiento en el control de fuerza, como parte que controlará el movimiento del robot.

En lo referente al control de fuerza completo, es necesario comprobar una serie de factores que complementan su funcionamiento. El primero de ellos es referido a la evolución del determinante de la matriz Jacobiana que representa al robot y el cálculo en consecuencia del coeficiente de amortiguamiento.

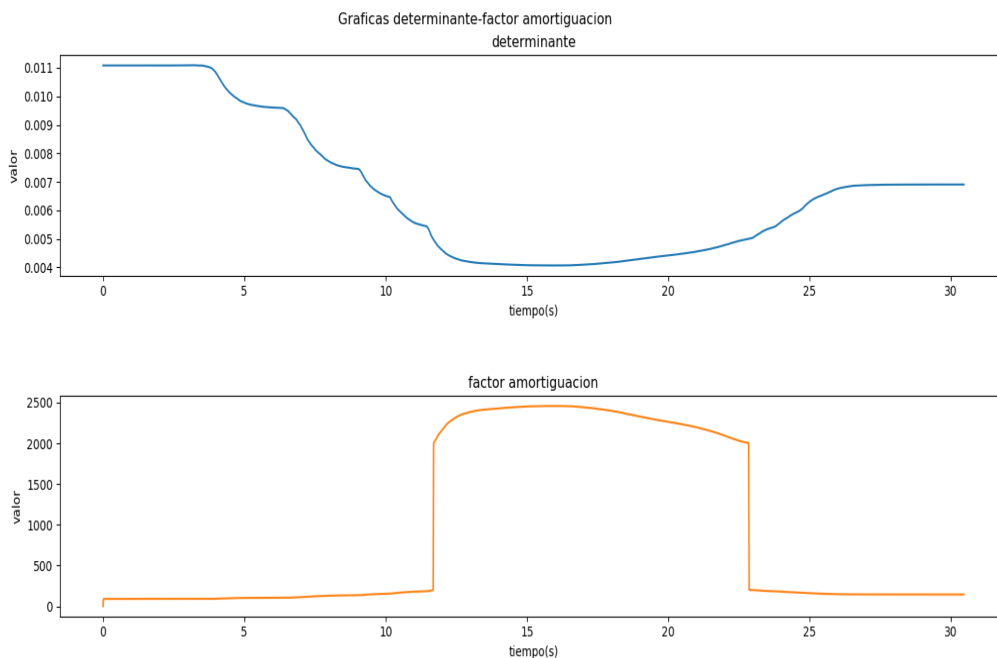


Figura 38: Gráficas determinante - factor amortiguamiento. Fuente elaboración propia

En la evolución de la gráfica, representada en la figura anterior, se ha querido mostrar la respuesta dada cuando el usuario alcanza una situación comprometida cerca de una singularidad. Como se puede observar, el factor de amortiguamiento es mínimo y estable dentro de unos valores seguros del determinante, sin embargo, al sobrepasar el límite es imperativo obstruir el avance del usuario, mediante un aumento significativo del coeficiente. De este modo, la continuación del movimiento en la misma dirección será complicado y, por tanto, al volver a un entorno seguro del espacio de trabajo del robot, los valores vuelven a la situación normal.

Con esta prueba podemos verificar el funcionamiento de este primer prototipo de sistema de prevención para evitar movimientos peligrosos e inesperados por parte del robot. No obstante, es importante destacar que si se siguiera en la dirección, a pesar del incremento del coeficiente de amortiguamiento, el programa detectaría la disminución del determinante hasta su valor límite, momento en el que la aplicación se detendría.

La siguiente prueba es necesaria para establecer el control real de la aplicación sobre la velocidad final que se mandará al robot. Para ello, el procedimiento será variar la influencia de la velocidad calculada a través del control de admitancia. La fórmula que manejaba esta parte de la aplicación es la número

Por lo tanto, se cambiará el valor inicial del denominador de la fracción para comprobar su efecto en el movimiento final del robot, empezando por comprobar el valor actual.

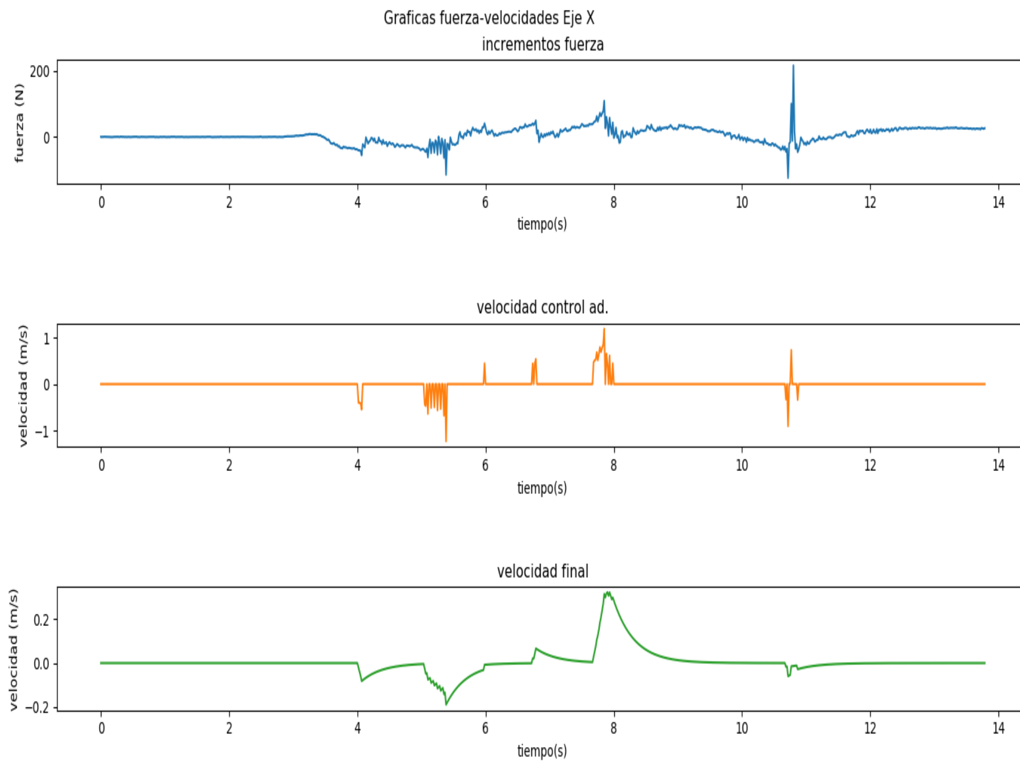


Figura 39: Gráfica fuerza-velocidades con una influencia 1/20. Fuente elaboración propia

Como se ha dedujo anteriormente, al incluir el movimiento del robot al análisis de fuerza, se puede comprobar que estos se mantienen menos estables, ya que son compensados por el movimiento del robot. Es por ello que los valores de velocidad calculados no permanecen estables y funcionan de forma parecida a impulsos.

Esta será una de las razones principales por las que se incluyó el subsistema para valorar la variación de la velocidad calculada por el control de admitancia. De esta forma, se dota al robot de cierta estabilidad de movimiento, sin embargo al ser un espacio de trabajo reducido, los movimientos no son muy amplios y no se puede comprobar un movimiento continuado. En las condiciones actuales, la fuerza detectada influye durante el principio del movimiento y frena progresivamente.

Por consiguiente, el subsistema de control de la velocidad final del robot debe adaptarse para no provocar velocidades excesivas que provoquen falsas mediciones de las fuerzas del extremo y no resulte en un error del programa. Por otro lado, este nos permitirá crear una estabilidad en función de las velocidades obtenidas en el control de admitancia, permitiendo evitar movimientos erróneos como el que se puede observar en la última parte de la gráfica anterior, donde se obtuvo un valor inusual.

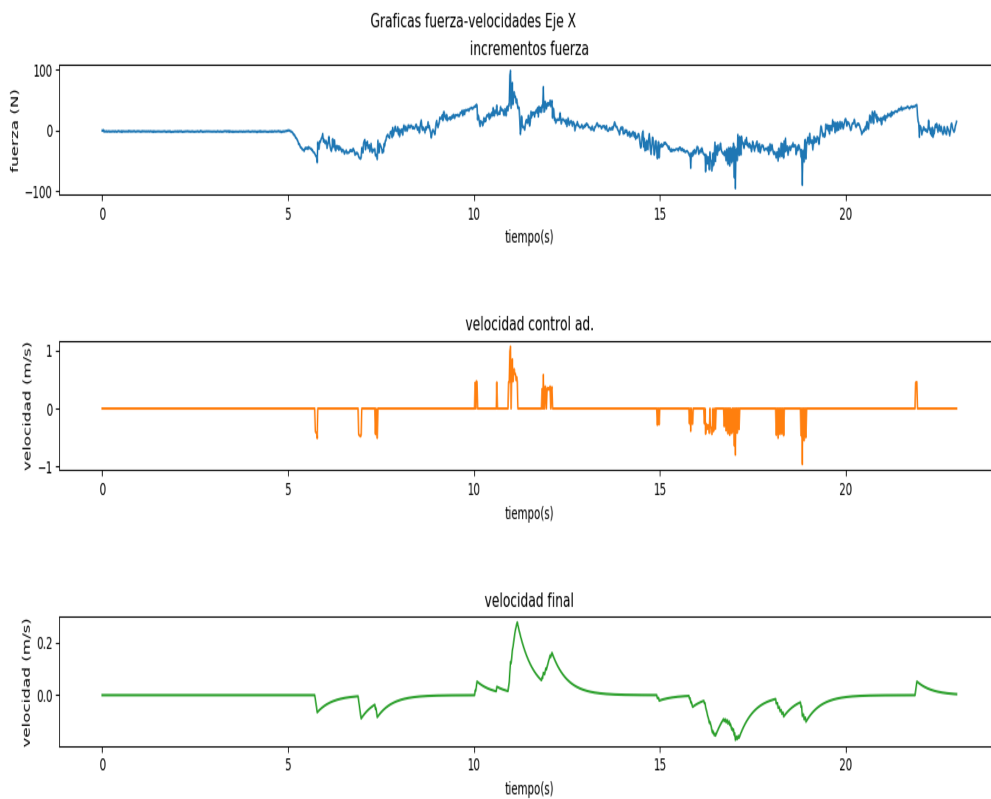


Figura 40: Gráfica fuerza-velocidades con una influencia 1/25. Fuente: elaboración propia

Disminuyendo la influencia, se puede observar como el valor de incrementos de fuerza no disminuye tan rápido, puesto que la compensación por parte del movimiento disminuye. Eso permite detecta mejor las intenciones del usuario al no ser compensadas tan rápido. Por otra parte, es posible que la disminución de la velocidad sea aún rápida, necesitando un apoyo constante del usuario, por lo que sería conveniente observar el resultado de influencias menores en las que el usuario no necesitaría mantener tanto el esfuerzo para realizar el movimiento.

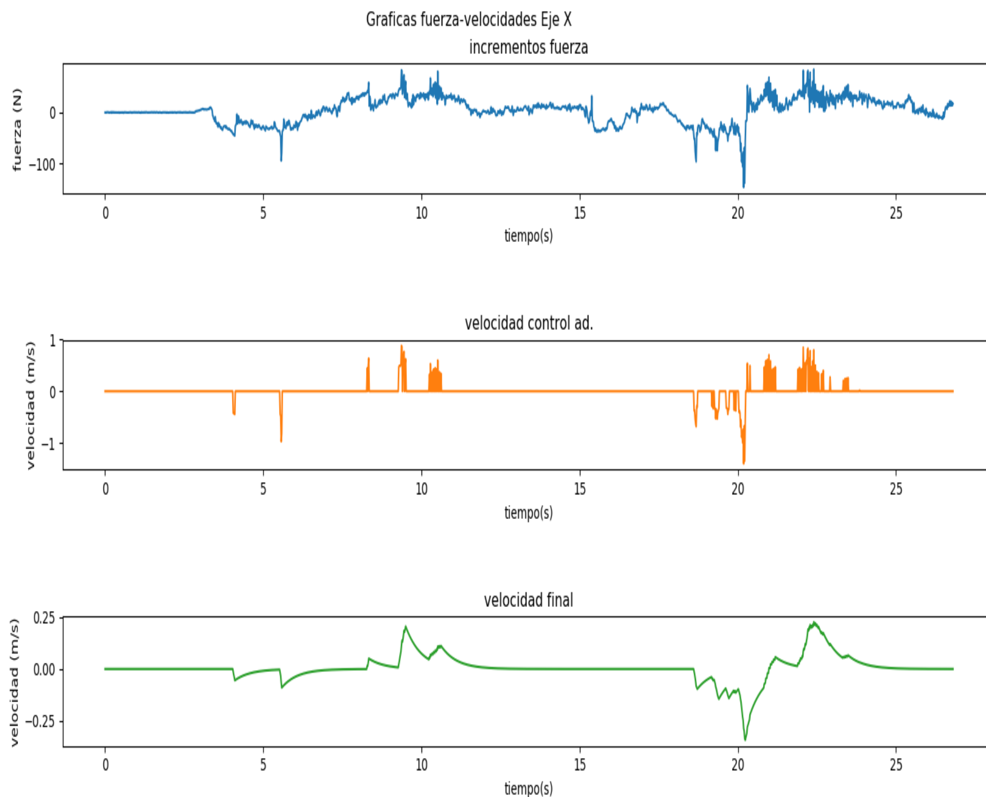


Figura 41: Gráfica fuerza-velocidades con una influencia 1/30. Fuente: elaboración propia

Con la influencia actual se puede observar todavía que los valores detectados de los incrementos de fuerza son más estables, permitiendo la detección del programa de una manera más eficiente.

Sin embargo, las gráficas de velocidad demuestran aún un movimiento discontinuo que puede ser un problema para el usuario, que sentirá torpe el control a la hora de realizar sus movimientos de rehabilitación, por lo que se deberá probar con valores que disminuyan más este valor de influencia.

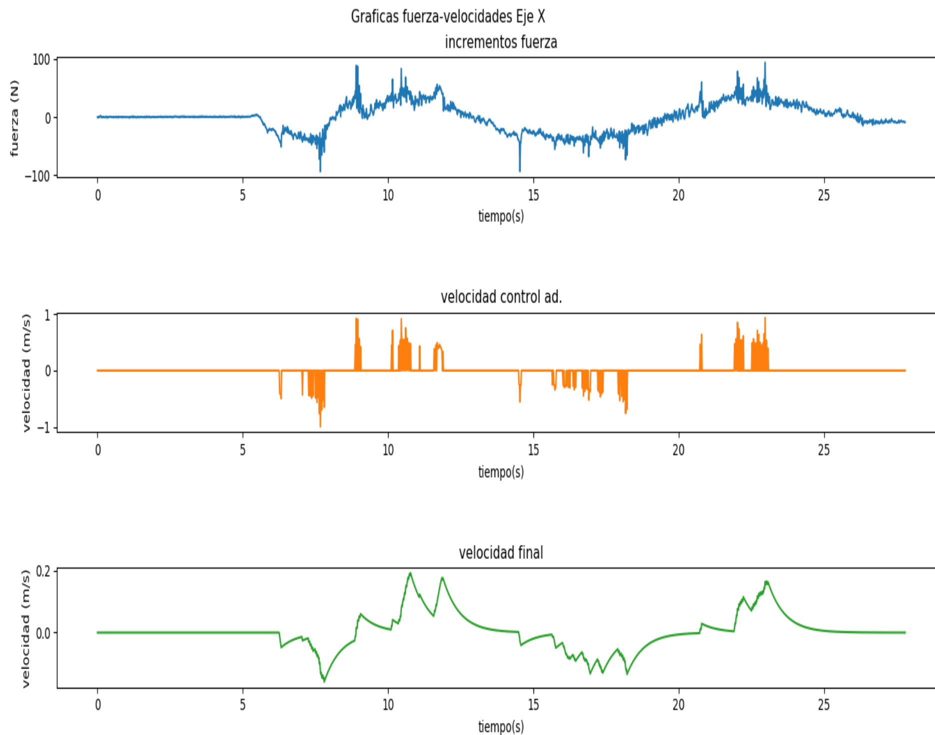


Figura 42: Gráfica fuerza-velocidades con una influencia 1/35. Fuente: elaboración propia

En este caso, la evolución del incremento de fuerzas se estabiliza de una forma que los controladores pueden trabajar correctamente y, en consecuencia, el movimiento es más suave en los movimientos de las pruebas. Es esta interfaz la que podría ser usada en una aplicación real, puesto que la velocidad de control obtiene valores adecuados para el avance de la herramienta. Sin embargo, la curva final del movimiento puede resultar prolongada y provocar un exceso de deslizamiento.

Para resaltar más el efecto de deslizamiento comentado, decidió disminuir, todavía más, la influencia, como se puede ver en la siguiente gráfica:

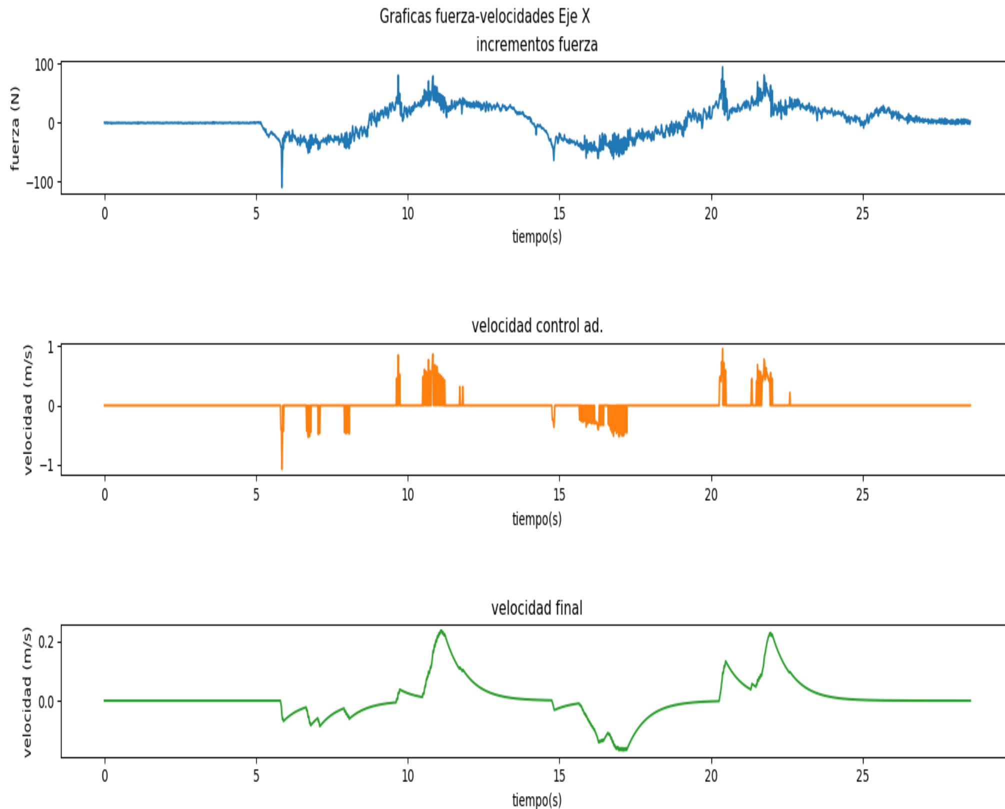


Figura 43: Gráfica fuerza-velocidades con una influencia 1/40. Fuente: elaboración propia

En este caso, al ser tan baja la influencia, la actualización respecto a los valores de la velocidad calculada por el control de admitancia es lenta y resultará en un problema para el control, ya que no se adaptará a los cambios del paciente con rapidez. Un efecto más de esta idea, se puede observar en que se empiezan a obtener valores contantes en el control de admitancia, consecuencia que ocurría cuando el extremo del robot permanecía quieto. Esto quiere decir, que la actualización puede ser lo suficientemente lenta como para detectar durante un periodo de tiempo la intención del usuario sin llegar a responder, produciendo un retardo.

Por último, se decidió comprobar un último valor de influencia entre 1/30 y 1/35 para encontrar un punto medio entre esas dos implementaciones.

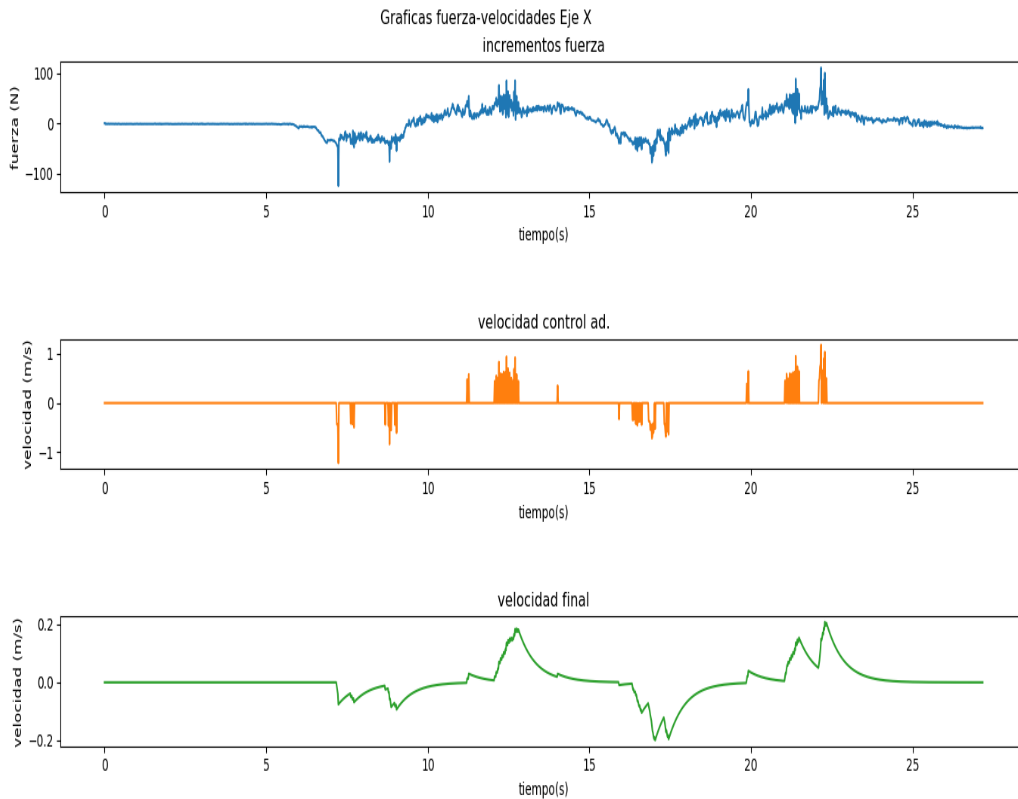


Figura 44: Gráfica fuerza-velocidades con una influencia 1/32.5. Fuente: elaboración propia

En este caso, el efecto de deslizamiento es menos frecuente, permitiendo a su vez una mayor adaptación a la necesidad del usuario. En cuanto al control, se siente un control no tan retardado como anteriormente, aunque al tratarse de un primer prototipo, presenta en ciertas ocasiones interrupciones. A mismo tiempo, los datos de fuerza son lo suficientemente estables como para poder realizar el movimiento.

Tras las pruebas realizadas, donde se ha comprobado diferentes valores para la influencia de las velocidades cartesianas, se decidió con el valor 1/32.5.

$$velocidad_final = velocidad_anterior + \frac{error}{32{,}5} \tag{17}$$

Es importante destacar, que esta combinación es la que mejores sensaciones ha producido a la hora de poner en marcha la aplicación y los valores se contrastan con las gráficas. Sin embargo, los resultados pueden variar en el modo en que se opera con la herramienta, ya que aplicar un excesivo esfuerzo no permite un control limpio, por lo que se ha pretendido siempre aplicar una fuerza constante y no excesiva.

Con este último ajuste se podrá comprobar el funcionamiento completo en los tres ejes de movimiento del extremo con las gráficas que se muestran a continuación.

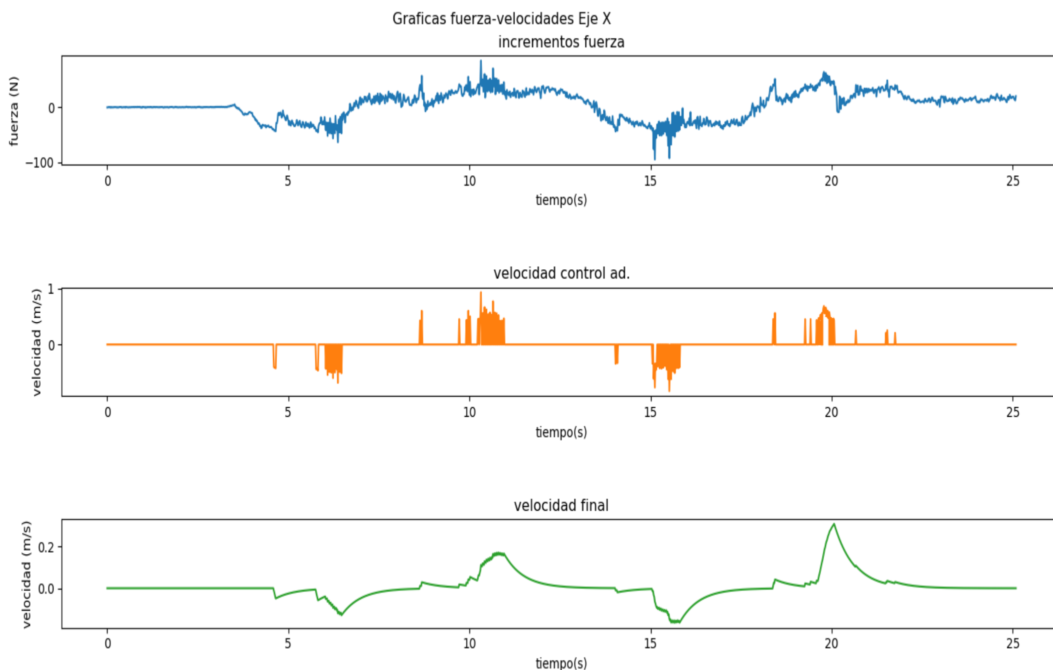


Figura 45: Gráfica fuerza-velocidades en el eje X. Fuente: elaboración propia

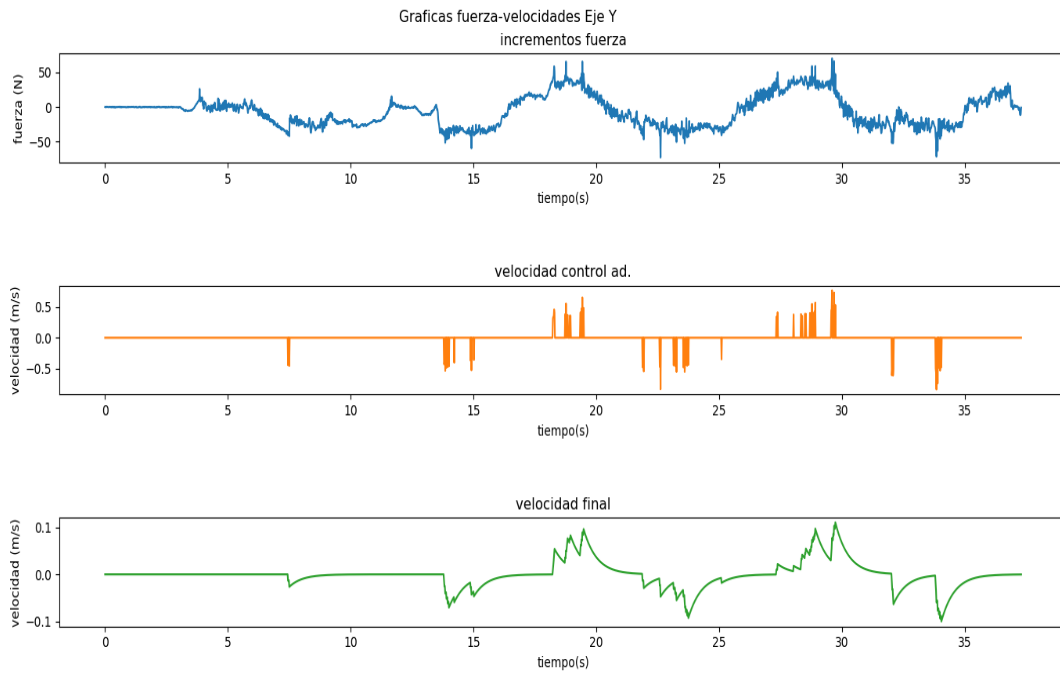


Figura 46: Gráfica fuerza-velocidades en el eje Y. Fuente: elaboración propia

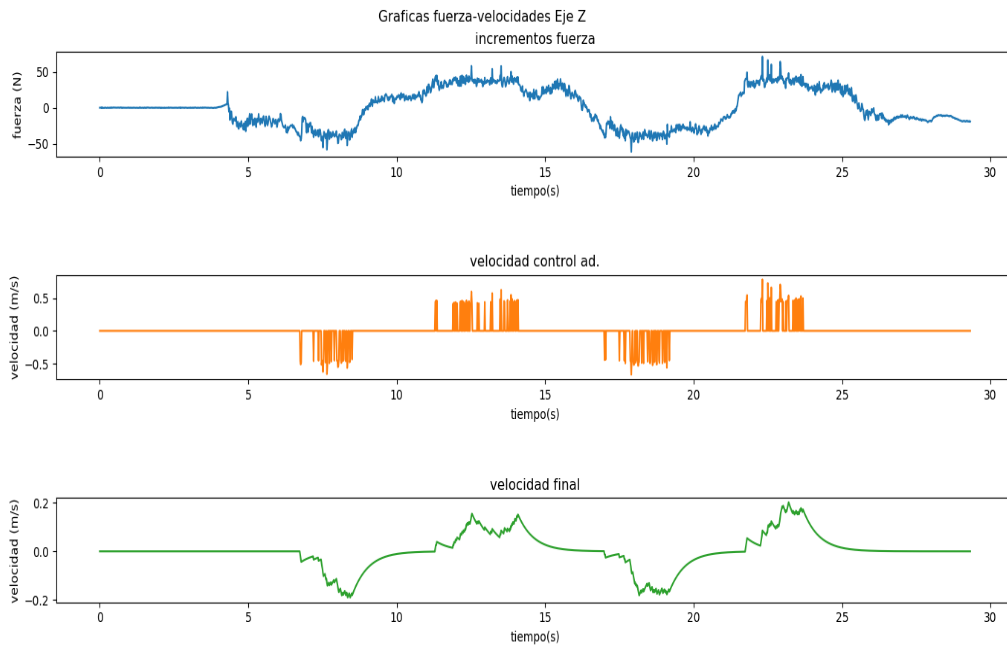


Figura 47: Gráfica fuerza-velocidades en el eje Z. Fuente: elaboración propia

En primer lugar, se puede observar que el movimiento conseguido por parte del usuario es más fluido en los ejes X y Z. Esto puede ser debido a que es más fácil aplicar un esfuerzo en estas direcciones, mientras que en el eje Y resulta más complicado atraer o alejar la herramienta. Otra de las posibles causas puede ser que no se detecte correctamente el esfuerzo en dicho eje, no obstante, el programa depende de las lecturas del sensor interno del robot.

Por otra parte, se consigue una respuesta adecuada a las intenciones del usuario con el fin de acompañar su movimiento. De manera ilustrativa, es importante presentar una ejecución práctica del robot con los últimos ajustes implementados para observar su funcionamiento en un entorno real, de manera que se pueda presentar otro punto de vista de la comparativa. Por lo tanto, en el siguiente enlace se muestran las experimentaciones de movimiento para los tres ejes de coordenadas:

Enlace vídeo

8. Conclusiones

Para terminar el presente proyecto en este último apartado, tras exponer todo el contenido de la aplicación, se expondrán las consecuencias obtenidas a partir de los resultados analizados, además de estudiar los posibles trabajos futuros.

En referencia a los objetivos presentados al principio de la memoria, en primer lugar, podemos asegurar que se consigue una conexión suficiente entre el robot y el ordenador externo, que permitirá la realización de los controladores explicados. Este hecho se consigue gracias a la utilización de la plataforma ROS, junto con el driver comentado, que recoge una serie de técnicas y mecanismos que habilitan la conexión, siguiendo un esquema de *topics* esencial en la plataforma ROS.

El siguiente objetivo es el desarrollo del módulo de control de posición que nos permita realizar rutinas de rehabilitación pasiva. En este caso, se consiguió la implementación completa de un algoritmo capaz de mandar al robot posiciones junto con su tiempo de ejecución, con el fin de controlar la velocidad de movimiento. Las métodos de esta parte permitirán realizar trayectorias lineales y circulares para su adecuación a futuras rutinas de ejercicio. Además, a modo de ejemplo, se planteo una rutina de rehabilitación pasiva para visualizar el alcance de la aplicación.

El último objetivo trata de la creación de un módulo capaz de percibir las fuerzas externas realizadas por el usuario con el objetivo de actuar en consecuencia, a modo de rehabilitación activa. El prototipo creado es capaz de percibir las fuerzas aplicadas en una herramienta externa diseñada y acoplada al extremo del robot y, a partir de estos datos, calcular una velocidad cartesiana que se deberá traducir a articulares, a fin de ser interpretadas por el robot. Sin embargo, es importante destacar que la aplicación en ciertas ocasiones puede presentar discontinuidades y falsas mediciones que afectan al comportamiento general, aunque en general este es estable y permite mover la herramienta de manera adecuada, como se puede ver en el vídeo complementario de esta parte.

Por consiguiente, se han revisado los tres objetivos planteados al principio del proyecto y se consideran cumplidos, sin embargo, es importante destacar las opciones de mejora posibles que se pueden realizar a fin de crear una aplicación completa que considere aspectos que no han sido expuestos en este proyecto. Es por ello, que se presentan a continuación los trabajos futuros planeados.

8.1. Trabajos futuros

En cuanto a los trabajos posibles que podemos destacar que al ser un primer prototipo que verifica la posibilidad de crear los controladores desarrollados dentro de un conexión externa entre un ordenador con ROS y el robot UR3, solo se ha realizado una primera definición sin explorar por completo las posibilidades y mejores del control de fuerza. Es por ello, que la primera proposición recae en el estudio de controles de fuerza más avanzados como los que utilizan el aprendizaje por refuerzo para su adecuación a la plataforma utilizada.

Por otra parte, en la realización de este trabajo el arranque de los diferentes archivos y la introducción de parámetros al programa se ha realizado a través de la terminal de Linux. Por esta razón, se aclara que un importante trabajo a futuro sería la creación de una interfaz gráfica con la que manejar de manera eficiente la aplicación de rehabilitación, eligiendo el control deseado o la introducción más intuitiva de los parámetros de las trayectorias. Asimismo, permitiría una adaptación sencilla por parte del usuario para crear rutinas sin entrar en detalles a bajo nivel del código.

Por otra parte, dentro del trabajo se podría plantear la realización de una combinación entre ambos controles principales, en otras palabras, posición y fuerza. La principal ventaja de este planteamiento es que habilitaría una situación intermedia entre el escenario donde el paciente apenas puede aplicar fuerza al extremo final y en la que es capaz de mover con cierta facilidad el robot. En este punto intermedio, el paciente sería capaz de crear variaciones en la trayectoria del programa, que recalcularía la misma teniendo en cuenta la desviación. En este caso, siguen existiendo una imposición de trayectoria del robot, pero con la posibilidad de ser modificada por el paciente que conseguiría un paso intermedio en su recuperación.

Estas serías las tres líneas principales pensadas con el fin de establecer una continuación del trabajo que desarrollan diferentes partes de la aplicación desarrollada, como es la interfaz o la modificación del control desarrollado.

Bibliografía

[1] U. Costa, "Robótica para la rehabilitación." https://n9.cl/53nsb, 2020.

[2] C. Macao and J. Nacipucha, *Diseño e implementación de un prototipo de exoesqueleto destinado a la rehabilitación de codo*. PhD thesis, Universidad de Cuenca, http://dspace.ucuenca.edu.ec/handle/123456789/25820, 2016.

[3] M. Montenegro, A. Lugo, J. Núñez, and R. González, *Análisis y diseño de un prototipo virtual de tipo exoesqueleto para rehabilitación de codo*. PhD thesis, Facultad de Electromecánica, Universidad de Colima, https://n9.cl/szkan, 2014.

[4] R. Gonçalves, *A Robotic System for Musculoskeletal Rehabilitation of the Shoulder*. PhD thesis, Instituto Superior Técnico, Universidad Pública de Lisboa, 2016.

[5] R. Gassert and V. Dietz, "Rehabilitation robots for the treatment of sensorimotor deficits: a neurophysiological perspective," *Journal of NeuroEngineering and Rehabilitation*, 2016.

[6] F. Molteni, G. Gasperini, G. Cannaviello, and E. Guanziroli, *Exoskeleton and End-Effector Robots for Upper and Lower Limbs Rehabilitation: Narrative Review*. PhD thesis, American Academy of Physical Medicine and Rehabilitation, https://n9.cl/y87in, 2018.

[7] A. Casals, J. Aranda, M. Vinagre, A. Martínez, R. Jané, and J. Amat, *Control de robots en neurorehabilitación. Desde sensores neurales a sensores portables*. PhD thesis, Centro de Investigación en Ingeniería Biomédica, CREB-UPC, https://n9.cl/lxnof, 2014.

[8] L. E. Amigo, Q. Fernandez, X. Giralt, A. Casals, and J. Amant, "Study of patient-orthosis interaction forces in rehabilitation therapies," *The Fourth IEEE RAS/EMBS International Conference on Biomedical Robotics and Biomechatronics*, 2012.

[9] L. de Azebedo, J. Lima, P. Leitao, and A. Yoshiro, *Using a Collaborative Robot to the Upper Limb Rehabilitation*. PhD thesis, Universidade Tecnológica Ferderal do Paraná, Paraná, Brazil, 2019.

[10] L. de Azebedo, T. Brito, L. Piardi, J. Lima, and P. Leitao, *A Real Framework to Apply Collaborative Robots in Upper Limb Rehabilitation*. PhD thesis, Universidade Tecnológica Ferderal do Paraná, Paraná, Brazil, 2020.

[11] Universal Robot, https://www.universal-robots.com/media/240787/ur3_us.pdf, *UR3 Technical specifications*, 2015.

[12] Universal Robot, https://www.cfzcobots.com/wp-content/uploads/2017/03/ur3_user_manual_es_global.pdf, *Manual de usuario*, 2015.

[13] ROS, “About ros.” https://www.ros.org/about-ros/.

[14] J. Playán, *Navegación de robots manipuladores en el entorno de ROS*. PhD thesis, Universidad de Zaragoza – Zaguan, https://zaguan.unizar.es/record/85045/files/TAZ-TFG-2019-3387.pdf, 2019.

[15] “Moveit.” https://moveit.ros.org/, 2021.

[16] “Universal_robots_ros_driver.” https://github.com/UniversalRobots/Universal_Robots_ROS_Driver, 2021.

[17] F. Pucher, “Ros control, an overview.” https://fjp.at/posts/ros/ros-control/, 2020.

[18] V. Duchaine and C. Gosselin, *General Model of Human-Robot Cooperation Using a Novel Velocity Based Variable Impedance Control*. PhD thesis, Universite Laval, Quebec (QC), Canada, 2007.

[19] T. Moulard, D. Lu, K. Hawkins, A. E. Khoury, and E. Cousineau, “urdf_parser_py.” https://github.com/ros/urdf_parser_py, 2020.

[20] K. Hawkins, “pykdl_utils.” http://wiki.ros.org/pykdl_utils, 2013.

[21] J., “Cómo detectar la pulsación de una tecla en python.” https://n9.cl/79i2t, 2021.

Apéndice 1: Código prueba

En este primer apéndice se presenta el código utilizado, mediante el programa MATLAB, en la primera prueba de conexión realizada con el robot:

```
% Establecer conexion
socket_ur3 = tcpip('192.168.2.2', 30003);
fopen(socket_ur3);

% Posiciones importantes
q_home = [0,-pi/2,0,-pi/2,0,0];                    % Home
q_preparacion = [pi/2,-pi/2,pi/2,-pi,-pi/2,0];  % Colocacion
pos_obj = [0.2, -0.3946, 0.1242, 1.57, 0, 0];   % Objetivo 1
pos_obj2 = [0.2, -0.3946, 0.5242, 1.57, 0, 0];  % Objetivo 2

% Habilito lectura
UR3_lectura = UR5_Read(socket_ur3);

% Movimientos
cadena_prep = sprintf('movej([%f, %f, %f, %f, %f, %f],2,4,4,0)',
   q_preparacion);
fprintf(socket_ur3,cadena_prep);

pause (4)

fuerza0 = UR3_lectura.TCP_force
cadena=sprintf('movel(p[%f, %f, %f, %f, %f, %f],2,4,5,0)',
   pos_obj);
fprintf(socket_ur3,cadena);

pause(6)

fuerza1 = UR3_lectura.TCP_force
cadena=sprintf('movel(p[%f, %f, %f, %f, %f, %f],2,4,20,0)',
   pos_obj2);
fprintf(socket_ur3,cadena);

pause(21)

cadena = sprintf('movej([%f, %f, %f, %f, %f, %f],2,4,4,0)',
   q_home);
fprintf(socket_ur3,cadena);
```

Apéndice 2: Código del control de posición

En este apéndice se recoge el algoritmo utilizado en la elaboración del control de posición:

```
#!/usr/bin/env python
# importamos las librerias a utilizar
import sys
import rospy
import moveit_commander
import geometry_msgs.msg
import math
import actionlib
import numpy as np
from control_msgs.msg import *
from trajectory_msgs.msg import *

JOINT_NAMES = ['shoulder_pan_joint', 'shoulder_lift_joint',
    'elbow_joint',
               'wrist_1_joint', 'wrist_2_joint', '
   wrist_3_joint']
client = None

# Obtenemos el grupo del manipulador entero
arm_group = moveit_commander.MoveGroupCommander("
   manipulator")

# FUNCIONES-------------------------------------------------

# Funcion para pasar de grados de euler a cuaternio
def eu_to_q(roll, pitch, yaw):

    qx = np.sin(roll/2) * np.cos(pitch/2) * np.cos(yaw/2) -
    np.cos(roll/2) * np.sin(pitch/2) * np.sin(yaw/2)
    qy = np.cos(roll/2) * np.sin(pitch/2) * np.cos(yaw/2) +
    np.sin(roll/2) * np.cos(pitch/2) * np.sin(yaw/2)
    qz = np.cos(roll/2) * np.cos(pitch/2) * np.sin(yaw/2) -
    np.sin(roll/2) * np.sin(pitch/2) * np.cos(yaw/2)
    qw = np.cos(roll/2) * np.cos(pitch/2) * np.cos(yaw/2) +
    np.sin(roll/2) * np.sin(pitch/2) * np.sin(yaw/2)

    return [qw, qx, qy, qz]
```

```
# Funcion para ir a la posicion home
def home():

    # Situamos el robot en la posicion guardado
    arm_group.set_named_target("up")
    plan = arm_group.go()

# Funcion para mover en el espacio articular
def mover_articular(art1, art2, art3, art4, art5, art6):

    # Guardar la posicion objetivo
    articulaciones = arm_group.get_current_joint_values()
    articulaciones[0] = art1
    articulaciones[1] = art2
    articulaciones[2] = art3
    articulaciones[3] = art4
    articulaciones[4] = art5
    articulaciones[5] = art6

    # Mandar el robot a la posicion
    arm_group.set_joint_value_target(articulaciones)
    plan = arm_group.go()

    # Funcion para mover en el espacio cartesiano
    def mover_cartesiano(x,y,z,roll,pitch,yaw):

    # Obtenemos los cuaternios
    rotacion = eu_to_q(roll, pitch, yaw)

    # Posicion objetivo
    pos_obj = geometry_msgs.msg.Pose()
    pos_obj.orientation.w = rotacion[0]
    pos_obj.orientation.x = rotacion[1]
    pos_obj.orientation.y = rotacion[2]
    pos_obj.orientation.z = rotacion[3]
    pos_obj.position.x = x
    pos_obj.position.y = y
    pos_obj.position.z = z

    # Ejecutamos el movimiento
    arm_group.set_pose_target(pos_obj)
    plan = arm_group.go()
```

```
# Funcion para mover el robot un trayectoria controlada
def mover_controlado(x, y, z, roll, pitch, yaw, duracion):

    # Obtenemos los cuaternios
    rotacion = eu_to_q(roll, pitch, yaw)

    # Guarda posicion
    auxiliar = geometry_msgs.msg.Pose()
    auxiliar.orientation.w = rotacion[0]
    auxiliar.orientation.x = rotacion[1]
    auxiliar.orientation.y = rotacion[2]
    auxiliar.orientation.z = rotacion[3]
    auxiliar.position.x = x
    auxiliar.position.y = y
    auxiliar.position.z = z

    # Obtenemos el espacio articular
    arm_group.set_joint_value_target(auxiliar, "ee_link",
   True)
    pos_art = arm_group.get_joint_value_target()

    # Realizamos la trayectoria con la posicion final
    g = FollowJointTrajectoryGoal()
    g.trajectory = JointTrajectory()
    g.trajectory.joint_names = JOINT_NAMES
    g.trajectory.points = [JointTrajectoryPoint(positions =
    pos_art, velocities = [0]*6, time_from_start = rospy.
   Duration(duracion))]
    client.send_goal(g)
    try:
        client.wait_for_result()
    except KeyboardInterrupt:
        client.cancel_goal()
        raise

# Funcion encargarda de la realizacion de un movimiento
   lineal
def movimiento_lineal(pos_ini, pos_final, duracion):
```

```
    # Obtenemos los cuaternios de cada posicion
    rot1 = eu_to_q(pos_ini[3], pos_ini[4], pos_ini[5])
    rot2 = eu_to_q(pos_final[3], pos_final[4], pos_final
[5])

    # Creamos un objetivo para la action
    g = FollowJointTrajectoryGoal()
    g.trajectory = JointTrajectory()
    g.trajectory.joint_names = JOINT_NAMES

    # Guardamos la posicion actual del robot
    pos_actual = arm_group.get_current_joint_values()

    # Fragmentamos el movimiento para realizar un bucle
precision
    iteraciones = 10
    tiempo = duracion/iteraciones
    for i in range(1,iteraciones+1):
        auxiliar = geometry_msgs.msg.Pose()
        auxiliar.orientation.w = rot1[0] + (rot2[0] - rot1
[0])/iteraciones * i
        auxiliar.orientation.x = rot1[1] + (rot2[1] - rot1
[1])/iteraciones * i
        auxiliar.orientation.y = rot1[2] + (rot2[2] - rot1
[2])/iteraciones * i
        auxiliar.orientation.z = rot1[3] + (rot2[3] - rot1
[3])/iteraciones * i
        auxiliar.position.x = pos_ini[0] + (pos_final[0] -
pos_ini[0])/iteraciones * i
        auxiliar.position.y = pos_ini[1] + (pos_final[1] -
pos_ini[1])/iteraciones * i
        auxiliar.position.z = pos_ini[2] + (pos_final[2] -
pos_ini[2])/iteraciones * i

        # Obtenemos la posicion articular actual y objetivo
        arm_group.set_joint_value_target(auxiliar, "ee_link
", True)
        pos_obj = arm_group.get_joint_value_target()
```

```
        # Obtenemos el vector velocidades para los puntos
    de paso
        vel = []
        for j in range(0,6):
            aux = ((pos_obj[j] - pos_actual[j]) / (duracion
    /iteraciones))
            vel.append(aux)

        # Anadimos el punto a la trayectoria
        if i == iteraciones+1:
            g.trajectory.points.append(JointTrajectoryPoint
    (positions = pos_obj, velocities = [0]*6,
    time_from_start = rospy.Duration(tiempo)))
        else:
            g.trajectory.points.append(JointTrajectoryPoint
    (positions = pos_obj, velocities = vel, time_from_start
    = rospy.Duration(tiempo)))

        # Aumentamos el tiempo de ejecucion
        tiempo = tiempo + duracion/iteraciones

        # Actualizamos la posicion actual
        pos_actual = pos_obj

    # Mandamos el objetivo y esperamos a que se realice la
    accion
    client.send_goal(g)
    try:
        client.wait_for_result()
    except KeyboardInterrupt:
        client.cancel_goal()
        raise

# Funcion para el movimiento circular
def movimiento_circular(pos_ini, radio, angulo, duracion):

    # Creamos la posicion final
    pos_final = []
    pos_final.append(pos_ini[0] + math.sin(angulo)*radio)
    pos_final.append(pos_ini[1])
```

```
 # Dependiendo del angulo de entrada debemos cambiar la
definion
 if (angulo > 0):
     pos_final.append(pos_ini[2] + radio * (1 - math.cos
(angulo)))
 else:
     pos_final.append(pos_ini[2] - radio * (1 - math.cos
(angulo)))

 pos_final.append(pos_ini[3])
 pos_final.append(pos_ini[4])
 pos_final.append(pos_ini[5])

 # Obtenemos los cuaternios de la rotacion
 rot = eu_to_q(pos_ini[3], pos_ini[4], pos_ini[5])

 # Creamos un objetivo para la action
 g = FollowJointTrajectoryGoal()
 g.trajectory = JointTrajectory()
 g.trajectory.joint_names = JOINT_NAMES

 # Guardamos la posicion actual del robot
 pos_actual = arm_group.get_current_joint_values()

 # Fragmentamos el movimiento para realizar un bucle
precision
 iteraciones = 10
 tiempo = duracion/iteraciones
 for i in range(1,iteraciones+1):

     # Creamos el punto auxiliar
     auxiliar = geometry_msgs.msg.Pose()
     auxiliar.orientation.w = rot[0]
     auxiliar.orientation.x = rot[1]
     auxiliar.orientation.y = rot[2]
     auxiliar.orientation.z = rot[3]

     # La coordenada y siempre sera la misma
     auxiliar.position.y = pos_ini[1] + (pos_final[1] -
pos_ini[1])/iteraciones * i
```

```
        # Dependiendo del angulo cambiamos las definiciones
        if (angulo > 0):
            auxiliar.position.x = pos_ini[0] + radio * math
.sin(angulo/iteraciones* i)
            auxiliar.position.z = pos_ini[2] + radio * (1 -
 math.cos(angulo/iteraciones * i))
        else:
            auxiliar.position.x = pos_ini[0] + radio * (
math.sin(-angulo + angulo/iteraciones*i) - math.sin(-
angulo))
            auxiliar.position.z = pos_ini[2] + radio * ((1
- math.cos(-angulo + angulo/iteraciones * i)) - (1 -
math.cos(-angulo)))

        # Obtenemos la posicion articular actual y objetivo
        arm_group.set_joint_value_target(auxiliar, "ee_link
", True)
        pos_obj = arm_group.get_joint_value_target()

        # Obtenemos el vector velocidades para los puntos
de paso
        vel = []
        for j in range(0,6):
            aux = ((pos_obj[j] - pos_actual[j]) / (duracion
/iteraciones))
            vel.append(aux)

        # Anadimos el punto a la trayectoria
        if i == iteraciones+1:
            g.trajectory.points.append(JointTrajectoryPoint
(positions = pos_obj, velocities = [0]*6,
time_from_start = rospy.Duration(tiempo)))
        else:
            g.trajectory.points.append(JointTrajectoryPoint
(positions = pos_obj, velocities = vel, time_from_start
= rospy.Duration(tiempo)))

        # Aumentamos el tiempo de ejecucion
        tiempo = tiempo + duracion/iteraciones

        # Actualizamos la posicion actual
        pos_actual = pos_obj
```

```
    # Mandamos el objetivo y esperamos a que se realice la
   accion
    client.send_goal(g)
    try:
        client.wait_for_result()
    except KeyboardInterrupt:
        client.cancel_goal()
        raise

    return pos_final

# Ejericio 1 define el movimiento de subida y bajada del
   brazo frontal
def ejercicio_1():

    #Definimos las posiciones

    # Eje X
    #pos_1 = [0.03, 0.45, 0.4, 0, 0, math.pi/2]
    #pos_2 = [-0.23, 0.45, 0.4, 0, 0, math.pi/2]

    # Eje Y
    #pos_1 = [-0.11, 0.2, 0.3, 0, 0, math.pi/2]
    #pos_2 = [-0.11, 0.5, 0.3, 0, 0, math.pi/2]

    # Eje Z
    pos_1 = [-0.11, 0.45, 0.4, 0, 0, math.pi/2]
    pos_2 = [-0.11, 0.45, 0.05, 0, 0, math.pi/2]

    # Eje X-Z
    #pos_1 = [0.1, 0.45, 0.4, 0, 0, math.pi/2]
    #pos_2 = [-0.2, 0.45, 0.1, 0, 0, math.pi/2]

    # Eje X-Y
    #pos_1 = [0.2, 0.4, 0.4, 0, 0, math.pi/2]
    #pos_2 = [-0.2, 0.25, 0.4, 0, 0, math.pi/2]

    # Eje Y-Z
    #pos_1 = [-0.11, 0.2, 0.4, 0, 0, math.pi/2]
    #pos_2 = [-0.11, 0.4, 0.2, 0, 0, math.pi/2]
```

```
    # Movemos al comienzo de movimiento
    mover_cartesiano(pos_2[0],pos_2[1],pos_2[2],pos_2[3],
   pos_2[4],pos_2[5])

    # Pedimos el numero de repetaciones y la duracion del
   movimiento
    num_rep = int(input("Numero de repeticiones: "))
    duracion = float(input("Tiempo del recorrido: "))

    # Bucle de ejecucion
    for i in range(0,num_rep):
        movimiento_lineal(pos_2, pos_1, duracion)
        rospy.sleep(0.5)
        movimiento_lineal(pos_1, pos_2, duracion)
        rospy.sleep(0.5)

# Ejercicio 2 define el movimiento de subida y bajada del
   brazo lateral
def ejercicio_2():

    # Tiempo de espera
    rospy.sleep(3)

    # Definimos la posicion inicial
    pos_2 = [-0.11, 0.45, 0.05, 0, 0, math.pi/2]

    # Movemos al comienzo de movimiento
    mover_cartesiano(pos_2[0],pos_2[1],pos_2[2],pos_2[3],
   pos_2[4],pos_2[5])

    # Pedimos el numero de repetaciones y la duracion del
   movimiento
    num_rep = int(input("Numero de repeticiones: "))
    duracion = float(input("Tiempo del recorrido: "))
    radio = float(input("Radio del recorrido: "))
    angulo = float(input("Angulo del recorrido: "))

    # Pasamos el angulo a radianes
    angulo = math.radians(angulo)
```

```
    # Bucle de ejecucion
    for i in range(0,num_rep):
        pos_1 = movimiento_circular(pos_2, radio, angulo,
duracion)
        rospy.sleep(0.5)
        movimiento_circular(pos_1, radio, -angulo, duracion
)
        rospy.sleep(0.5)

# MAIN-----------------------------------------------------------
def main():
    global client
    try:
        # Iniciamos el nodo que manejara la ejecucion
        moveit_commander.roscpp_initialize(sys.argv)
        rospy.init_node('control_ur3', anonymous=True)

        # Asignar el cliente
        #client = actionlib.SimpleActionClient('/
scaled_pos_joint_traj_controller/follow_joint_trajectory
', FollowJointTrajectoryAction) #Real
        client = actionlib.SimpleActionClient('/
arm_controller/follow_joint_trajectory',
FollowJointTrajectoryAction)                    #Simulado
        print "Waiting for server..."
        client.wait_for_server()
        print "Connected to server"

        # Colocamos en la posicion home
        rospy.loginfo('Situando en casa')
        home()

        # Movemos a una posicion de paso
        rospy.loginfo('Posicion paso')
        mover_articular(math.pi/2, -math.pi/2, math.pi/2, -
math.pi, -math.pi/2, 0)

        # Movimiento lineal
        rospy.loginfo('Iniciando ejercicio 1')
        ejercicio_1()
```

```
        # Movimiento circular
        rospy.loginfo('Iniciando ejercicio 2')
        ejercicio_2()

        # Volver a home
        rospy.sleep(3)
        home()

    except KeyboardInterrupt:
        rospy.signal_shutdown("KeyboardInterrupt")
        raise

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

Apéndice 3: Código del control de fuerza

En este apéndice se recoge el algoritmo utilizado en la elaboración del control de fuerza:

```
#!/usr/bin/env python
# importamos las librerias a utilizar
import sys
import rospy
import moveit_commander
import geometry_msgs.msg
import math
import actionlib
import operator
import numpy as np
import scipy.integrate as integrate
import matplotlib.pyplot as plt
from control_msgs.msg import *
from trajectory_msgs.msg import *
from std_msgs.msg import Float64MultiArray
from std_msgs.msg import MultiArrayDimension
from geometry_msgs.msg import WrenchStamped
from controller_manager_msgs.srv import SwitchController
from pykdl_utils.kdl_kinematics import KDLKinematics
from urdf_parser_py.urdf import URDF
from pynput import keyboard as kb

# Cargamos el archivo urdf
file_name = "/home/carlos/catkin_ws/src/
    fmauch_universal_robot/ur_description/urdf/ur3.urdf"

# Nombre de las articulaciones
JOINT_NAMES = ['shoulder_pan_joint', 'shoulder_lift_joint',
     'elbow_joint',
               'wrist_1_joint', 'wrist_2_joint', '
    wrist_3_joint']
client = None

# Obtenemos el grupo del manipulador entero
arm_group = moveit_commander.MoveGroupCommander("
    manipulator")
```

```
# PARADA --------------------------------------------------
def pulsa(tecla):
    print('Se ha pulsado la tecla ' + str(tecla))

def suelta(tecla):

    # Variables globales utilizadas
    global parada
    global switch_controller

    # Si se pulsa la tecla q
    if tecla == kb.KeyCode.from_char('q'):

        # Paramos la publicacion de velocidades
        print ('Parando el programa')
        parada = True

        # Damos tiempo a alejarse del robot
        rospy.sleep(5)

        # Lo devolvemos a la posicion original
        ret = switch_controller(['
   scaled_pos_joint_traj_controller'], ['
   joint_group_vel_controller'], 2, True, 0)
        mover_articular(0, -math.pi/2, 0, -math.pi/2, 0,0)

        # Paramos el programa
        rospy.signal_shutdown('KeyboardInterrupt')
        return False

# FUNCIONES -----------------------------------------------

# Funcion de para obtener los parametros del ur3
def urdf():

    # Abrimos el archivo urdf
    f = file(file_name)

    # Creamos el robot a partir del urdf
    robot = URDF.from_xml_string(f.read())
```

```
    # Guardamos los links de la base y el extremo
    base_link = robot.get_root()
    end_link = robot.link_map.keys()[len(robot.link_map)-1]

    return robot, base_link, end_link

# Funcion para obtener la posicion del robot
def posicion():

    # Definimos la posicion
    pos = arm_group.get_current_pose()

    return pos

# Funcion para obtener el valor de las articulaciones del
   robot
def articulaciones():

    # Definimos la posicion
    art = arm_group.get_current_joint_values()
    return art

# Funcion para mover en el espacio articular
def mover_articular(art1, art2, art3, art4, art5, art6):

    # Creamos la posicion articular
    pos_art = [art1, art2, art3, art4, art5, art6]

    #Duracion
    duracion = 4.0

    # Realizamos la trayectoria con la posicion final
    g = FollowJointTrajectoryGoal()
    g.trajectory = JointTrajectory()
    g.trajectory.joint_names = JOINT_NAMES
    g.trajectory.points = [JointTrajectoryPoint(positions =
    pos_art, velocities = [0]*6, time_from_start = rospy.
   Duration(duracion))]
    client.send_goal(g)
```

```
    try:
        client.wait_for_result()
    except KeyboardInterrupt:
        client.cancel_goal()
        raise

# Funcion que calcula la posicion deseada dependendiendo de
    la velocidad calculada
def calculo_pos_d(velocidades_cart, t_int, pos_inicial):

    # Definimos las posiciones iniciales
    pos_ini_x = (pos_inicial.pose.position.x, 0)
    pos_ini_y = (pos_inicial.pose.position.y, 0)
    pos_ini_z = (pos_inicial.pose.position.z, 0)

    # Calculamos las componentes deseadas x y z
    pos_d_x = tuple(map(operator.add, integrate.quad(lambda
    x:velocidades_cart[0], 0, t_int), pos_ini_x))
    pos_d_y = tuple(map(operator.add, integrate.quad(lambda
    x:velocidades_cart[1], 0, t_int), pos_ini_y))
    pos_d_z = tuple(map(operator.add, integrate.quad(lambda
    x:velocidades_cart[2], 0, t_int), pos_ini_z))

    # Definimos el vector de posicion
    pos_d = []
    pos_d.append(pos_d_x[0])
    pos_d.append(pos_d_y[0])
    pos_d.append(pos_d_z[0])

    return pos_d

# Funcion que realiza el control en velocidad cartesiano
def control_vel_cart(velocidades_cart, t_int, pos_inicial,
   pos_referencia, kdl_kin):

    # Obtenemos la posicion deseada en funcion de la
   velocidad
    pos_deseada = calculo_pos_d(velocidades_cart, t_int,
   pos_referencia)
```

```
# Obtenemos el error entre la posicion deseada y la de
referencia
error_pos =  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
error_pos[0] = pos_deseada[0] - pos_referencia.pose.
position.x
error_pos[1] = pos_deseada[1] - pos_referencia.pose.
position.y
error_pos[2] = pos_deseada[2] - pos_referencia.pose.
position.z

# Definimos la velocidad cartesiana de control
v = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# Obtenemos los valores absolutos de la velocidad
cartesiana
valores = []
for i in range(0,3):
    valores.append(abs(velocidades_cart[i]))

# Obtenemos la media y el valor maximo
media = (valores[0] + valores[1] + valores[2])/3
maximo = np.amax(valores)

# Valoramos la importancia de cada componente
for i in range(0,3):
    if valores[i] <= media/2:
        v[i] = velocidades_cart[i] + 0.1 * error_pos[i]
    elif valores[i] <= media:
        v[i] = velocidades_cart[i] + 0.3 * error_pos[i]
    elif valores[i] <= media + (maximo - media)/2:
        v[i] = velocidades_cart[i] + 0.6 * error_pos[i]
    else:
        v[i] = velocidades_cart[i] + 1 * error_pos[i]

# Devolvemos la posicion
q_ini = articulaciones()

# Jacobiana de la base
J_base = kdl_kin.jacobian(q_ini,None)

# Determinante de la jacobiana
DJ = np.linalg.det(J_base)
```

```
    # Inversa
    J_base_inv = np.linalg.inv(J_base)

    # Calculo velocidad articular
    vel_art = np.matmul(J_base_inv,v)
    vel_art = vel_art.tolist()

    # Pasamos de matriz a vector
    vel_articulaciones = []
    for i in range(0,6):
        vel_articulaciones.append(vel_art[0][i])

    return vel_articulaciones, DJ

# Funcion que calcula las velocidades a partir de la
   fuerzas
def control_admitancia(fuerza_primera, determinante):

    # Definicion de variables
    global fuerza
    velocidades = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    c = 1
    m = 10.9

    # Calculamos la amortiguazion en funcion al
   determinante
    if determinante > 0:

        if (determinante > 0.005):
            c = c + 1 / (determinante)
        else:
            c = c + 10 /determinante

    # Definimos la fuuncion de transferencia
    funcion_trans = (1/c) / (m/c + 1)

    # Calculamos la velocidad en cada eje
    for i in range(0,3):

        # Incremento de la fuerza
        incremento_f = (fuerza[i] - fuerza_primera[i])
```

```
        # Si esta dentro de los limites tendra un valor
        if (incremento_f >= 40.0 and incremento_f <= 200.0)
    :
            velocidades[i] = funcion_trans * incremento_f
        elif (incremento_f >= -200.0 and incremento_f <=
    -40.0):
            velocidades[i] = funcion_trans * incremento_f
        else:
            velocidades[i] = 0.0

    return velocidades

# RESPUESTA AL MENSAJE -----------------------------------
def callback(data):

    # Definimos una varaible global que guarde los
    componentes fuerza
    global fuerza
    fuerza = [0.0, 0.0, 0.0]

    # Guardamos las componentes
    fuerza[0] = data.wrench.force.x
    fuerza[1] = data.wrench.force.z
    fuerza[2] = -data.wrench.force.y


# MAIN ---------------------------------------------------
def main():

    # Esperamos al servicio que para variar entre
    controladores
    rospy.wait_for_service('/controller_manager/
    switch_controller')

    # Varianbles globales
    global client
    global fuerza
    global switch_controller
    global parada
```

```
    # Comenzamos la ejecucion
    try:

        # Variable de parada negativa
        parada = False

        # Iniciamos el variador de controladores
        switch_controller = rospy.ServiceProxy('/
controller_manager/switch_controller', SwitchController)

        # Nos relacionamos como publicadores al nodo de
velocidades
        pub = rospy.Publisher('/joint_group_vel_controller/
command', Float64MultiArray, queue_size=10)

        # Realizamos la subscripcion al nodo de fuerzas del
 extremo del robot
        rospy.Subscriber("/wrench", WrenchStamped, callback
)

        # Lanzamos el nodo controlador
        rospy.init_node('control_impedancia', anonymous=
True)

        # Cliente para el movimiento articular
        client = actionlib.SimpleActionClient('/
scaled_pos_joint_traj_controller/follow_joint_trajectory
', FollowJointTrajectoryAction)
        print "Waiting for server..."
        client.wait_for_server()
        print "Connected to server"

        # Ratio de control de 10hz
        frec = 60.0
        rate = rospy.Rate(frec)

        # Movemos el robot a la posicion inicial
        mover_articular(math.pi/2, -math.pi/2, math.pi/2, -
math.pi, -math.pi/2, 0)

        # Obtenemos los componentes del urdf
        robot, base_link, end_link = urdf()
```

```
        # Creamos un objeto KDL
        kdl_kin = KDLKinematics(robot, base_link, end_link)

        # Cambiamos a un control en velocidad
        ret = switch_controller(['
joint_group_vel_controller'], ['
scaled_pos_joint_traj_controller'], 2, True, 0)

        # Obtenemos la posicion inicial
        pos_ini = posicion()

        # Tiempo del bucle
        t = 1 / frec
        tt = t

        # Definicion velocidades cartesianas
        velocidades_cart = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

        # Definicion del vector de velocidades del robot
        velocidades = Float64MultiArray()
        velocidades.layout.dim.append(MultiArrayDimension()
)   # speed
        velocidades.layout.dim[0].label = "speed"
        velocidades.layout.dim[0].size = 1
        velocidades.layout.data_offset = 1

        print ('Pulsa cualquier tecla para empezar')
        print ('Para parar pulse la tecla q en cualquier
momento')

        # Peticion inicial de paro
        tecla = raw_input()
        if tecla == 'q':
            rospy.signal_shutdown("KeyboardInterrupt")

        # Iniciamos el escuchador del teclado
        escuchador = kb.Listener(pulsa, suelta)
        escuchador.start()

        # Variables iniciales
        fuerza_primera = fuerza
        vel_anterior = velocidades_cart
```

```
      determinante = -1

      # Bucle de control
      while not rospy.is_shutdown() and  escuchador.
is_alive:

          # Velocidades cartesianas del extremo
          velocidades_cart = control_admitancia(
fuerza_primera, determinante)

          # Obtengo la referencia de posicion actual de
referencia
          pos_ref = posicion()

          # Velocidad final que pasarle al robot
          vel_final = []

          # Se da cierta importancia al valor anterior
          for i in range (0,6):
              error_vel = velocidades_cart[i] -
vel_anterior[i]
              vel_final.append(vel_anterior[i] +
error_vel/32.5) # VALOR DIVISION

          # Llamo al control en velocidad
          vel_art, determinante = control_vel_cart(
vel_final, t, pos_ini, pos_ref,kdl_kin)
          vel_anterior = vel_final

          # Cambio velocidades dependiendo del
determinante
          if determinante >= 0.001:
              velocidades.data = vel_art
          else:
              velocidades.data = [0.0, 0.0, 0.0, 0.0,
0.0, 0.0]

          # Si se procede a la parada
          if parada:
              velocidades.data = [0.0, 0.0, 0.0, 0.0,
0.0, 0.0]
```

```
            # Publicar velocidades
            pub.publish(velocidades)

            # Ajusto el intervalo a la frecuencia
            rate.sleep()

    except KeyboardInterrupt:
        rospy.signal_shutdown("KeyboardInterrupt")
        raise

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