UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"IMPLEMENTACIÓN EXOESQUELETO DE MANO EN ROS"

TRABAJO FIN DE GRADO

Diciembre-2020

AUTOR: Pablo Valea Trueba

DIRECTOR/ES: Jorge Antonio Díez Pomares
José María Catalán Orts

RESUMEN

Este proyecto consiste en introducir un exoesqueleto de mano dedicado a fines terapéuticos en ROS (Robotic Operation System) con el objetivo de dotarlo de versatilidad y mejora constante, ya que se trata de un robot que trabaja con personas que necesitan ayuda para poder tener una calidad de vida mejor y no podemos permitirnos perjudicarles bajo ningún concepto.

Ros proporciona múltiples herramientas para pruebas y visualización que nos facilita comprobar cualquier funcionalidad sin poner en riesgo el estado del robot, algo muy importante en el campo de la investigación y especialmente en la creación de la etapa inicial de un robot.

La principal ventaja de esta implementación en Ros son las futuras modificaciones en cuanto a funcionamiento y especialmente en desarrollo de aplicaciones de control del dispositivo.

Palabras clave: Ros, exoesqueleto, mano, nodos, control.

ABSTRACT

This project consists of introducing an exoskeleton hand dedicated to therapeutic purposes in ROS (Robotic Operation System) with the aim of providing it with versatility and constant improvement due that it is a robot that works with people who need help for a better life quality and that won’t harm them by any means.

Ros provides multiple tools for testing and visualization that make it easy for us to check any functionality without putting the state of the robot at risk, a key point in the field of research and particularly in the creation of the robot’s initiation stage.

The main advantage of this implementation in Ros is the future modifications in terms of operation and specially in the development of control applications of the device.

Keywords: Ros, exoskeleton, hand, nodes, control.

ÍNDICE

ÍNDICE DE FIGURAS

1.- INTRODUCCIÓN

En la actualidad nuestra sociedad se ve controlada por el avance de la tecnología, se está alcanzando un punto en el que disponemos de herramientas tecnológicas muy potentes, y es precisamente por esto que debemos ser conscientes y utilizarlas de manera inteligente, ya que, de lo contrario podría ser catastrófico.

Este trabajo consiste en aprovechar los recursos disponibles en el ámbito de la robótica aplicada en la medicina y fisioterapia por medio de exoesqueletos para continuar investigando siempre con el objetivo de mejorar la calidad de vida de personas que realmente lo necesitan.

Para lograr este objetivo, el trabajo se centra en el exoesqueleto de mano desarrollado por el laboratorio de Neuroingeniería Biomédica (NBIO) de la Universidad Miguel Hernández de Elche, robot creado desde cero para mejorar la calidad de vida de personas que han perdido movilidad parcial o incluso total en las manos debido principalmente a enfermedades cerebro-vasculares.

Existen infinitas formas de controlar un dispositivo robótico hoy en día, pero tras una profunda investigación, se ha llegado a la conclusión de la importancia de tener un software estandarizado de código abierto con el apoyo de una gran comunidad de usuarios del ámbito de la robótica y tan enraizado en la investigación como es el caso de ROS (Robotic Operation System).

En el transcurso del presente trabajo se pretende implementar el exoesqueleto de mano en ROS, creando su propia programación, estructura y cinemática con el fin de controlar el dispositivo para poder contar con prestaciones como:

- Posibilidad de usar prácticamente cualquier lenguaje de programación (Python, C++, Java…).
- Facilidad para cualquier modificación.
- Simulaciones sin necesidad de tener el dispositivo físicamente.
- Comunicar el robot virtual con el real.
- Visualización en tiempo real.
- Controlar desde el ordenador el dispositivo.

La gran ventaja de lograr tener implementado el robot en ROS es el margen de mejora que proporciona de cara al futuro, ya que se puede controlar el robot desde cualquier parte con un ordenador en configuración de control remoto e incluso lograr una virtualización completa para poder trabajar sin necesidad del dispositivo real.

2.- MOTIVACIÓN Y OBJETIVOS

Hay varios motivos que me hicieron decantarme por este tema, el principal es ayudar a mejorar la calidad de vida de personas que por desgracia sufren pérdida de movilidad. Soy muy consciente de ello porque mi abuela sufrió un ictus cerebral que le provocó parálisis en el lado izquierdo de su cuerpo, convirtiéndola de un día para otro en una persona dependiente. A raíz de este suceso que es más habitual de lo que mucha gente piensa, nos informamos de la importancia de la rehabilitación sobre todo en el menor corto plazo posible para poder conseguir recuperar parte de movilidad o incluso, en algunos casos, su totalidad.

Cabe destacar la situación actual que estamos viviendo con el mundialmente conocido “Coronavirus” o “COVID-19”, la duración del estado de alarma ha marcado un antes y un después, consiguiendo paralizar casi todo por completo. Precisamente una de las vías de escape más importante ha sido en el ámbito de la informática y por eso me parece fundamental hoy en día tener la posibilidad de trabajar por ejemplo con un robot en estado de control remoto, algo que se puede lograr con este trabajo y mejorar con futuras investigaciones.

Se encuentran a continuación los objetivos principales:

- Comprender y profundizar en el ámbito de la robótica y en especial en ROS (Robotic Operation System).
- Lograr implementar el exoesqueleto de mano (Helium) desarrollado por el grupo de Neuroingeniería Biomédica (NBIO) de la universidad UMH dentro del sistema ROS:

- ✓ Crear el paquete principal de ROS personalizado para el robot Helium.
- ✓ Visualizar los eslabones principales con ayuda del simulador RVIZ
- ✓ Desarrollar su cinemática directa empleando la programación URDF.
- ✓ Programar los *nodes*, *topics* y *messages* necesarios para su funcionamiento.

- Comunicar en tiempo real el dispositivo (exoesqueleto de mano) con la simulación virtual en ROS.
- Controlar el dispositivo.

3.- ESTADO DEL ARTE

Existen numerosos dispositivos robóticos hoy en día dedicados a la rehabilitación del miembro superior, en especial a recuperar la movilidad de miembros como son el codo y el hombro. El principal objetivo de estos dispositivos es lograr una terapia de rehabilitación integral, es decir, contar con un dispositivo robótico especializado para cada parte del cuerpo humano. El presente trabajo se centra en un exoesqueleto de mano, ya que es imprescindible para los miembros superiores incluir la función motora de la mano.

Se han desarrollado algunos exoesqueletos enfocados a la mano, como por ejemplo:

- **Hexxor**, creado por la Universidad Católica de América [1].

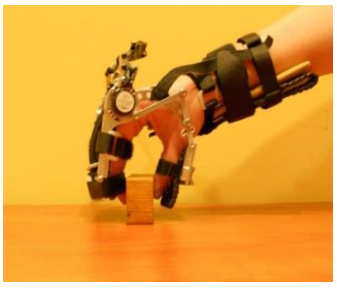

Figura 1: Hexxor

- **Amadeo**, creado por la compañía de ingeniería médica australiana *Tyromotion* [2].

Figura 2: Amadeo

- **Gloreha**, realizado por la empresa italiana *Idrogenet* en colaboración con la Universidad de Brescia, Italia [3].

Figura 3: Gloreha

Merece la pena destacar que tras una profunda búsqueda, los dispositivos anteriormente citados junto con algunos pocos más [4] [5] son una minoría considerable frente a otros dispositivos encargados de rehabilitar diferentes miembros del cuerpo humano.

Es muy importante definir bien la aplicación del exoesqueleto manual, ya que puede presentar distintas funciones. Es decir, un exoesqueleto de mano utilizado para rehabilitación necesita ser manejable y a su vez permitir un amplio rango de movimientos. No obstante, también debe ser lo suficientemente rígido para asegurar un agarre firme de los objetos presentes durante las actividades cotidianas. Para satisfacer estas diferentes necesidades existen diversas arquitecturas de transmisión de fuerza:

Algunos dispositivos utilizan enlaces para transmitir la fuerza del actuador a las articulaciones humanas. Esta arquitectura requiere una gran alineación entre la cinemática del robot y las articulaciones, pero permite un buen control e incluso alcanzar movimientos complejos debido a la flexibilidad del diseño.

Otra arquitectura muy utilizada es el guante de cable, considerada más flexible y simple. Esta alternativa depende de las propias articulaciones humanas, evitando así posturas incómodas. Por contra necesita ayuda de poleas para lograr fuerzas elevadas y es más difícil de controlar en posiciones intermedias. Además, son necesarios un par de cables para la extensión y flexión.

Por último, algunos dispositivos utilizan actuadores deformables, como músculos neumáticos que se unen directamente a la mano por medio de un guante. El problema de esta arquitectura es que la colocación de los actuadores no es en el lugar más favorable para lograr una fuerza elevada.

4.- DISEÑO

4.1.- DISPOSITIVOS

En la figura 4, se muestra el diagrama de bloques de los dispositivos utilizados en el presente trabajo:

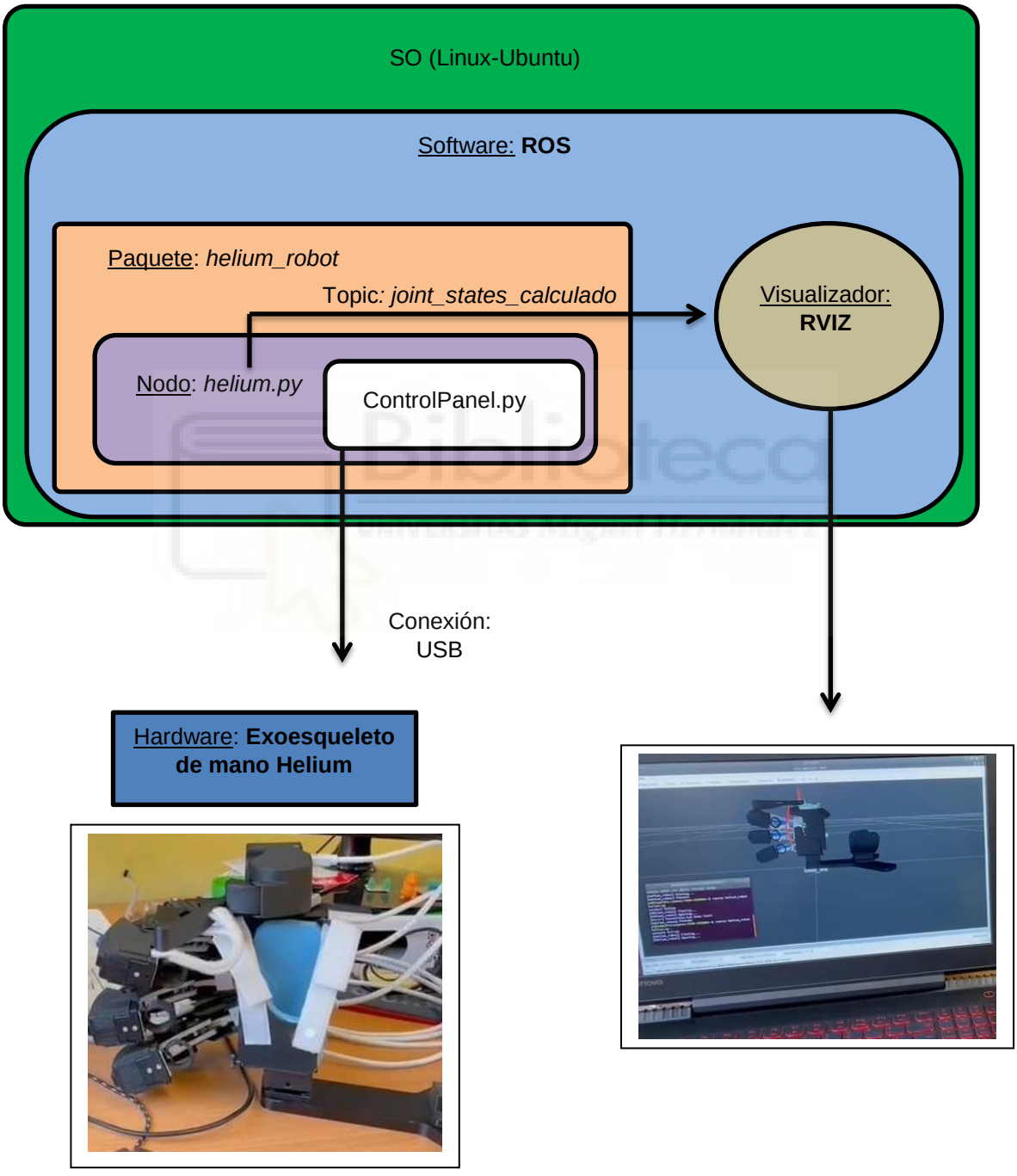

Figura 4: Diagrama de bloques de los dispositivos del trabajo

Se puede observar una estructura claramente diferenciada, se necesita por un lado la arquitectura hardware formada en este caso por el exoesqueleto de mano Helium, arquitectura software basada en ROS y para finalizar el visualizador RVIZ, encargado de la simulación del dispositivo virtual.

4.1.1.- EXOESQUELETO DE MANO HELIUM

El presente exoesqueleto en el que se basa el trabajo fue desarrollado por el laboratorio de Neuroingeniería Biomédica (NBIO) de la Universidad Miguel Hernández de Elche [6]. Está basado en la aproximación de vinculación para lograr una buena aplicación en el ámbito de la rehabilitación, para el que está pensado y diseñado. La transmisión del movimiento consiste en un mecanismo de barra que facilita el acoplamiento de las falanges, consiguiendo de esta forma un movimiento natural de la mano, y lo más importante, empleando únicamente un grado de libertad activo por dedo. Asimismo, gracias al diseño que presenta el exoesqueleto, es capaz de transmitir cargas de tracción, así como de compresión, permitiendo el movimiento de extensión y flexión.

El exoesqueleto está formado por tres módulos de dedos idénticos encargados de impulsar el índice, el dedo medio o corazón y el par formado por los dedos anular y meñique. Otro módulo dedicado al pulgar, este módulo sigue en fase de desarrollo e investigación para mejorar el movimiento ya que, al presentar un grado de libertad de oposición, su movimiento no se puede aproximar bien mediante un mecanismo plano. Para la mano se ha creado una órtesis comercial semirrígida y una base para todo el exoesqueleto dónde se apoya el brazo del usuario.

Cabe destacar el diseño del sistema de fijación entre el exoesqueleto y el usuario, formado por unas piezas en forma de anillo que se insertan en las falanges proximales y mediales a presión, permitiendo así un montaje rápido y sencillo, pero sobretodo y más importante, una seguridad extra para el usuario ante cualquier complicación ya que se puede desacoplar de manera inmediata. De una manera similar, todos los módulos de dedo se fijan a la órtesis de la mano mediante una ranura en forma de guía que permite ajustar la posición del módulo a lo largo de la dirección longitudinal del dedo.

Todas las piezas que forman el exoesqueleto han sido impresas mediante impresora 3D o mecanizadas.

Figura 5: Dispositivo exoesqueleto de mano Helium

4.1.2.- ROS (ROBOTIC OPERATION SYSTEM)

El origen de ROS se remonta a principios del **año 2007**, se creó desde cero en el **Laboratorio de Inteligencia Artificial de Stanford** bajo el nombre de ***switchyard***. Dos estudiantes de doctorado, Kenneth Sailsbury y Keenan Wyrobek mientras trabajaban en robots aplicados a entornos humanos se dieron cuenta que muchos compañeros presentaban numerosas dificultades en el ámbito de la robótica. Fue entonces cuando surgió la idea de crear un sistema base que proporcionará un punto de partida y estructuras básicas para ayudar y facilitar el trabajo a todos.

ROS es un ***framework***, para lograr entender bien este concepto hay que tener claro que es un conjunto de herramientas y módulos que se pueden utilizar para varios proyectos. En el caso particular de ROS, se trata de un sistema de referencia flexible para escribir software de robot, formado por un conjunto de herramientas, bibliotecas y convenciones cuyo objetivo es simplificar la tarea de crear un comportamiento robótico complejo y robusto en una amplia variedad de plataformas robóticas.

ROS a pesar de no ser un sistema operativo con todas sus propiedades, provee servicios estándar como la abstracción del hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el mantenimiento de paquetes. Se basa en una arquitectura de grafos en la que el procesamiento se produce en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. Todo este proceso es complejo y está compuesto por numerosos factores determinantes como por ejemplo: Ros *master*, *topics*, *messages*…

Se puede diferenciar dos partes básicas: La parte de **sistema operativo** (**ros**) y la parte que consiste en un **conjunto de paquetes** aportados por la contribución de usuarios (**ros-pkg**).

ROS es un software libre bajo términos de **licencia BSD** (Berkeley Software Distribution), este tipo de licencia permite ver y modificar el código e incluso cerrar el sistema o aplicación. Gracias a la misma hay libertad para uso comercial e investigador.

Su instalación se puede consultar en el **Anexo 2: Instalación ROS**

ROS se puede estructurar en tres niveles: sistemas de archivos, gráfico de cálculo y comunidad. A continuación se explica cada uno de ellos con el máximo detalle posible.

1. Nivel de sistema de archivos ROS
    - **Paquetes:** Unidad principal de organización del software en ROS. Cada paquete puede contener procesos en tiempo de ejecución de ROS (nodos), una biblioteca dependiente de ROS, conjuntos de datos, archivos de configuración…. Los paquetes son el elemento de construcción y lanzamiento.
    - **Metapaquetes:** Los metapaquetes son paquetes especializados que únicamente sirven para representar un grupo de paquetes relacionados entre sí. Normalmente

solo se utilizan como un marcador de posición compatible con versiones anteriores.

- **Manifiestos de paquetes:** Los manifiestos (package.xml) proporcionan metadatos sobre un paquete, incluyendo su nombre, versión, descripción, información de licencia, dependencias y otra meta información como por ejemplo, paquetes exportados.
- **Repositorios/Pilas:** Los repositorios o pilas son una colección de paquetes que comparten un sistema VCS común, es decir, comparten la misma versión y por tanto se pueden lanzar juntos usando la herramienta de automatización de liberación de catkin (bloom). Los repositorios pueden contener un único paquete.
- **Tipos de mensajes (msg):** Las descripciones de los mensajes, almacenadas en *my_package / msg / MyMessageType.msg*, definen las estructuras de datos para los mensajes enviados en ROS.
- **Tipos de servicios (srv):** Las descripciones de los mensajes, almacenadas en *my_package / srv / MyServiceType.srv*, definen las estructuras de datos de solicitud para los servicios en ROS

2. Nivel de gráfico de cálculo ROS

El gráfico de cálculo es la **red peer-to-peer** de procesos ROS. El concepto de red peer-to-peer simplemente es una red de ordenadores en la que todos o algunos aspectos funcionan sin clientes ni servidores fijos, sino una serie de nodos que se comportan como iguales entre sí. Estas redes permiten el intercambio directo de información, independientemente del formato, entre todos los ordenadores interconectados.

Figura 6: Red peer-to-peer

Los conceptos básicos de gráficos de cálculos se implementan en el repositorio *ros_comm* y son:

- **Nodos:** Los nodos son procesos encargados de realizar cálculos. Un sistema de control de robot normalmente está formado por muchos nodos, ya que ROS está diseñado para ser modular. Por ejemplo, un nodo controla los motores de las ruedas, un nodo realiza la planificación de la ruta, y así sucesivamente. Los nodos se escriben con ayuda de una biblioteca cliente ROS, como roscpp (implementación C++) o rospy (implementación Python).
- **Master:** El ROS Master proporciona registro de nombres y búsqueda del resto del gráfico de cálculo. Sin este nodo maestro, los nodos no podrían encontrarse, intercambiar mensajes o invocar servicios.
- **Servidor de parámetros:** El servidor de parámetros forma parte del máster y su función es permitir que se almacenen los datos en una ubicación central.
- **Mensajes:** Los nodos se comunican entre sí por medio de mensajes. Un mensaje no es nada más que una estructura de datos. Los mensajes admiten los siguientes tipos de datos: primitivos estándar (entero, punto flotante, booleano…) y matrices primitivas. También pueden presentar estructuras y matrices anidadas.
- **Temas (*topics*):** La manera en la que un nodo envía un mensaje es publicándolo en un tema determinado. El tema es un nombre que se utiliza para identificar el contenido del mensaje. Los nodos deben suscribirse al tema apropiado para el tipo de dato que desean. Cada tema puede tener varios editores y suscriptores a la vez. Además un nodo también puede publicar o suscribirse a múltiples temas.
- **Servicios:** Son similares a los temas pero en lugar de usar la comunicación publicación/suscripción, utilizan la **solicitud/respuesta**. Los servicios se definen mediante un

par de estructuras de mensajes: una para la solicitud y otra para la respuesta.

Un nodo proveedor ofrece un servicio con un nombre y un cliente utiliza el servicio enviando el mensaje de solicitud y esperando la respuesta.

- **Bolsas:** Las bolsas son formatos para guardar y reproducir datos de mensajes ROS. Son muy útiles para almacenar datos, como por ejemplo los datos de los sensores, que pueden ser difíciles de recopilar, pero son necesarios.

Resumen básico y simplificado para comprender el funcionamiento de ROS:

Los **procesos** independientes son lanzados como **nodos**. Los nodos son programas ejecutables gestionados por un nodo principal, denominado **ROS Master** (línea de comandos: roscore).

Estos nodos se pueden comunicar por medio de: **Temas (*topics*)**, uno o varios nodos publican un mensaje en un determinado tema y otro/s nodos se suscriben a ese mismo tema. **Servicios** (solicitud/respuesta) o a través del **servidor de parámetros** (se almacenan y obtienen parámetros en tiempo de ejecución).

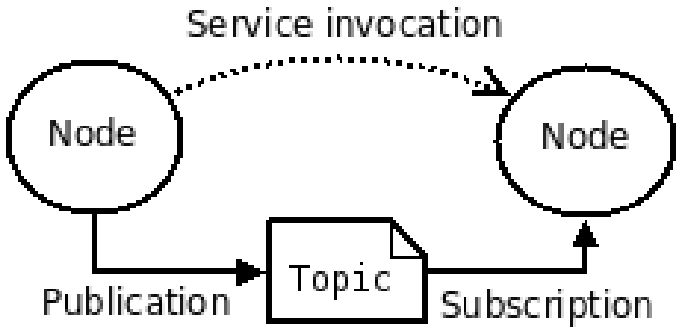

Figura 7: Funcionamiento gráfico de ROS

3. Nivel de comunidad ROS

ROS permite y facilita a comunidades separadas intercambiar software y conocimiento.

- **Distribuciones:** Las distribuciones ROS son colecciones de pilas o versiones que se pueden instalar. Las distribuciones juegan un papel similar a las de Linux, es decir, facilitan la instalación y mantienen versiones consistentes de software.
- **Repositorios:** Ros se basa en una red federada de repositorios de código, en la cual, diferentes instituciones pueden desarrollar y lanzar sus propios softwares robóticos.
- **Wiki de ROS:** Es el foro principal para documentarse acerca de ROS [8].
- **Listas de correos:** Es el canal de comunicación principal para conocer las actualizaciones de ROS, así como un foro para hacer preguntas sobre el software de ROS.
- **Blog:** blog ros.org

Además antes de comenzar es necesario explicar brevemente las herramientas que se utilizarán posteriormente a lo largo del trabajo por línea de comandos.

Resumen herramientas importantes de línea de comandos ROS:

- ls: permite visualizar el contenido de un directorio o carpeta, es necesario estar situado en la misma.
- rosls: con esta orden podemos visualizar el contenido de la carpeta ROS que queramos desde cualquier directorio.
- cd: cambia el directorio actual a la ruta que se le indique, es necesario estar situado en el directorio anterior.
- roscd: permite cambiar de directorio desde cualquier posición.
- cp: copiar archivos de una ruta a otra de forma sencilla.
- mkdir: para crear una carpeta indicando el nombre a continuación.

- catkin_make: compila los paquetes que estén en el espacio de trabajo, es importantísimo ejecutarlo con cualquier modificación.
- catkin_create_pkg: sirve para crear un paquete desde cero.
- roscore: ejecuta el nodo maestro, que sirve para conectar el resto de nodos.
- rosrun: sirve para ejecutar un nodo. rosrun [paquete] [nombre nodo].
- rosnode, rostopic, rosmsg y rossrv muestran información sobre los nodos, temas, mensajes y servicios. Además se les puede añadir para elegir la información que se quiere obtener de ellos los siguientes comandos:
  -info: muestra la información de un nodo, tema o servicio.
  -list: lista todos los nodos, temas o servicios que están activos.
  -echo: muestra por pantalla los mensajes que se están publicando.
  -pub: publica datos en un tema.
  -type: muestra el tipo de tema o servicio especificado.
  -kill: elimina un nodo en funcionamiento.
- roslaunch: sirve para ejecutar un lanzador.

Si se necesita obtener más información sobre los comandos con los que cuenta cada herramienta y su método de empleo, se puede escribir desde el terminal:

[Herramienta] –h

Para concluir, simplemente decir que estas no son todas las herramientas, pero sí las más utilizadas y las que se emplearán a lo largo del trabajo.

4.1.3.- RVIZ

Rviz es una herramienta de visualización de datos en 3D para aplicaciones de ROS muy completa y útil. Proporciona vista del modelo de robot, captura la información de los sensores del robot y reproduce los datos capturados.

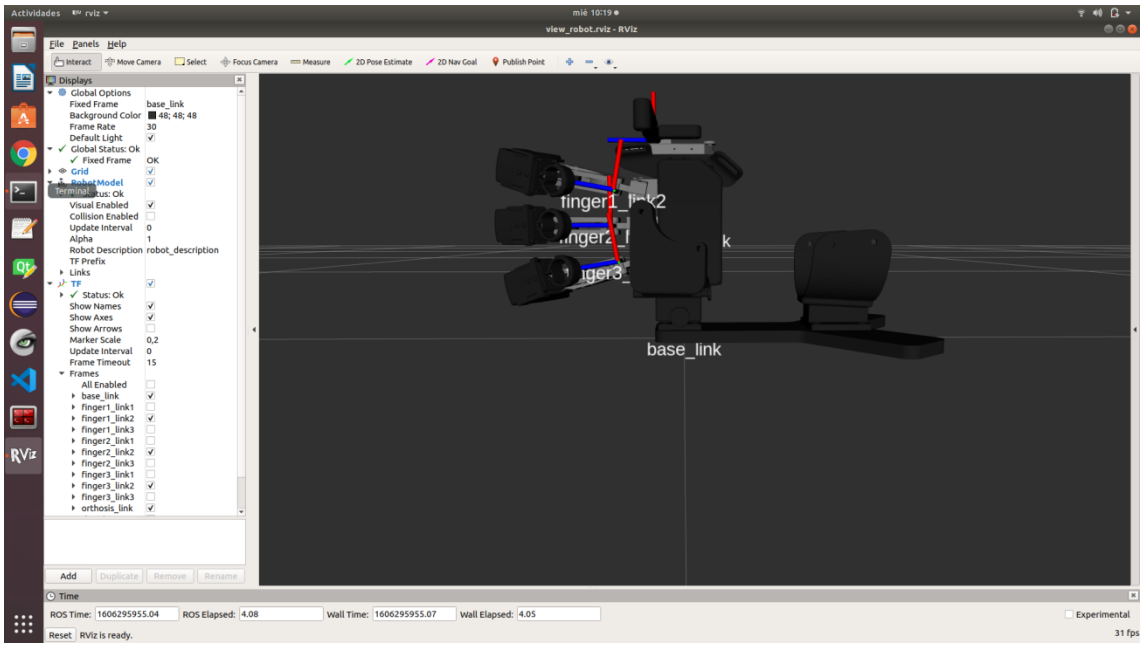

Figura 8: RVIZ exoesqueleto completo

Rviz permite cambiar la configuración prácticamente al gusto del usuario y el mejor método para aprender su funcionamiento es utilizándolo. En la figura 8 se puede observar en el margen izquierdo una serie de opciones globales y complementos que son necesarios explicar antes de comenzar a usarlo.

Hay dos sistemas de referencias de coordenadas importantes:

- **El sistema de referencia Fijo (*Fixed frame*)**: Es el sistema de referencia que se toma como referencia para mostrar todos los datos. Aparece en la parte izquierda en *Global Options*. Se debe asignar a un punto estático como por ejemplo el mundo o el mapa. En caso de no tenerlo, se podría asignar a la odometría del robot. No se debe asignar nunca a un sistema de referencia en movimiento, como sería por ejemplo, la base de un robot móvil.
- **El sistema de referencia objetivo (*Target frame*)**: Es el sistema de coordenadas que se asigna a la vista. Este sí que se podría asignar a la base del robot, por ejemplo. Se pueden crear diferentes vistas para ir cambiando de una a otra en función de lo que se desee ver.

Además, RVIZ también cuenta con diferentes modos de operación que se pueden seleccionar en la parte superior del menú interacción. Por defecto siempre está en interactuar y se puede mover la cámara, seleccionar elementos (y ver sus valores, como, por ejemplo, coordenadas de puntos…)

Para finalizar simplemente comentar que en la parte inferior izquierda del menú se encuentra la opción de añadir complementos en Rviz utilizando el botón (**Add**). Las opciones que dispone son: Axes, Effort, Camera, Grid, Grid Cells, Image, InteractiveMarker, Laser Scan, Map, Markers, Path, Point, Pose, Pose Array, Point Cloud, Polygon, Odometry, Range, RobotModel, TF, Wrench y Oculus.

En el caso del presente robot ha sido necesario activar:

Grid: Es el encargado de mostrar por pantalla una cuadrícula completamente configurable encargada de aportar un suelo al mundo virtual.

PROPIEDADES			
NOMBRE	**DESCRIPCIÓN**	**VALORES VÁLIDOS**	**DEFAULT**
Plane Cell Count	Numero de celdas a dibujar en el plano de la cuadrícula.	>1	10
Normal Cell Count	Numero de celdas a dibujar a lo largo de la cuadrícula. El valor 0 es para 3D.	>0	0
Cell Size	Longitud en metros del lado de cada celda.	>0.0001	1
Line Style	Operación de renderizado utilizada en función de la distancia de la cámara.	Líneas, Vallas publicitarias	Líneas
Line Width	Ancho de las líneas en metros únicamente para vallas publicitarias.	>0.0001	0.03
Color	El color de las líneas.	([0-255], [0-255], [0-255])	(127, 127, 127)
Alpha	Transparencia aplicada a las líneas.	[0-1]	0.5
Plane	El plano en el que se dibuja la cuadrícula.	XY, XZ, YZ	XY

RobotModel: Muestra por pantalla los enlaces de un robot (previamente definidos en un archivo de programación **URDF [Capítulo 5.2.1]**), en sus correctas posiciones de acuerdo con el árbol de transformación **TF**.

PROPIEDADES			
NOMBRE	**DESCRIPCIÓN**	**VALORES VÁLIDOS**	**DEFAULT**
Visual Enabled	Permite dibujar el robot.	TRUE OR FALSE	TRUE
Collision Enabled	Permite representar la colisión del robot.	TRUE OR FALSE	FALSE
Update Rate	Velocidad de actualización de la posición de cada enlace en segundos.	>0.01	0.1
Alpha	Transparencia aplicada a los enlaces o eslabones.	[0-1]	1
Robot Description	Parámetro del que recuperar el archivo URDF. Se usa searchParam().	Cualquier recurso gráfico	Robot_description
TF Prefix	Sirve para preparar los nombres de los enlaces cargados desde el URDF.	Cualquier prefijo TF	vacío
Show Trail	Dibuja un rastro detrás del enlace.	TRUE OR FALSE	FALSE
Show Axes	Dibuja ejes en el origen del vínculo.	TRUE OR FALSE	FALSE

TF: Es un paquete que permite al usuario realizar un seguimiento de numerosos sistemas de referencia durante un largo periodo de tiempo. Permite la transformación de puntos, vectores… entre dos sistemas de referencia en cualquier momento deseado.

El proceso para visualizar el exoesqueleto de mano Helium en RVIZ (figura 8) se explicará en el transcurso del trabajo.

5.- IMPLEMENTACIÓN

En este apartado se va a explicar la implementación de cada uno de los elementos que componen el sistema.

5.1.- IMPLEMENTACIÓN EXOESQUELETO DE MANO HELIUM

Para realizar la implementación del robot se debe programar una librería de comunicación serie formada por diferentes scripts, estos scripts deben incorporarse dentro del paquete creado llamado en este caso (**helium_robot**), más concretamente en la subcarpeta source (**src**) y a continuación en la subcarpeta creada con el nombre **scripts** para cumplir con los estándares de ROS.

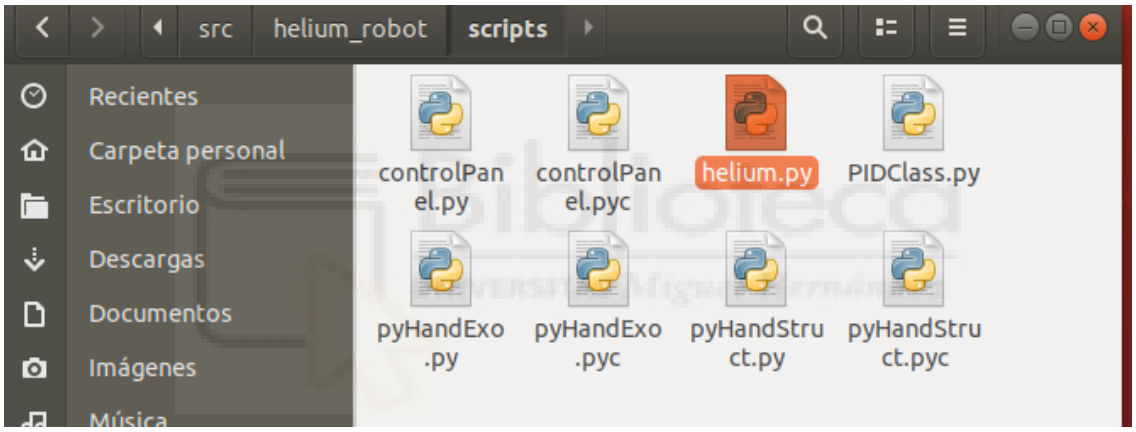

Figura 9: Scripts encargados de la implementación del robot

La gran ventaja añadida de trabajar en ROS es la aceptación de diferentes lenguajes de programación, gracias a esto se han podido reutilizar gran cantidad de códigos ya creados con anterioridad para el control del dispositivo.

El control del dispositivo se realizaba con esta misma librería de comunicación serie pero creada para un sistema operativo diferente (Windows) y controlada a bajo nivel por el programa Matlab. Se ha tenido que adaptar modificando ciertos parámetros para poder utilizarla en Linux-Ubuntu.

El lenguaje de programación utilizado es Python principalmente por ser un lenguaje de alto nivel con una sintaxis muy sencilla de comprender y sobre todo por tener una ejecución rápida sin necesidad de ser compilado (gran ventaja frente a por ejemplo C o C++).

El funcionamiento de esta implementación es un poco complejo de explicar sin entrar en profundidad en códigos de programación, precisamente por eso a continuación en la Figura 10 se ha creado un esquema para poder comprenderlo de manera visual. El resumen simplificado es, una estructura de scripts encargados de cerciorarse de la conexión del robot al ordenador vía USB gracias al envío en tiempo real de unos mensajes llamados **mensajes de keep alive.** Cuando estos mensajes se reciben correctamente a una frecuencia igual o superior a unos 0.5 Hz aproximadamente se indica que la conexión es correcta, de lo contrario se muestra por pantalla un mensaje de error. Una vez establecida la conexión, el sistema se encarga de convertir los valores introducidos por el usuario por teclado (siempre y cuando sean correctos, es decir, dentro del rango indicado) en las órdenes de ejecución para accionar los respectivos motores del exoesqueleto encargados del movimiento.

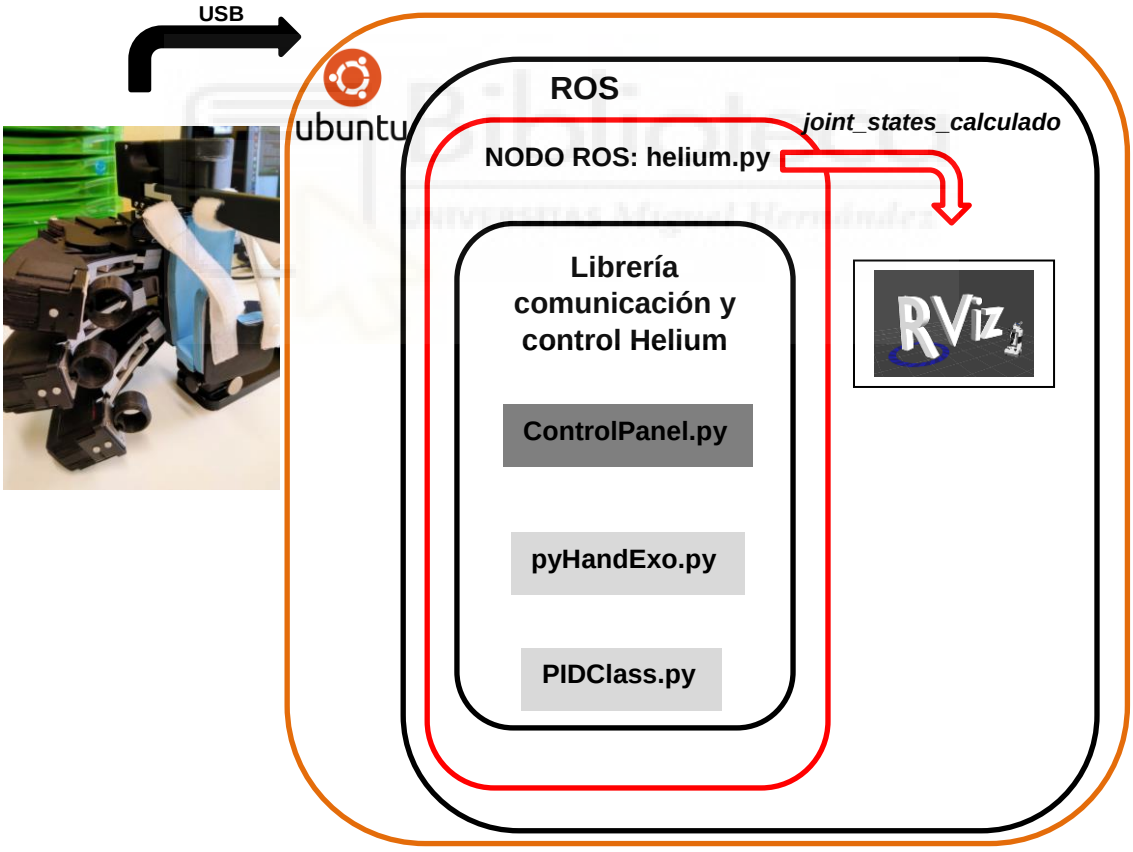

Figura 10: Esquema Implementación del exoesqueleto de mano Helium

Para comprender mejor este apartado, es importante entender que la implementación del robot (exoesqueleto de mano Helium), es independiente a ROS, es decir, se podría controlar gracias a esta librería de comunicación serie el robot con cualquier programa desarrollado en Python. Sin embargo, el objetivo del sistema creado en el presente trabajo es controlarlo desde ROS al mismo tiempo que se realiza la simulación en la herramienta de ROS (RVIZ).

Por tanto, el script **helium.py** es el encargado de controlar el exoesqueleto de mano Helium desde ROS. La manera de lograr esto es convirtiendo dicho script en un **nodo de ROS**.

En el siguiente apartado se explica el proceso de implementación del robot en simulación.

5.2.- IMPLEMENTACIÓN DEL ROBOT EN SIMULACIÓN

Para lograr introducir el robot en simulación se debe partir del modelo desarrollado en formato CAD, concretamente desarrollado en Autodesk Inventor. El procedimiento a seguir es aislar cada eslabón para exportarlo en formato ***stl***, prestando especial atención al sistema de coordenadas, ya que los archivos exportados mantienen su posición y posteriormente es necesario conocerla para introducirlos en ROS.

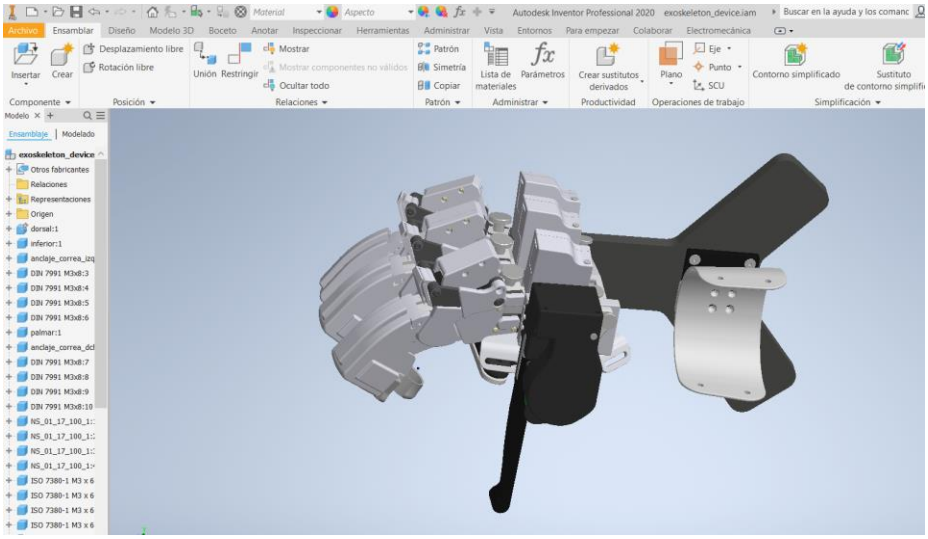

Figura 11: Captura del diseño del exoesqueleto de mano Helium en Autodesk Inventor

Una vez exportados todos los eslabones que forman el exoesqueleto de mano completo con la extensión (.stl) se deben introducir en ROS. La forma de hacerlo es utilizando un archivo **URDF** y creando dentro del paquete una carpeta o directorio denominado ***meshes*** para cumplir con los estándares de ROS.

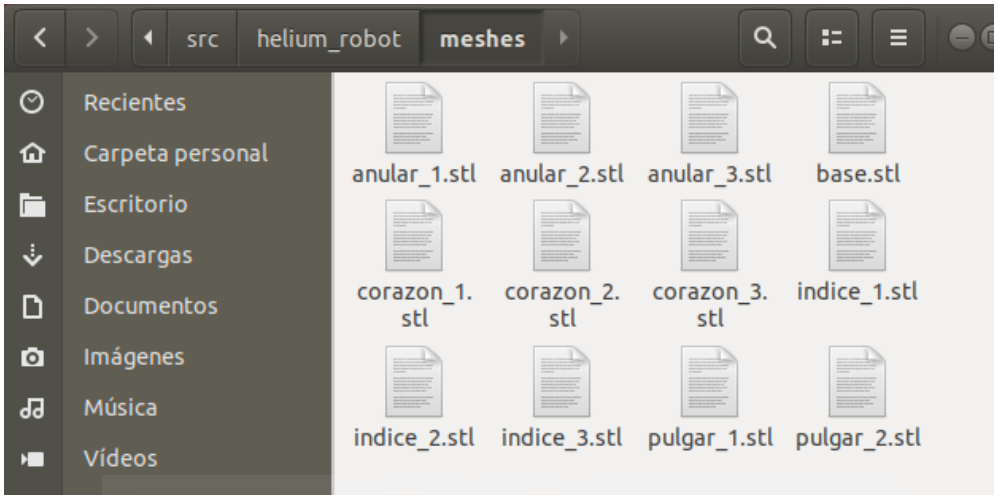

Figura 12: Carpeta meshes compuesta por los archivos STL del exoesqueleto para su simulación en 3D

5.2.1.- ARCHIVO URDF

Un archivo URDF (Unified Robot Description Format) es un sistema para la escritura de software robótico de desarrollo libre. Por lo tanto, el usuario puede personalizar los parámetros y características del robot a su gusto. Una vez definido el software, se puede comprobar y realizar simulaciones en el entorno deseado, con el objetivo de posteriormente ejecutarlo en tiempo real.

La estructura de los archivos URDF siempre es igual, se basa en una estructura en forma de árbol entre eslabones (***links***) y articulaciones (***joints***). Para comprender mejor este tipo de estructura se puede apreciar de manera visual un ejemplo en la figura 13. Se puede observar como la estructura parte de un primer eslabón denominado link 1, cuya función es ejercer de base para todo el sistema.

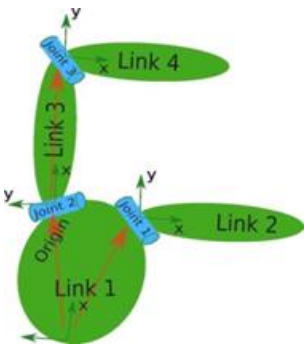

Figura 13: Ejemplo de estructura URDF

En la cabecera de este documento siempre se debe poner la versión y el elemento raíz del archivo de descripción del robot, es decir, un robot. A modo de ejemplos se utilizan pequeños fragmentos del propio código URDF creado para la implementación del robot (exoesqueleto de mano Helium) que se encuentra completo en el **Anexo 3** del presente trabajo:

<?xml version=”1.0**”?>**

<robot name= “exoesqueleto_Helim **”>**

A continuación, es importante comenzar explicando todos los elementos que se encuentran encapsulados dentro del elemento raíz (robot). Estos elementos son: **link, joint, transmission y gazebo.**

- **Link:** El elemento de enlace o eslabón describe un cuerpo rígido con inercia, características visuales y propiedades de colisión como se observa en la figura 14. A continuación se explica con detalle y empleando ejemplos cada una de las características que forman este elemento:

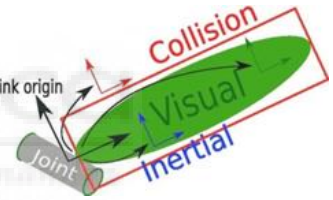

Figura 14: Elemento link

<link name= “ ”> (Obligatorio)
El atributo principal del enlace es su nombre.

<inertial> (Opcional: Por defecto tienen valores nulos).
Propiedades inerciales del enlace.

<origin> (Opcional: El valor predeterminado es la identidad,1).
Posición del sistema de referencia inercial en relación con el sistema de referencia del enlace. El origen del sistema de referencia inercial debe estar en el centro de gravedad.

xyz (Opcional: Por defecto es un vector nulo,[0,0,0]).
Representa la posición en el espacio tridimensional.

rpy (Opcional: Por defecto es el vector identidad,[1,1,1]).

Representa los ángulos en radianes de rotación (*Roll*, *Pitch* y *Yaw*).

<mass>

La masa del enlace está representada por el atributo de valor de este elemento.

<inertia>

La matriz de inercia rotacional 3x3, debido a que es simétrica es suficiente con indicar 6 elementos mediante el uso de los atributos: ixx,ixy,ixz,iyy,iyz,izz.

</inertial>

<visual> (Opcional)

Propiedades visuales del enlace. Este elemento especifica la forma del objeto (caja, cilindro, esfera…). Pueden existir varias etiquetas <visual> para un mismo enlace.

name (Opcional)

Se puede especificar un nombre para una parte de la geometría de un vínculo.

<origin> (Opcional: El valor predeterminado es la identidad,1).

El sistema de referencia del elemento visual respecto al sistema de referencia del enlace.

xyz (Opcional: Por defecto es un vector nulo,[0,0,0]).

Representa la posición en el espacio tridimensional.

rpy (Opcional: Por defecto es el vector identidad,[1,1,1]).

Representa los ángulos en radianes de rotación (*Roll*, *Pitch* y *Yaw*).

<geometry> (Obligatorio)

Define la forma del objeto visual. Existe una amplia variedad, pero merece la pena destacar las más utilizadas:

<box> Ej: `<box size = "1 1 1" />`

El atributo *size*, contiene las tres longitudes de los lados de la caja, teniendo como origen su centro.

<cylinder> Ej: `<cylinder radius="1" length="0.5"/>`

Es necesario definir su radio y longitud, teniendo el origen en su centro.

<sphere> Ej: `<sphere radius="0.002"/>`

Solo hace falta el radio y su origen está en el centro.

<mesh>

Un elemento trimesh especificado por un nombre y una escala opcional. Cualquier formato de geometría es aceptable, pero el formato recomendado es Collada con extensión (.dae).

<material> (Opcional)

El material del elemento visual. Se permite definir un material fuera del propio objeto enlace, es decir, se puede hacer referencia al material por su nombre.

name Nombre del material

<color> (Opcional)

rgba El color se crea en función de 4 dígitos que representan la mezcla entre los colores (rojo, verde, azul y alfa), cada uno en el rango [0-1].

<texture> (Opcional)

La textura se especifica mediante un archivo.

</visual>

<collision> (Opcional)

Propiedades de colisión de un enlace. Este elemento se suele utilizar para reducir el tiempo de cálculo en modelos muy complejos. Pueden existir dentro del mismo enlace varias etiquetas <collision>. La unión de la geometría que definen forma la representación de colisión del enlace.

name

<origin>

xyz

rpy

<geometry>

</collision>

Proceso similar al explicado en el elemento anterior (visual).

- **Joint:** Es el elemento de unión que describe la cinemática y la dinámica de la junta. Además, es el encargado de especificar los límites de seguridad de dicha junta. En la figura 15 se puede observar un ejemplo genérico de su correcta estructura.

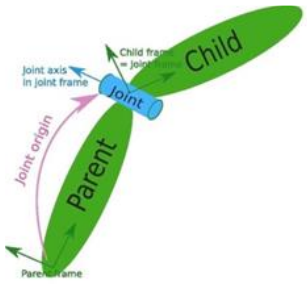

Figura 15: Elemento joint

Este elemento de unión tiene dos atributos:

name (Obligatorio)

Especifica un nombre exclusivo para la articulación.

type (Obligatorio)

Indica el tipo de junta en función de las necesidades a cumplir entre las siguientes:

- **“revolute”:** Articulación de bisagra continua que gira a lo largo del eje y presenta un rango de funcionamiento entre el límite superior e inferior.
- **“continuous”:** Articulación de bisagra continua que gira alrededor del eje pero sin límites.
- **“prismatic”:** Junta deslizante a lo largo del eje, rango: [límite superior-inferior].
- **“fixed”:** Todos los grados de libertad están bloqueados, por tanto no es realmente una articulación. Sirve para realizar una unión entre eslabones fija.
- **“floating”:** Articulación que permite el movimiento de los 6 grados de libertad.
- **“planar”:** Articulación que permite el movimiento de un plano perpendicular al eje.

Ejemplo: **<joint name=" " type=" ">**

En cuanto a las características que forman un elemento de unión:

<origin> (Opcional: El valor predeterminado es la identidad,1).

Es la transformación del vínculo principal al secundario. La articulación se encuentra en el origen del vínculo secundario como se observaba en la figura 12.

xyz (Opcional: Por defecto es un vector nulo,[0,0,0]).
Representa la posición en el espacio tridimensional.
rpy (Opcional: Por defecto es el vector nulo,[0,0,0]).

Representa la rotación alrededor de un eje fijo: primero el giro alrededor de “x”, a continuación alrededor de “y” y por último gira alrededor de “z”. Todos los ángulos siempre en radianes.

<parent> (Obligatorio)

Nombre del enlace principal con atributo obligatorio.

<child> (Obligatorio)

Nombre del enlace secundario con atributo obligatorio.

<axis> (Opcional: Por defecto (1,0,0))

Es el eje de rotación para las articulaciones revolucionarias, el eje de traslación para las prismáticas y la superficie normal para las planas. Las juntas fijas y flotantes no utilizan este campo.

xyz (Obligatorio)

<calibration> (Opcional)

Posiciones de referencia de la articulación utilizadas para calibrar su posición absoluta.

rsing (Opcional)

Cuando la articulación se mueve en dirección positiva activando un flanco ascendente.

falling (Opcional)

Cuando la articulación se mueve en dirección negativa activando un flanco descendente.

<dynamics> (Opcional)

Especifica las propiedades físicas de la articulación. Muy útil para simulación

damping (Opcional: Predeterminado en 0)

Valor de amortiguamiento físico de la articulación.

friction (Opcional: Predeterminado en 0)

Valor de fricción estática de la articulación.

<limit> (Obligatorio para articulaciones: revolute y prismatic)

Los límites pueden contener los siguientes atributos:

lower (Opcional: Predeterminado en 0)

Especifica el límite inferior de la articulación. En radianes para las articulaciones revolutas y metros para las prismáticas. Se omite en caso de articulaciones continuas.

upper (Opcional: Predeterminado en 0)

Especifica el límite superior de la articulación. En radianes para las articulaciones revolutas y metros para las prismáticas. Se omite en caso de articulaciones continuas.

effort (Obligatorio)

Encargado de hacer cumplir el máximo esfuerzo conjunto.

velocity (Obligatorio)

Encargado de cumplir la máxima velocidad articular.

<mimic> (Opcional)

Se utiliza para especificar que la articulación definida imita a otra existente. Se puede calcular: valor= multiplicador * other_joint_value + offset.

joint (Obligatorio)

Especifica el nombre de la articulación a imitar.

multiplier (Opcional)

Factor multiplicativo en la fórmula anterior.

offset (Opcional)

<safety_controller> (Opcional)

El controlador de seguridad puede tener los siguientes atributos:

soft_lower_limit (Opcional: Predeterminado en 0)

Especifica el límite inferior de la articulación donde el control de seguridad comienza a limitar la posición de la propia articulación. Este límite debe ser mayor al límite inferior de la articulación.

soft_upper_limit (Opcional: Predeterminado en 0)

Especifica el límite superior de la articulación donde el control de seguridad comienza a limitar la posición de la propia articulación. Este límite debe ser menor al límite superior de la articulación.

k_position (Opcional: Predeterminado en 0)

Especifica la relación entre la posición y los límites de velocidad.

k_velocity (Obligatorio)

Especifica la relación entre el esfuerzo y los límites de velocidad.

- **Transmission:** El elemento de transmisión es una extensión del modelo de descripción de robot (URDF) que se utiliza para describir la relación entre un actuador y una articulación. Gracias a este elemento se pueden modelar relaciones de transmisión y enlaces paralelos. Se pueden conectar múltiples actuadores y articulaciones entre sí utilizando una transformación compleja.

  A continuación, se explican detalladamente todos los elementos que componen una transmisión:

  <type> (Una aparición)

  Especifica el tipo de transmisión.

  <joint> (Una o más apariciones)

  Articulación a la que está conectada la transmisión. La articulación se especifica por su atributo de nombre: **<joint name=" ">** y los subelementos:

  <hardwareInterface> (Una o más apariciones)

  Especifica una interfaz de hardware de espacio articular compatible. Su valor debe ser (***EffortJointInterface***) cuando la transmisión se carga en Gazebo.

  <actuator> (Una o más apariciones)

  Actuador al que está conectada la transmisión. El actuador se especifica por su nombre: **<actuator name= " ">** y los subelementos:

  <mechanicalReduction> (Opcional)

  Especifica una reducción mecánica en la transmisión (articulación/actuador).

- **Gazebo:** Gazebo es un simulador de entornos 3D que posibilita evaluar el comportamiento de un robot en un mundo virtual. Permite, entre muchas otras opciones, diseñar robots de forma totalmente personalizada, crear mundos virtuales usando herramientas CAD e importar modelos ya creados. Además es posible sincronizarlo con ROS de forma que los robots publiquen la información de sus sensores en nodos, así como implementar una lógica y un control que dé ordenes al robot.

El formato de descripción robótica unificada (URDF) no se puede usar en Gazebo (a diferencia del visualizador RVIZ), pero si que es posible convertirlo añadiendo algunas etiquetas adicionales específicas de simulación para que funcione correctamente con Gazebo. Realizando estas modificaciones Gazebo convertirá el **URDF** a **SDF** automáticamente.

Con el objetivo de asimilar correctamente los conceptos anteriormente explicados, se incorpora a continuación el código empleado para introducir el primer ***link*** o eslabón y un par de ***joints*** o articulaciónes del robot Helium. Es importante comentar que no ha sido necesario incorporar todas las características, como por ejemplo, inercia o colisión. Esto se debe a la ventaja de utilizar el visualizador RVIZ en lugar de un simulador como por ejemplo Gazebo, que necesita obligatoriamente el empleo de inercias.

Ejemplo link: “base_link” exoesqueleto Helium:

```
<link name="base_link">
   <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
     <sphere radius="0.002"/>
    </geometry>
    <material name="black_plastic"/>
   </visual>
 </link>
```

En el ejemplo se observa perfectamente la forma de proceder a la hora de crear un archivo urdf. Merece la pena destacar como se introduce dentro del parámetro geometría, el archivo STL que contiene en este caso la base del robot (exportada previamente del modelo creado en formato CAD como bien se explicó en el apartado anterior).

Posteriormente es imprescindible mostrar de igual manera un par de ejemplos de las articulaciones del robot:

Ejemplo joint: “base_link_to_orthosis_link” exoesqueleto Helium:

```
<joint name="base_link_to_orthosis_link" type="fixed">
  <parent link="base_link"/>
  <child link="orthosis_link"/>
  <origin xyz="0 0.040892 0.091666" rpy="0 3.14 0"/>
</joint>
```

El método es siempre el mismo para las articulaciones, se debe definir el nombre de los eslabones padre e hijo sobre los que actúan. Para simplificar la lectura y entendimiento se recomienda otorgarle el nombre a la articulación de los dos eslabones que relaciona entre sí. Además se debe indicar el tipo de articulación, en este caso, *fixed* porque se trata de una articulación fija. Y por último, ubicar las coordenadas correspondientes de igual manera que en el caso de los eslabones.

Se añade otro ejemplo de articulación utilizado, el objetivo es poder comparar entre los diferentes tipos:

```
<joint name="finger1_link1_to_finger1_link2" type="revolute">
  <parent link="finger1_link1"/>
  <child link="finger1_link2"/>
  <origin xyz="0.057191 -0.047503 -0.032499" rpy="0 1.75 0"/>
  <limit lower="0" upper="0.75" effort="10" velocity="3"/>
</joint>
```

Para finalizar, simplemente repetir el proceso tantas veces como sea necesario, es decir, tantas veces como eslabones tenga el robot en cuestión. Sin olvidarse de cerrar el archivo, en este caso el comando del robot.

```
</robot>
```

5.2.2.- CONTROL DEL ROBOT VIRTUAL EN TIEMPO REAL

El objetivo de este capítulo, es explicar los dos posibles métodos de control del robot virtual implementado en el presente trabajo. A lo largo del mismo, se explicará mejor la forma de configurar el sistema para activar un método u otro, más concretamente en el capítulo (5.2.3.1.- ARCHIVO LAUNCH).

1. El primer método consiste en controlar únicamente el robot virtual, sin necesidad de disponer del robot real, utilizando un par de nodos y temas predefinidos en ROS. En la figura 16 se puede observar dicho proceso gráficamente.

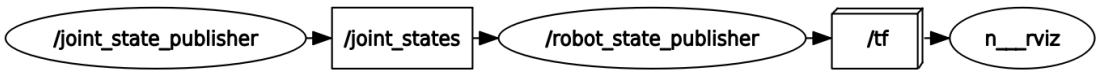

Figura 16: Proceso gráfico para controlar únicamente el robot virtual

El nodo ***joint_state_publisher*** publica mensajes ***sensor_msgs / jointstate*** para el robot. Lee el parámetro ***robot_description***, encuentra todas las articulaciones no fijas y publica un mensaje ***jointstate*** con todas esas articulaciones definidas. Además, se ejecuta el nodo ***robot_state_publisher*** a través del tema ***joint_states*** para publicar también las transformaciones para todos los estados conjuntos.

Es importante destacar que el nodo principal ***joint_state_publisher*** se dividió a principios del presente año (2020), formando un nuevo nodo independiente llamado ***joint_state_publisher_gui*.** Presisamente este último nodo es el que permite controlar el movimiento del robot virtual por medio de una ventana emergente que muestra las posiciones de las articulaciones como controles deslizantes. Cada control deslizante presenta unos límites que coinciden con los límites mínimo y máximo de las articulaciones, excepto en el caso de las articulaciones continuas, que tienen un rango comprendido entre (-π y + π).

Por último simplemente se conecta mediante el tema **tf** a **rviz.**

2. El otro método es un poco más complejo, consiste en asociar el movimiento del robot virtual al del robot real. Para lograr este proceso es necesario disponer del exoesqueleto de mano Helium, además de la librería de comunicación implementada en ROS. A continuación, se puede visualizar el proceso con la ayuda de la figura 17.

Figura 17: Esquema de conexiones de los nodos y topics encargados del control del robot virtual

El funcionamiento se puede resumir para que resulte facilmente comprensible. Se utiliza el nodo creado (***helium.py***) para publicar la posición a la que se manda mover el robot Helium, mediante un tema o topic creado (***joint_states_calculado***) asociado al nodo ***joint_state_publisher***. Para ser más concretos, se publica la posición introducida desde el terminal en la característica posición (***float64[ ] position***) del mensaje anteriormente explicado (***sensor_msgs / jointstate***). La gran diferencia con el método anterior, es el uso del control del robot real, en lugar del nodo ***joint_state_publisher_gui*.**

En el próximo apartado se explica la cinemática directa del robot, necesaria para el correcto funcionamiento del robot virtual.

5.2.2.1.- CINEMÁTICA DIRECTA DEL ROBOT

La cinemática estudia el movimiento que realiza el robot con respecto a un sistema de referencia sin considerar las fuerzas que intervienen.

En otras palabras, para posicionar nuestro robot, debemos conocer cómo mover y orientar todas sus articulaciones para lograr alcanzar la posición y dirección deseada, y de esto es precisamente de lo que se encarga la cinemática.

Existen dos problemas fundamentales: El cinemático directo y el inverso.

Este trabajo se centra en la **cinemática directa**, consiste en determinar la posición y orientación del extremo final de nuestro robot con respecto a un sistema de coordenadas que se toma como referencia. Para ello hemos de conocer los valores de las articulaciones y los parámetros geométricos de los elementos que componen el robot.

La cinemática directa se refiere al uso de ecuaciones cinemáticas para calcular la posición de su actuador final a partir de ciertos parámetros. Existen diferentes métodos para el análisis de la cinemática directa, como por ejemplo: Transformación de matrices, geometría y transformación de coordenadas.

El presente trabajo se basa en la geometría, método en el que es necesario buscar una relación geométrica que permita definir una fórmula matemática para el cálculo del extremo. Esta fórmula puede ser relativamente fácil de localizar, pero se puede llegar a complicar demasiado, haciéndola incluso poco operativa.

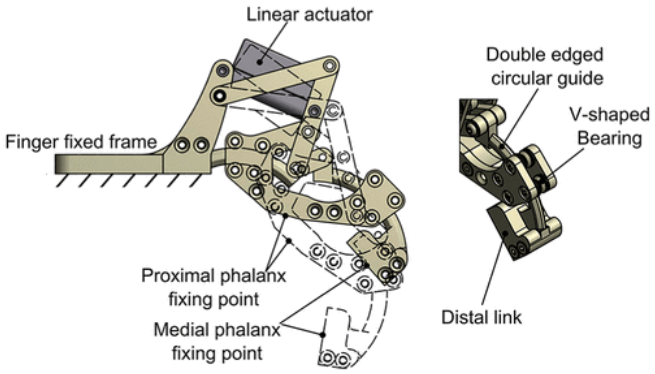

Figura 18: Cadenas cinemáticas dedo (exoesqueleto HELIUM)

En la figura 18, se puede observar que el mecanismo esta formado por dos enlaces paralelos que dan lugar a dos cadenas cinemáticas cerradas. A pesar del sistema no lineal de ecuaciones con múltiples soluciones que se presenta, se ha optado por aplicar una aproximación lineal [7] entre la carrera del actuador (s) y los ángulos articulares (q1 y q2).
Obteniendo de esta manera dos ecuaciones lineales (1) y (2) con un error inferior al 1%. En estas expresiones el trazo se expresa en milímetros y los ángulos de las articulaciones en grados, con qi=0 cuando las falanges se alinean con las anteriores.

$$q1 = 1.40\ s + 14.9 \quad (1)$$

$$q2 = 1.19\ s + 32.32 \quad (2)$$

Estas ecuaciones lineales son las utilizadas en el nodo principal creado en ROS (helium.py), cuya función es transmitir al robot virtual la posición de sus eslabones gracias al ángulo obtenido de sus articulaciones en función del recorrido del actuador introducido.

5.2.3- VISUALIZACIÓN 3D ROS/RVIZ

Una vez alcanzado este punto del trabajo, ya está el sistema preparado para ejecutar correctamente la visualización del robot (exoesqueleto de mano Helium) con su correspondiente funcionamiento. Para ello es necesario explicar una última herramienta que facilita mucho esta tarea.

5.2.3.1.- ARCHIVO LAUNCH

Los archivos de lanzamiento son muy comunes en ROS tanto paras usuarios como para desarrolladores. Proporcionan una forma conveniente de iniciar varios nodos y un maestro, así como otros requisitos de inicialización, como por ejemplo, la configuración de parámetros.

Los archivos de lanzamiento tienen una extensión (.launch) y utilizan un formato XML específico. Se pueden situar en cualquier lugar dentro del directorio del paquete, pero es más correcto crear un directorio específico llamado ***launch*****,** como se puede observar en la figura 19.

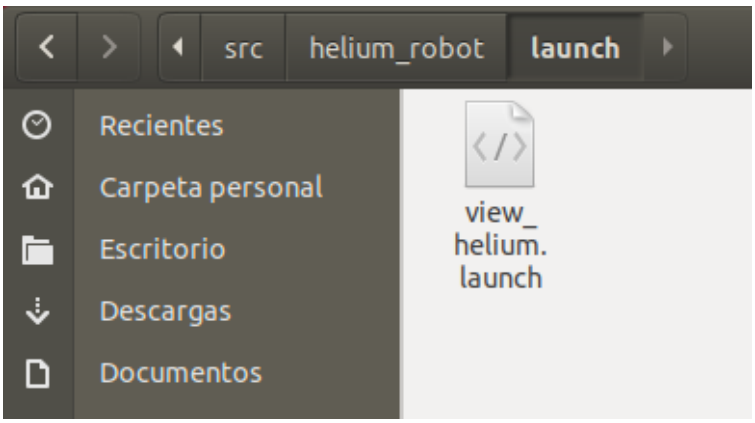

Figura 19: Carpeta launch que contiene el archivo de lanzamiento "view_helium.launch"

Roslaunch es el comando utilizado para abrir estos archivos de inicio. Hay dos formas posibles de ejecutarlo, especificando el paquete en el que están contenidos los archivos de lanzamiento seguido del nombre del archivo de lanzamiento, o especificando la ruta del archivo de lanzamiento:

```
roslaunch package_name launch_file

roslaunch ~/.../.../.../launch_file
```

Antes de comenzar a explicar su estructura es conveniente comentar unas características basicas:

- Lo primero que hace el comando de ejecución ***roslaunch***, antes de iniciar cualquier nodo, es determinar si ***roscore*** ya se está ejecutando y, en caso contrario, lo iniciará automáticamente.
- Todos los nodos de un archivo de inicio se arrancan aproximadamente al mismo tiempo. Por tanto es imposible estar seguro del orden de inicialización de los mismos.

- Por defecto, la salida estándar de los nodos lanzados no es el terminal sino un archivo de registro *(~/.ros/log/run_id/node)*.
- Para finalizar un lanzaiento activo de roslaunch, se usa Ctrl-C. Este comando intentará cerrar correctamente cada nodo activo.

Estructura del archivo de lanzamiento:

1. El elemento **root: <launch> … </launch>**

   Como todos documentos XML, los archivos de lanzamiento tienen un elemento raíz, en este caso, llamado launch. Todos los demás elementos del archivo de lanzamiento deben estar comprendidos entre estas dos etiquetas launch.

2. El elemento **node: <node> … </node>**

   Los nodos lanzados desde este archivo presentan una serie de atributos:

   - **pkg (Obligatorio)**

     Nombre del paquete al que pertenece el nodo
   - **type (Obligatorio)**

     Nombre del archivo ejecutable que inicia el nodo
   - **name (Obligatorio)**

     El nombre del nodo, este nombre tendrá prioridad sobre el que se le asigna en el nodo. Hay que tener cuidado con los nombres de recursos gráficos para no crear conflictos internos.
   - **respawn** (Opcional)

     Recurso que sirve para reiniciar el nodo cuando finaliza. Se activa estableciendo su valor en “true”.
   - **required** (Opcional)

     Si se etiqueta un nodo como obligatorio, cuando finaliza, roslaunch finaliza todos los demás nodos.
   - **launch-prefix** (Opcional)

     Se utiliza para insertar un prefijo al comienzo de la línea de comando que ejecuta el nodo. Por ejemplo, “xterm-e” inicia una ventana de terminal nueva.
   - **output** (Opcional)

Si se establece el valor en “screen” (para un solo nodo), permite cambiar la salida estándar del nodo al terminal.

- **ns** (Opcional)
  Sirve para establecer un espacio de nombres.

3. El elemento **include: <include> … </include>**
   Este elemento permite incluir el contenido de otro archivo de inicio, incluyendo todos sus nodos y parámetros.

```
<include file = “ruta-al-archivo-de-lanzamiento” />
```

Se puede usar un recurso para no tener que especificar explícitamente la ruta de la siguiente forma:

```
<include file = “$ (buscar nombre-paquete) / launch-file-name” />
```

4. El elemento **arg: <arg> … </arg>**
   Este elemento se utiliza para ayudar a configurar los archivos de inicio. Se define de la siguiente manera:

```
<arg name = “*arg-name*” />
```

Y se usa con una sustitución de arg:

```
$ (arg *nombre-arg*)
```

Una vez explicada la estructura de los archivos de lanzamiento, es importante concluir explicando un par de atributos necesarios para la construcción del archivo de lanzamiento propio del presente trabajo.

Atributos **if** y **unless**:

Todas las etiquetas anteriormente explicadas admiten estos dos atributos encargados de incluir o excluir una etiqueta según la evaluación de un valor. Los valores “**1**” y “**true**” se consideran verdaderos y por el contrario “**0**” y “**false**” se consideran falsos. El resto de valores serán erroneos.

if=valor (opcional)

Si el valor se evalúa como verdadero, incluye la etiqueta y su contenido.

unless=valor (opcional)

A menos que el valor se evalúe como verdadero, es decir, si el valor es falso, incluye la etiqueta y su contenido.

Es importante efectuar una breve explicación del código utilizado en el presente trabajo para inicializar el sistema encargado del control del robot (exoesqueleto de mano Helium). El archivo de lanzamiento completo se puede encontrar en el **Anexo 3.**

En el código creado se incluyen argumentos y parámetros, esto permite crear archivos de lanzamiento más reutilizables y configurables especificando valores pasados a través de línea de comandos. Una declaración arg es específica de un solo archivo de inicio, al igual que un parámetro local en un método.

La línea de código: <arg name “gui” default “true/false” /> se utiliza para en función del valor true o false que se le otorgue, arranque unos nodos u otros.

Se puede observar que si se cumple la condición, es decir, si el valor de arg name “gui” es por defecto true, se arranca el suguiente nodo:

```
<node if=”$(arg gui)” name=”joint_state_publisher”
pkg=”joint_state_publisher_gui” type=”joint_state_publisher_gui”/>
```

Este nodo se utiliza para dotar de movimiento al robot virtual en rviz desde un widget por pantalla como se puede observar en la figura 20.

Figura 20: Movimiento del robot virtual desde widget

Por el contrario, si el valor es false, se arranca otro nodo distinto:

Esta es la correcta configuración para el sistema propuesto en este trabajo.

```
<node unless=”$(arg gui)” name=”joint_state_publisher”
pkg=”joint_state_publisher” type=”joint_state_publisher”/>

<rosparam param=”source_list”> [joint_states_calculado”]</rosparam>
```

Esta última línea de código es muy importante para definir el parámetro que permite crear un topic nuevo dentro del nodo joint_state_publisher. El topic **joint_states_calculado** es el encargado de transmitir el movimiento en tiempo real al robot virtual y se crea en el nodo **helium.py** (Nodo que no interesa crear en el archivo de lanzamiento para poder ejecutarlo de manera independiente).

El resto de código que se observa en la Figura 15, a parte del recién explicado, son los nodos **rviz** y ***robot_state_publisher*** encargados de arrancar el visualizador RVIZ utilizando el archivo URDF especificado por el parámetro *robot_desciption* y las posiciones de las articulaciones del tema o topic *joint_states* para calcular la cinematica de avance del robot y publicar los resultados a través de tf:

```
<param name=”robot_description” textfile=”$(find helium_robot)/urdf/cinematica_exoesqueleto_HELIUM.urdf”/>

<arg name=”rvizconfig” default=”$(find helium_robot)/cfg/view_robot.rviz”/>
```

view_robot.rviz: Archivo encargado de guardar la configuración seleccionada de RVIZ.

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

Como bien se ha explicado con anterioridad en este mismo apartado, existían dos formas de ejecutar los archivos de lanzamiento, en este trabajo se opta por la opción de indicar el paquete en el que se encuentra.

roslaunch+(nombre del paquete)+(nombre del archivo launch)

En el caso particular del sistema:

```
roslaunch helium_robot view_helium.launch
```

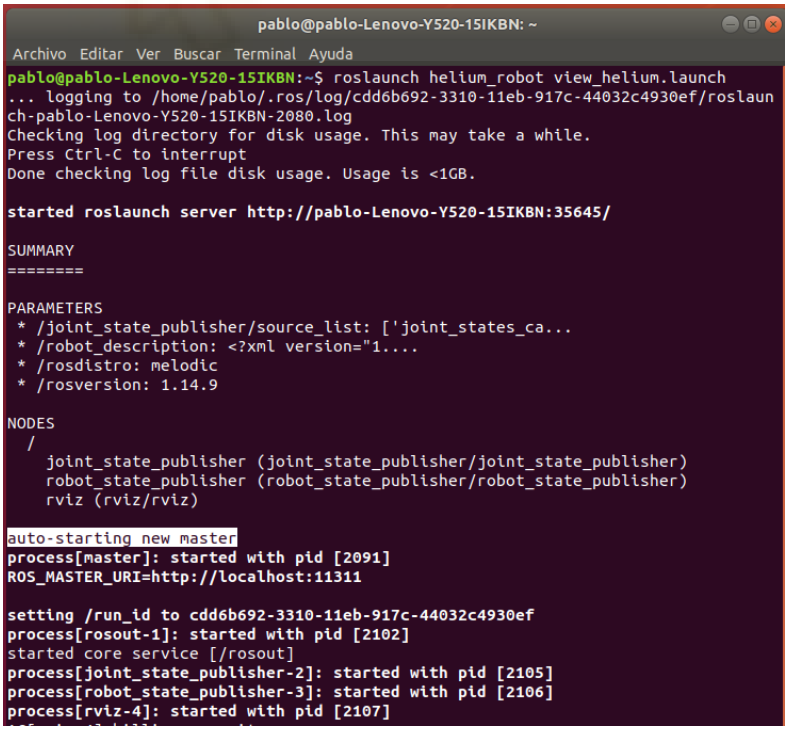

Figura 21: Ejemplo de ejecución del archivo de lanzamiento. (Se observan los nodos arrancados, incluido el MASTER.)

6.- EXPERIMENTACIÓN

6.1.- PUESTA EN MARCHA

En este apartado se detallan todos los pasos necesarios para el correcto funcionamiento del sistema en cuestión.

6.1.1.- ENTORNO DE ROS Y CONFIGURACIÓN

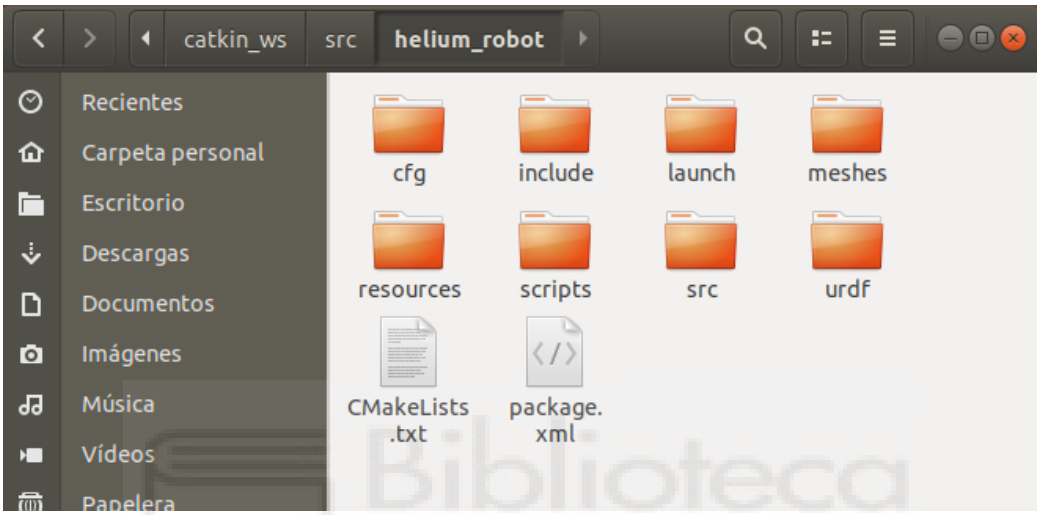

Figura 22: Contenido de la carpeta src/ del espacio de trabajo de Catkin

Este es el paquete de ROS (helium_robot) creado y debidamente cumplimentado a lo largo del presente trabajo que permite el correcto funcionamiento del sistema implementado. Es muy importante su ubicación dentro del espacio de trabajo ya que, al realizar cualquier modificación, por pequeña que sea, para que se tome en consideración es imprescindible compilar el espacio de trabajo o catkin_ws.

Para llevar a cabo la compilación, ROS dispone de dos herramientas de compilación: **catkin_make** y **catkin_build**

Ambas instrucciones son correctas, pero se recomienda utilizar la segunda (catkin_build) ya que proporciona una compilación a alto nivel, es decir, más profunda.

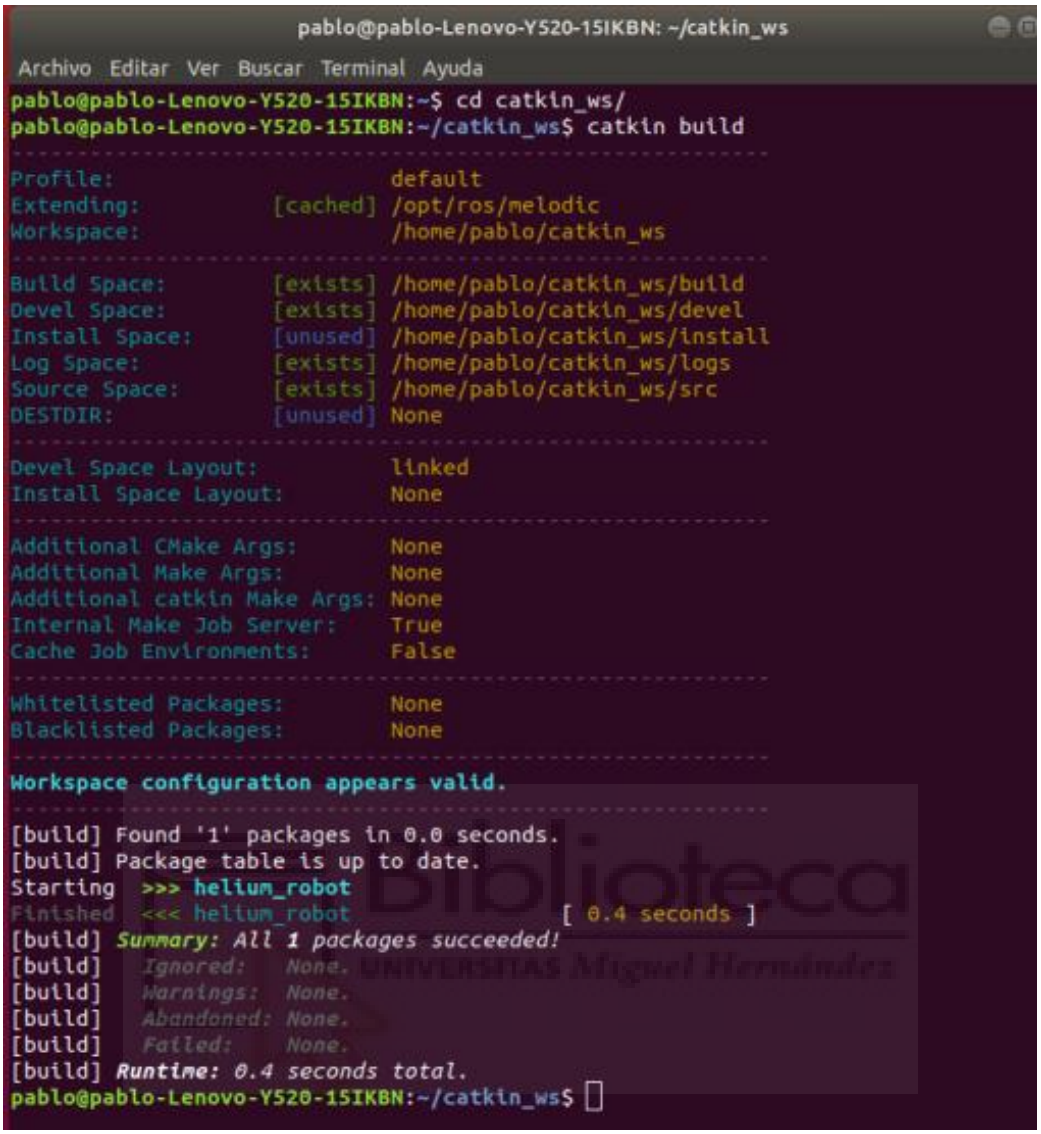

Figura 23: Compilación catkin_build del paquete helium_robot

En este momento, tras realizar una última compilación al espacio de trabajo de ROS, se puede dar por finalizado y listo para ejecutarse. No obstante, antes de comenzar a explicar la conexión necesaria, es importante destacar la tecnología utilizada y los requisitos previos para configurarla de manera adecuada.

La unidad de control encargada de establecer la comunicación entre el ordenador y el dispositivo (Helium) es una Placa de desarrollo USB Teensy. Teensy es un sistema completo de desarrollo de microcontroladores basado en USB, en un espacio muy pequeño, capaz de implementar diferentes tipos de proyectos.

Figura 24: Unidad de Control formada por placa Teensy

Funciona a la perfección con Linux, pero es necesario seguir unos pasos de configuración previos:

1º Copiar el archivo **49-teensy.rules** facilitado por el fabricante para que Linux-Ubuntu sea capaz de detectar el USB del Teensy. En este caso se ha ubicado en la carpeta resources.

2º Instalar las siguientes dependencias de Python para que funcione el script:

```
sudo apt install python-pip

pip install numpy

pip install serial

pip install scipy

pip install pyserial
```

6.1.2.- CONEXIÓN CON EL ROBOT

La conexión con el robot se realiza mediante USB a través de la unidad de control del exoesqueleto. Esta electrónica es la encargada del control y alimentación del exoesqueleto. Para ello, cuenta con una fuente de

alimentación externa, por lo que la comunicación USB es únicamente empleada para el envío y recepción de mensajes de control.

Figura 25: Conexión robot – PC

Se añade a continuación un par de figuras a modo de detalle en las que se puede observar mejor la conexión de los diferentes dedos del exoesqueleto a la unidad de control: D0 para el pulgar, D1 dedo índice, D2 corazón y D3 para el par formado por el dedo anular y meñique.

Figura 26: Conexión dedos del exoesqueleto

Una vez conectado correctamente el exoesqueleto de mano Helium al ordenador con ROS instalado y debidamente configurado según las indicaciones anteriores, se debe ejecutar el archivo launch de la siguiente manera:

```
roslaunch helium_robot view_helium.launch
```

Al ejecutar un archivo de lanzamiento, no es necesario realizar previamente un **roscore**, ya que se arranca automáticamente el nodo maestro al introducir el comando **roslaunch**.

De esta manera se ejecuta la simulación de nuestro robot en RVIZ obteniendo como resultado:

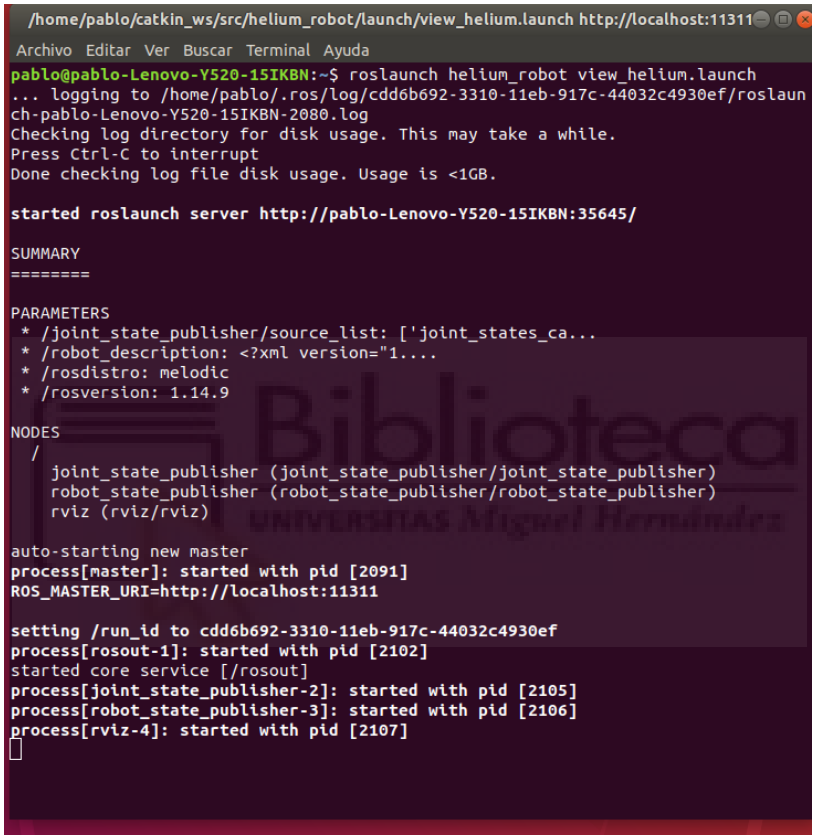

Figura 27: Ejemplo de ejecución del comando: roslaunch helium_robot view_helium.launch

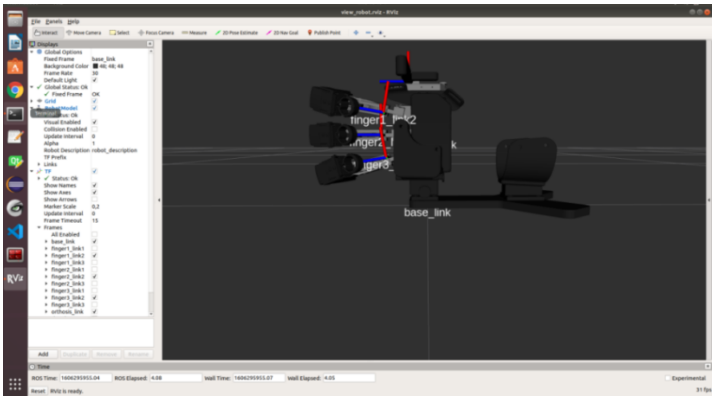

Figura 28: Resultado obtenido tras ejecutar el archivo de lanzamiento

6.1.3.- CONTROL

Para proceder a controlar el dispositivo, se debe abrir un nuevo terminal, sin cerrar el ejecutado con anterioridad para no perder los nodos arrancados ni la simulación en rviz.

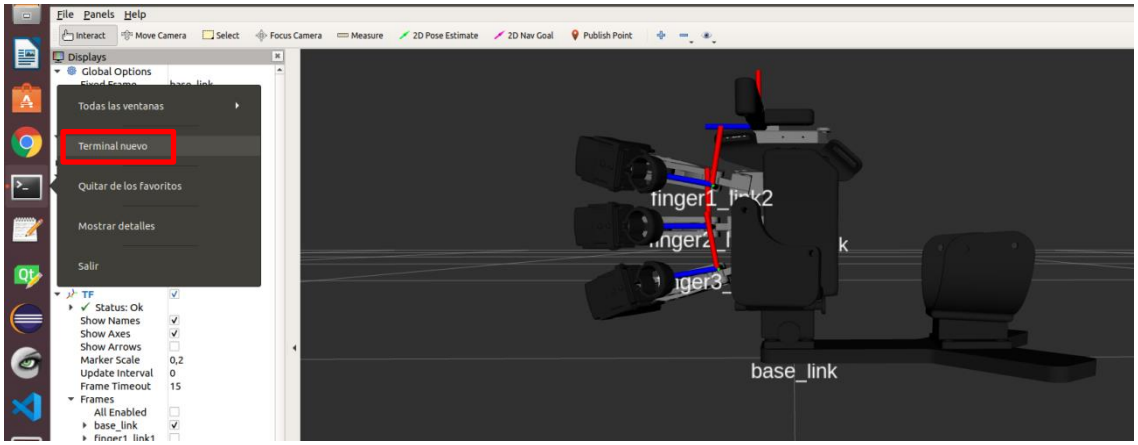

Figura 29: Abrir un terminal nuevo para poder arrancar otro nodo a parte del archivo de lanzamiento

A continuación, hay que arrancar el nodo principal creado ***helium.py*** que se puede observar en su totalidad con explicación incluida en el **Anexo 3**. Este nodo es el encargado de comprobar la correcta comunicación con el dispositivo y de su control, se arranca con el siguiente comando:

```
rosrun helium_robot helium.py
```

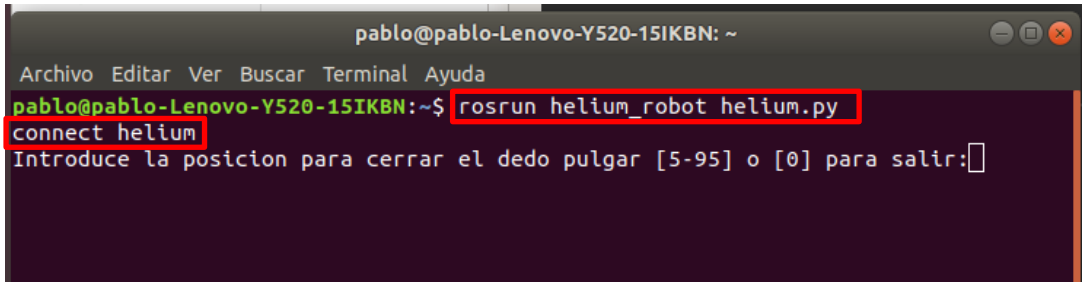

Figura 30: Ejemplo ejecución del nodo helium.py

Se puede observar en la figura 30 como al ejecutar el nodo, aparece por pantalla el mensaje “connect helium”, esto indica que se ha establecido la comunicación correctamente. Si en algún momento se perdiera la conexión aparecería de igual manera un mensaje de error.

Seguidamente se pide al usuario que indique la posición para cerrar cada dedo. Se parte siempre de la misma posición inicial (exoesqueleto de mano completamente abierto). El rango de apertura y cierre de la mano oscila entre el porcentaje (5 – 95), siendo 5 la máxima posición de apertura de la mano y 95 la máxima posición de cierre.

Con el objetivo de mostrar el correcto funcionamiento, se adjunta a continuación una sucesión de figuras en las que se puede observar:

1º En las figuras 31 y 32, como introducir el máximo valor que pueden alcanzar todos los dedos del exoesqueleto, logrando de esta manera su posición de cierre óptimo.

2º En las figuras 33 y 34, como introducir el mínimo valor que pueden alcanzar todos los dedos del exoesqueleto, logrando de esta manera su posición de apertura óptima.

3º Para finalizar, en las figuras 35 y 36, se ha querido demostrar la posibilidad de controlar la posición de cada dedo que compone el exoesqueleto de manera totalmente independiente, es decir, tener un completo control del exoesqueleto para poder realizar prácticamente cualquier movimiento.

Figura 31: Visualización general de la posición de cierre del exoesqueleto

Figura 32: Visualización en detalle de la posición de cierre del exoesqueleto

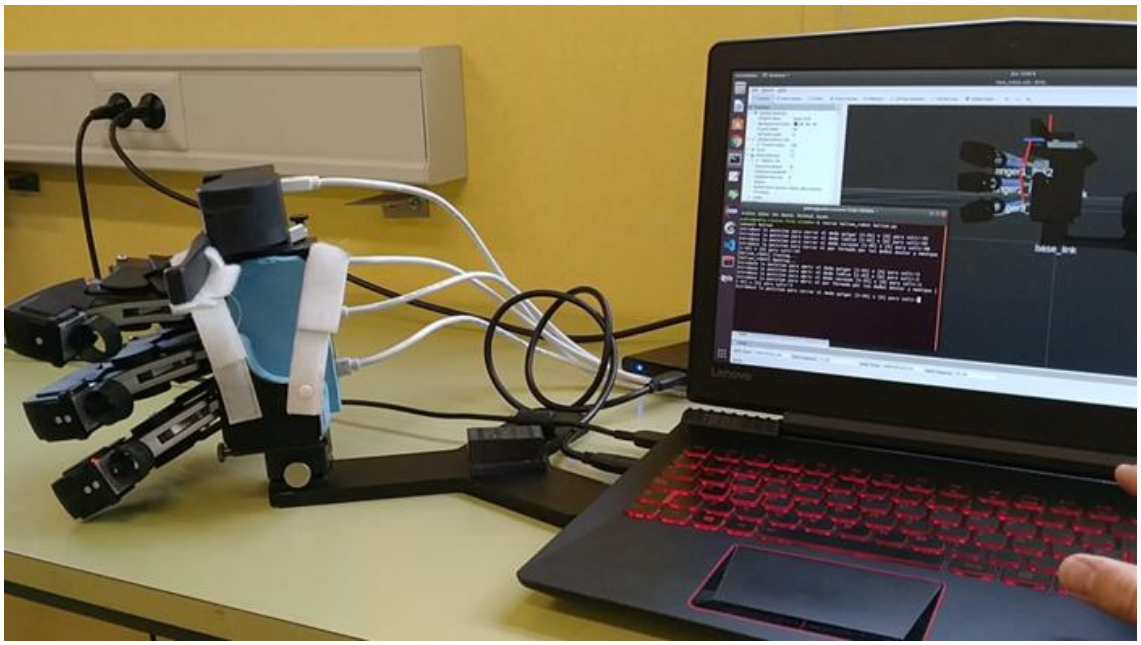

Figura 33: Visualización general de la posición de apertura del exoesqueleto

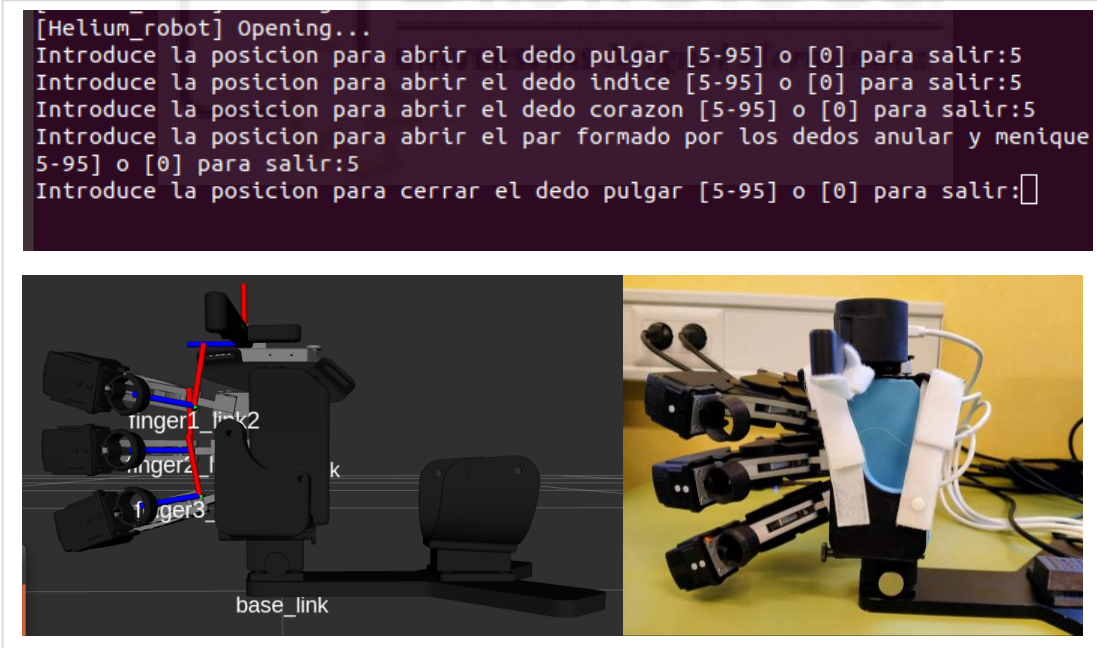

Figura 34: Visualización en detalle de la posición de apertura del exoesqueleto

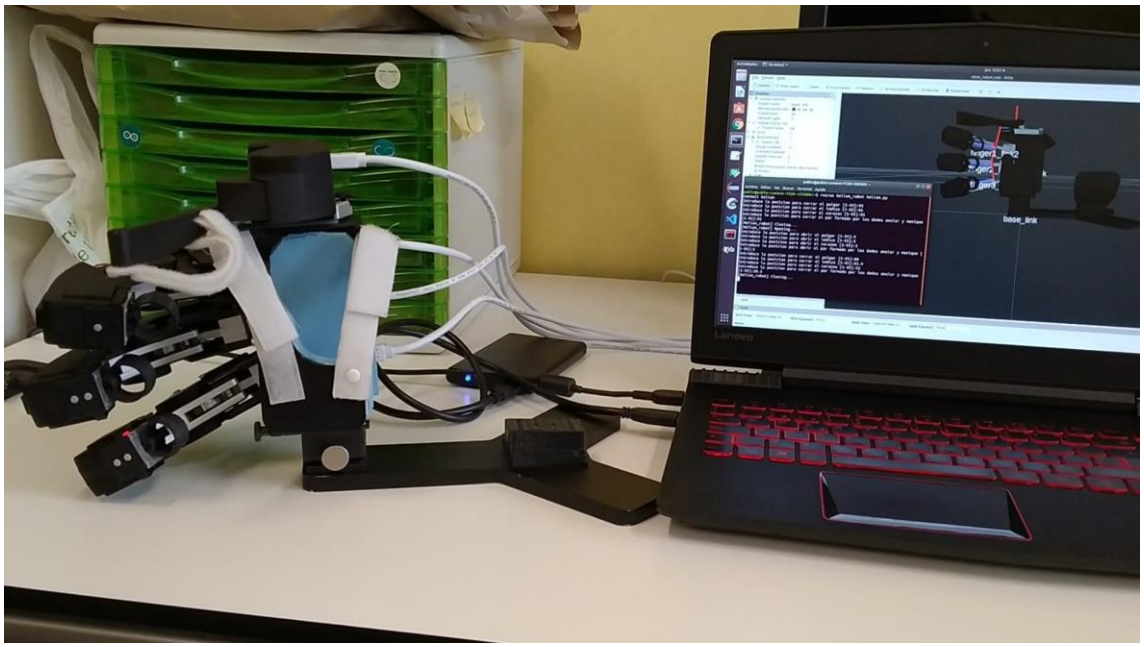

Figura 35: Visualización general de los dedos del exoesqueleto en posiciones diferentes

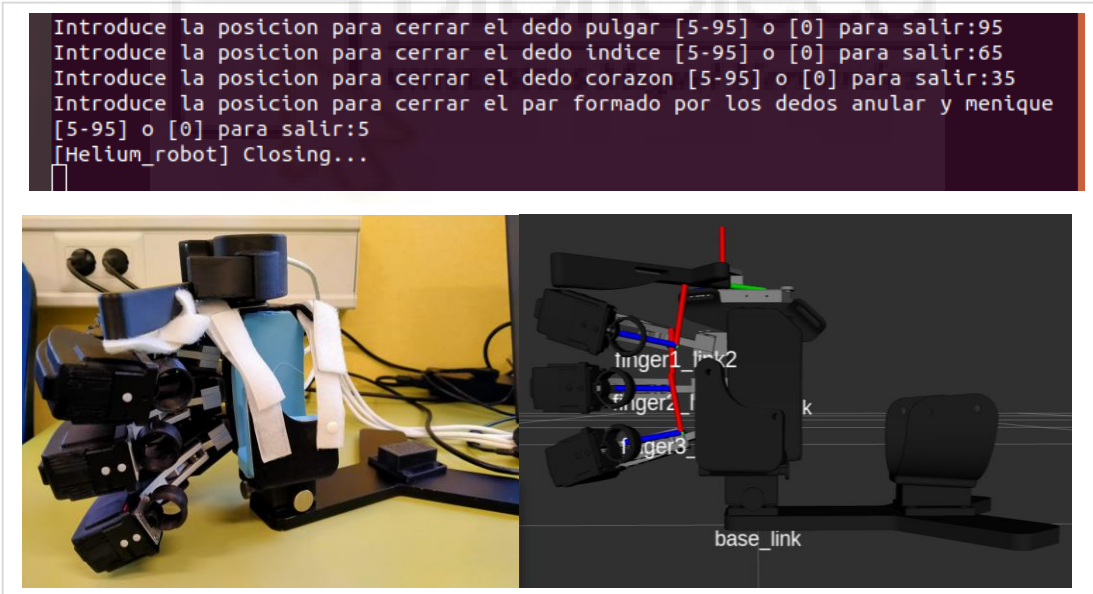

Figura 36: Visualización en detalle de los dedos del exoesqueleto en posiciones diferentes

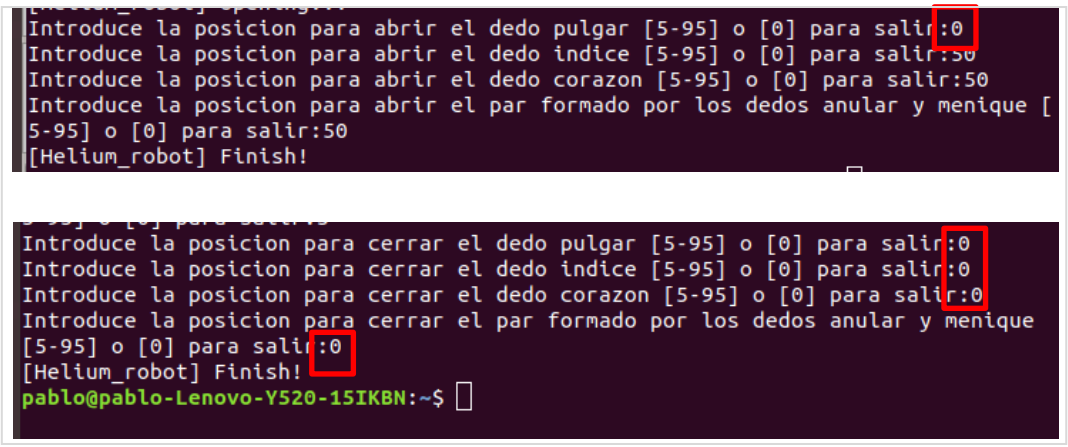

Figura 37: Finalización del control del sistema introduciendo el valor 0 en una o más posiciones

En la figura 37, se observa perfectamente como finaliza el bucle y por tanto se cierra el nodo desconectando el dispositivo al introducir en una o todas las posiciones de los dedos el valor 0.

De igual manera se puede volver a arrancar el nodo sin ningún problema para continuar controlando el dispositivo.

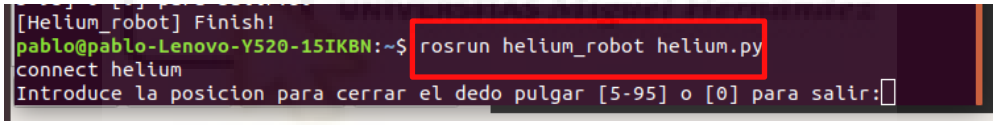

Figura 38: Repetir instrucción de arranque del nodo para volver a controlar el dispositivo

7.- CONCLUSIONES Y LÍNEAS FUTURAS

7.1.- CONCLUSIONES

Como conclusión final del presente trabajo, diría que se ha logrado alcanzar el objetivo final, implementar el exoesqueleto de mano Helium con éxito. Esto implica haber superado con éxito todos y cada uno de los objetivos iniciales planteados al comienzo del trabajo: Profundizar y comprender el funcionamiento de ROS, crear un paquete que engloba los métodos necesarios para visualizar el exoesqueleto de mano Helium, como por ejemplo, construir y programar archivos de descripción URDF, nodos necesarios… Así como comunicar en tiempo real el dispositivo con la simulación virtual y posibilitar su control.

7.2.- LÍNEAS FUTURAS

La implementación de un exoesqueleto robótico en ROS abre un mundo de posibilidades a la hora de trabajar en investigaciones para lograr mejoras considerables. A corto plazo, las principales líneas de trabajo que me gustaría continuar desarrollando sería buscar una implementación integral del dispositivo, es decir, conseguir una virtualización completa para poder trabajar con el robot sin la necesidad de tenerlo físicamente. Este trabajo simplemente es el primer paso para lograr este objetivo mayor.

Por otro lado, se podría continuar mejorando el presente trabajo, creando una interfaz gráfica de usuario que facilite el control del dispositivo de manera más visual a través de por ejemplo una pantalla táctil.

Para finalizar, otra idea interesante podría ser desarrollar una aplicación móvil que interactuara con el exoesqueleto de mano Helium.

8.- REFERENCIAS BIBLIOGRÁFICAS

[1] Schabowsky, C. N., Godfrey, S. B., Holley, R. J., & Lum, P. S. (2010). Development and pilot testing of HEXORR: hand EXOskeleton rehabilitation robot. *J Neuroeng Rehabil, 7(36), 1-16.*

[2] Stein, J., Bishop, L., Gillen, G., & Helbok, R. (2011, June). A pilot study of robotic-assisted exercise for hand weakness after stroke. In Rehabilitation Robotics (ICORR), 2011 IEEE International Conference on (pp. 1-4). IEEE.

[3] Borboni, A., Fausti, D., Mor, M., Vertuan, A., & Faglia, R. Un dispositivo CPM per la riabilitazione della mano.

[4] Santoja Guerrero, I. (2012). Diseño e implementación de sistemas de control aplicados a un exoesqueleto para la rehabilitación de la mano.

[5] Yihun, Y., Miklos, R., Perez-Gracia, A., Reinkensmeyer, D. J., Denney, K., & Wolbrecht, E. T. (2012, August). Single Degree-of-Freedom Exoskeleton Mechanism Design for Thumb Rehabilitation. In Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE (pp. 1916-1920). IEEE.

[6] SC Enriquez, Y Narváez, OA Vivas, J Diez, FJ Badesa, JM Sabater, N Garcia-Aracil. Neuoringeniería Biomédica (2014). “Sistema robótico de tipo exoesqueleto para rehabilitación de la mano”.

[7] Jorge A Diez, Andrea Blanco, José María Catalán, Francisco J Badesa, Luis Daniel Lledó, Nicolas Garcia-Aracil. (2018). “Hand exoskeleton for rehabilitation therapies with integrated optical force sensor”. Version digital: https://journals.sagepub.com/doi/full/10.1177/1687814017753881

[8] ROS (Robot Operating System). http://wiki.ros.org/ , 2020.

A.- ANEXO 1: INSTALACIÓN LINUX-UBUNTU EN ORDENADOR CON OTRO SISTEMA OPERATIVO

Ubuntu es un sistema operativo de software libre y código abierto y es una distribución de Linux basada en Debian.

La razón por la que se ha utilizado este sistema operativo en el presente trabajo es debido a que Ros está pensado y orientado para un sistema UNIX (Ubuntu (Linux)). Es verdad que hoy en día Ros se está adaptando a otros sistemas operativos, pero actualmente solo se pueden considerar experimentales, ya que presentan fallos. De hecho, se probó a instalar Ros directamente en un ordenador con sistema operativo Windows 10, en el siguiente anexo se explicará mejor este proceso. Finalmente se optó por instalarlo en Ubuntu para obtener el máximo rendimiento posible.

Para instalar un sistema operativo en un ordenador que ya presenta otro distinto se destacan 3 opciones:

1.- Máquina Virtual

Esta opción es de las más populares por comodidad, ya que simplemente se tiene que instalar una máquina virtual de las muchas que hay (gratuitas y versiones de pago) como por ejemplo “VMWare Workstation” que es precisamente con la que se comprobó el funcionamiento. En una máquina virtual se puede seleccionar la cantidad de memoria, capacidad de almacenamiento, tarjeta gráfica… que se desee proporcionar al nuevo sistema operativo, siempre dentro de unos límites relacionados con las capacidades del ordenador. Precisamente eso fue lo que hizo cambiar de idea, ya que este trabajo requiere realizar numerosas visualizaciones y para ello es necesario el máximo rendimiento del ordenador. Con este método perdemos el funcionamiento óptimo al tener que fraccionar dichas propiedades.

A pesar de esto, para muchas personas es una opción fantástica y fácil para trabajar con un sistema operativo dentro de otro.

2.- Partición Disco Duro Interno

Esta segunda opción no es tan conocida y consiste en crear diferentes espacios dentro de un disco de manera que en realidad es como si se tuvieran más de un disco duro, esto se puede utilizar para almacenar datos o instalar otros sistemas operativos.

Partición es el nombre que recibe cada división explicada anteriormente y a pesar de estar dentro del mismo disco duro físico, cada partición presenta su propio sistema de archivos y funciona de manera totalmente independiente.

Existen diferentes tipos de particiones: primarias, extendidas o secundarias y lógicas. Nosotros nos vamos a centrar en las particiones primarias ya que son las únicas en las que se puede instalar un sistema operativo y es en realidad lo que nos incumbe.

Las particiones primarias pueden ser como máximo cuatro por disco duro y son las que detecta automáticamente el ordenador al arrancar, en caso de formateo, por defecto se creará una única partición primaria formada de la capacidad máxima de almacenamiento del disco duro.

Se procede a explicar brevemente los pasos a seguir para realizar dicho proceso:

1º **Abrir el menú de inicio** (Windows 10 en este caso) y escribir:

```
partición
```

De manera que el propio buscador te sugerirá la herramienta ***Crear y formatear particiones del disco duro***. Cuando se vea esta opción simplemente hay que hacer click sobre ella y se abrirá el programa de administración de discos. El resultado debe ser similar al de la figura 34. En esta ventana se observan todos los discos duros instalados en el ordenador.

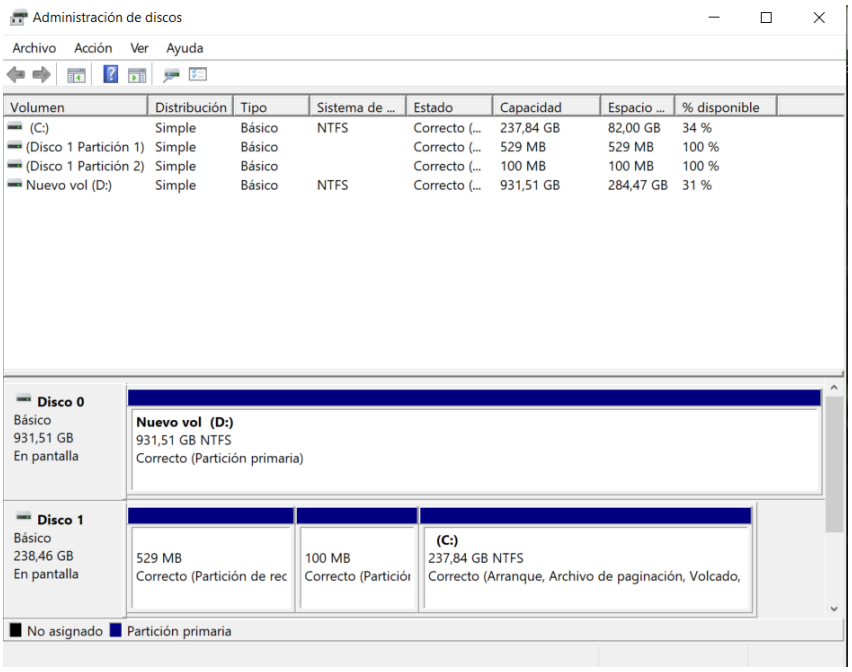

Figura 39: Ejemplo del administrador de discos para comenzar el proceso de partición del disco duro interno deseado

2º Para hacer una nueva partición es necesario tener hueco disponible. Por eso es necesario hacer **click derecho sobre el disco duro en el que se quiere hacer la partición**, y cuando aparezca el menú desplegable, elegir la opción ***Reducir Volumen***. Hay otras opciones posibles, como por ejemplo, borrar una partición o incluso formatear el disco completo.

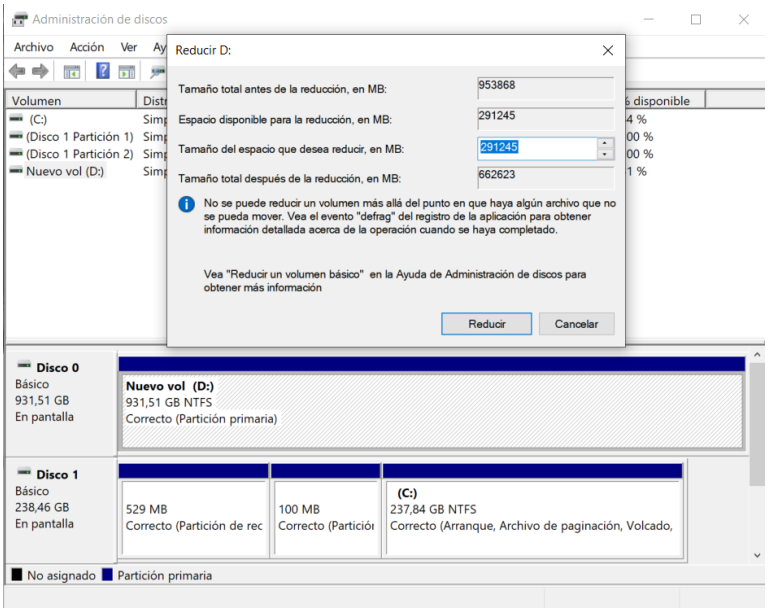

Figura 40: Ventana emergente para reducir el volumen del disco duro

En la figura anterior se muestra la ventana emergente que debe abrirse al seleccionar la opción anterior. En este paso hay que poner cuánto espacio se quiere dejar libre, teniendo en cuenta que el tamaño total del disco y considerando que las unidades son MB.

Una vez finalizado el procedimiento aparecerá en negro una zona llamada (No asignado), que es la capacidad del disco duro disponible para crear nuevas particiones.

3º Abrir un asistente para crear una nueva partición, para ello hay que hacer click derecho sobre el espacio libre creado en el apartado anterior como se observa en la figura 41 y seleccionar la opción ***Nuevo volumen simple***.

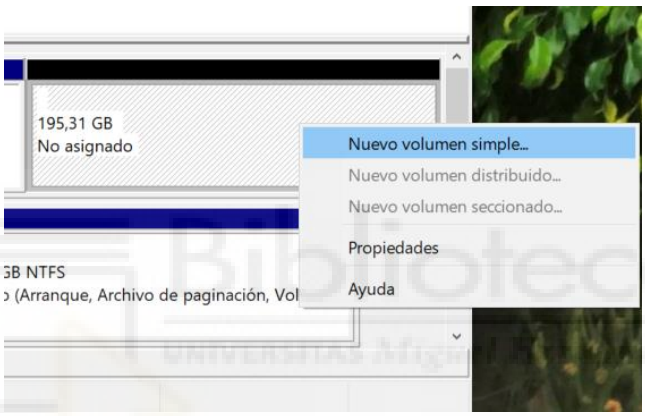

Figura 41: Ejemplo para abrir el asistente de creación de nueva partición de disco duro

4º Para finalizar simplementehay que aceptar el asistente de creación de una nueva partición y seguir las indicaciones:

- Elegir el tamaño de la nueva partición del disco duro, por defecto aparecerá la cantidad total disponible.
- Asignar una letra a modo de nombre para la nueva partición. En este paso se le asignará una letra en caso de querer usar la nueva partición como unidad de almacenamiento de Windows. Si por el contrario se pretende instalar un sistema operativo diferente es mejor no asignarla.
- Por último, aparecerála opción de formatear la partición (totalmente recomendable). También se puede elegir el sistema de archivos que

se quiere utilizar e incluso un hueco en el que escribir un nombre para hacer reconocible la partición.

5º Al finalizar el proceso anterior simplemente aparecera un resumen de la configuración seleccionada para la nueva partición del disco duro interno creada como se puede apreciar en la figura 42.

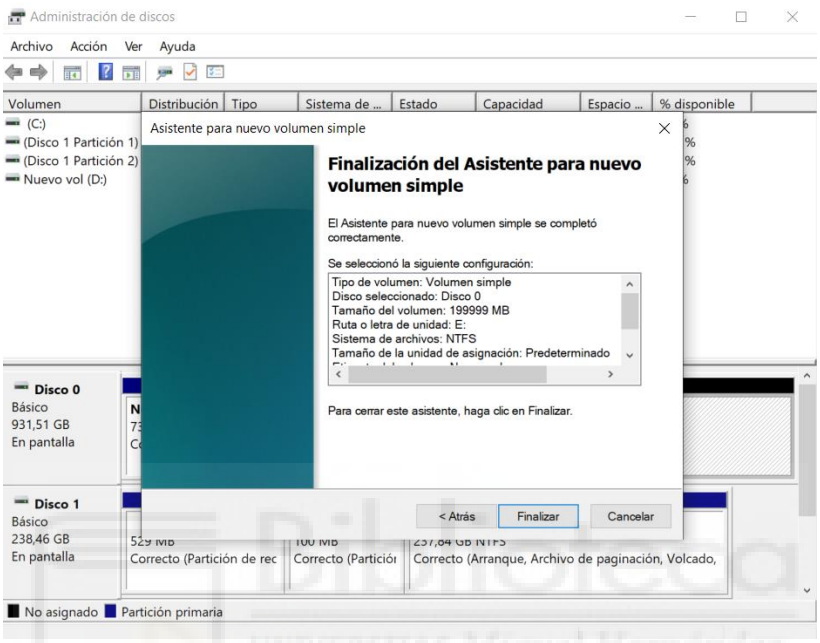

Figura 42: Ejemplo de la configuración elegida antes de finalizar el asistente de creación de partición del disco duro

6º Al pulsar en finalizar aparecera la ventana inicial del administrador de discos con la nueva partición creada con éxito. En este caso se trata de la particion (E:) que se puede observar en la siguiente figura:

Figura 43: Resultado final con la nueva partición creada con éxito

3.- Disco Duro Externo o USB gran capacidad (Mejor opción)

Esta tercera y última opción es similar a la anterior pero con la clara ventaja de tener bien diferenciado físicamente cada disco duro con su respectivo sistema operativo. Lo único que se debe hacer es conectar vía USB el disco duro externo o pen drive al encender el ordenador y abrir la BIOS para de esta manera poder elegir la opción de arrancar desde el externo.

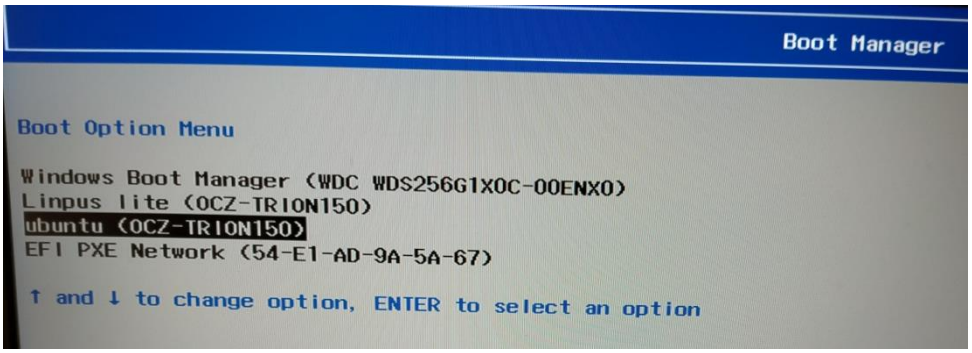

Figura 44: MENÚ BIOS para arrancar UBUNTU desde el disco duro externo

Existen unas carcasas que tienen como función convertir cualquier disco duro interno, independientemente de la marca, en un disco duro externo para conectar por vía USB. Esto hace que sea una opción fantástica a la hora de tener diferenciados claramente distintos sistemas operativos y poder ejecutarlos sin ninguna dificultad.

Figura 45: Carcasa para convertir disco duro interno en externo

B.- ANEXO 2: INSTALACIÓN ROS

VERSIÓN ROS	FECHA LANZAMIENTO	VERSIÓN PRINCIPAL LINUX	FECHA FINALIZACIÓN
NOETIC NINJEMYS	23-05-2020	UBUNTU 20.04 (FOCAL)	MAYO, 2025
MELODIC MOREINA	23-05-2018	UBUNTU 18.04 (BIONIC)	MAYO 2023
LUNAR LOGGERHEAD	23-05-2017	UBUNTU 17.04 (ZESTY)	MAYO 2019
KINETIC KAME	23-05-2016	UBUNTU 16.04 (XENIAL)	ABRIL 2021
JADE TURTLE	23-05-2015	UBUNTU 15.04	MAYO 2017
INDIGO IGLOO	22-07-2014	UBUNTU 14.04 (TRUSTY)	ABRIL 2019
HYDRO MEDUSA	04-09-2013	UBUNTU 12.04 (PRECISE)	MAYO 2015
GROOVY GALAPAGOS	31-12-2012	UBUNTU 12.04 (PRECISE)	JULIO 2014

Figura 46: Tabla Distribuciones ROS-UBUNTU

Antes de comenzar a instalar Ros, lo más importante es conocer la versión de Linux que se ha instalado con anterioridad, en este caso, Ubuntu 18.04 (bionic). No existe ningún motivo especial para decidirse por esta versión, simplemente por tener un par de años de recorrido desde su lanzamiento y presentar soporte hasta el año 2023.

En esta tabla se puede observar la necesidad de instalar para esta versión de sistema operativo (Ubuntu 18.04) la distribución de **Ros Melodic Moreina**.

La página oficial de Ros, ofrece gratuitamente diferentes tutoriales, entre los cuales se encuentra el tutorial de instalación, con todos los pasos específicos a seguir para cada distribución.

A continuación se detallan todos los comandos que se deben ejecutar a través del terminal de Ubuntu:

1º Configurar el ordenador para aceptar software de packages.ros.org

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

Tras introducir el código anterior aparecerá en pantalla una solicitud de contraseña de usuario que hay que teclear para proporcionar permisos de administrador.

2º Configurar el teclado

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Este proceso no debería generar ningún problema, pero existen dos de soluciones a posibles errores:

1. Sustituir “hkp: //pgp.mit.edu: 80 o hkp: //keyserver.ubuntu.com: 80” en su mismo lugar del comando anterior.
2. Sustituir el código entero por el siguiente.

```
curl -sSL 'http://keyserver.ubuntu.com/pks/lookup?op=get&search=0xC1CF6E31E6BADE8868B172B4F42ED6FBAB17C654' | sudo apt-key add -
```

3º Asegurarse que el paquete Debian está correctamente actualizado

```
sudo apt update
```

4º Instalar una configuración predeterminada: Ros recomienda la instalación completa de escritorio ya que incluye numerosas bibliotecas genéricas de robótica, simuladores…

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

Este proceso puede durar unos cuantos minutos dependiendo del ordenador, a continuación se puede observar una imagen.

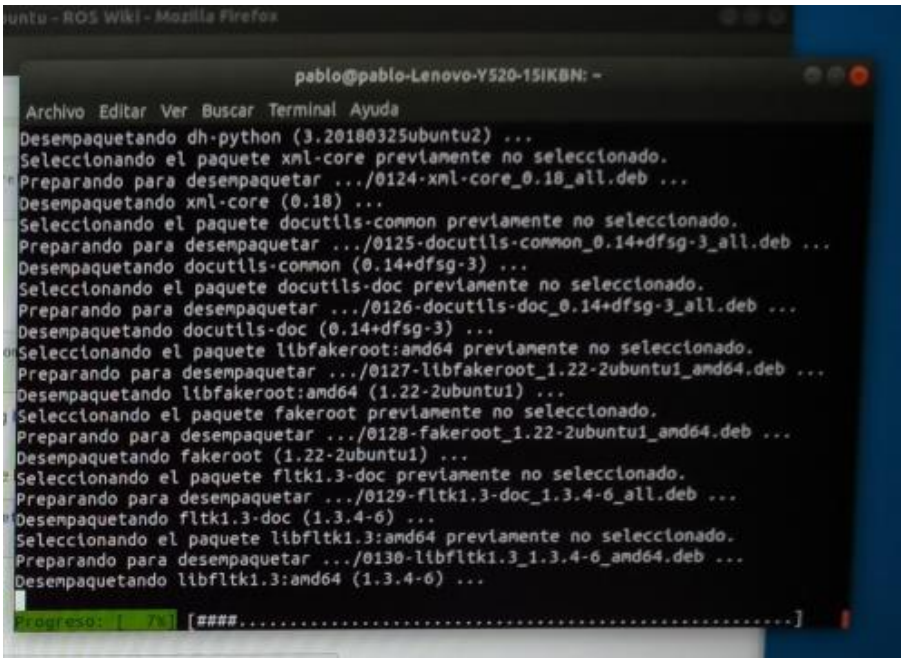

Figura 47: Progreso instalación ROS

Una vez finalizado, ya tenemos instalado Ros Melodic en nuestro sistema operativo ubuntu.

Es muy importante una vez instalado Ros, **crear** y **configurar** correctamente su **espacio de trabajo**, esto es algo que no se explica de forma clara y concisa en la mayoría de las fuentes investigadas

Creación del espacio de trabajo (catkin_ws):

Para crear el espacio de trabajo de ROS se deben introducir en la carpeta raíz del terminal las siguientes instrucciones:

```
mkdir catkin_ws

cd catkin_ws

mkdir src

catkin_make
```

Esto creará una carpeta llamada **catkin_ws** en archivos del ordenador compuesta por tres subcarpetas:

1. BUILD
2. DEVEL
3. SRC: En esta subcarpeta deben estar todos los paquetes que se quieran compilar con éxito.

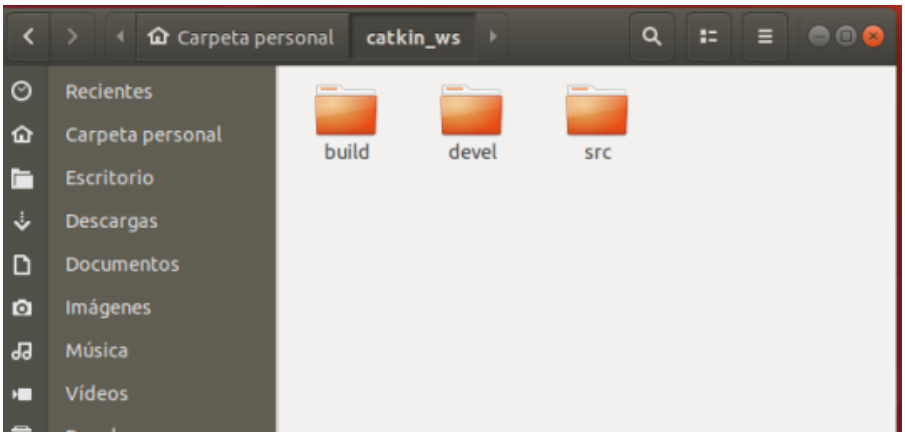

Figura 48: Espacio de trabajo de Catkin

Configuración del espacio de trabajo:

Ros está diseñado para trabajar únicamente en un espacio de trabajo denominado *catkin_workspace* o *catkin_ws*. Para que este espacio de trabajo se ejecute automáticamente sin necesidad de añadir ningún tipo de código, cada vez que se abra un nuevo terminal simplemente se debe hacer lo siguiente:

Desde el directorio raíz del terminal introducimos el siguiente comando:

```
gedit.bashrc
```

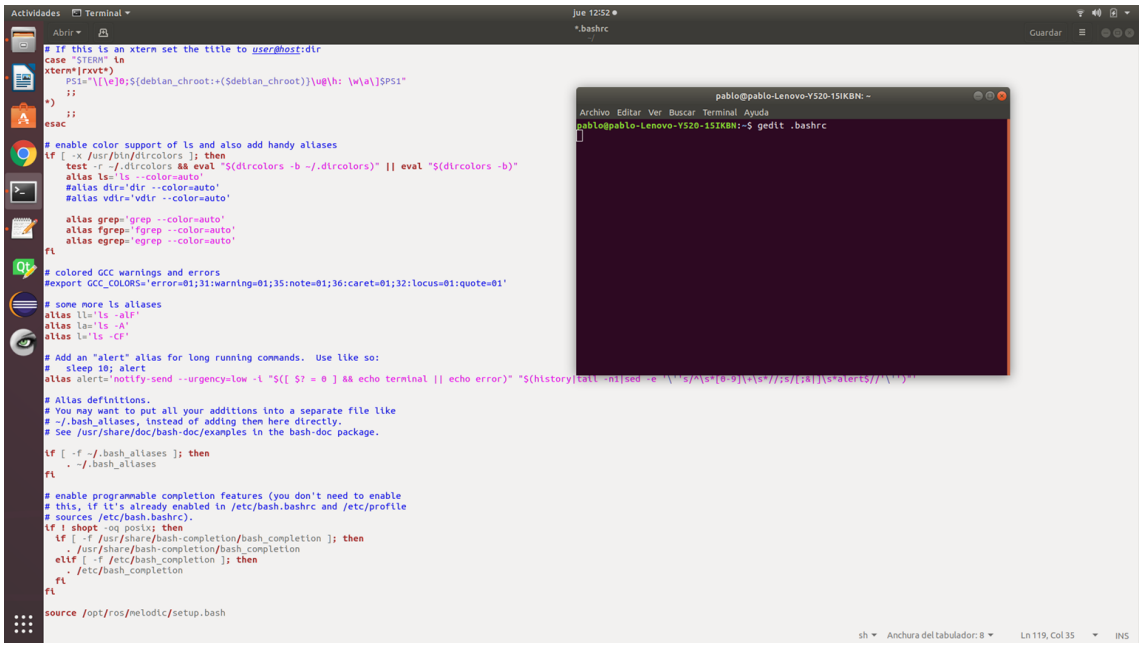

Figura 49: Editor de textos (gedit)

Esto abrirá el archivo .bashrc con el editor de textos **gedit** y se añade en la última línea del documento la siguiente instrucción:

```
source /home/pablo/catkin_ws/devel/setup.bash
```

De esta forma no es necesario introducir cada vez que se abre el terminal el código “source devel/setup.bash” para actualizar los paquetes del directorio src de nuestro espacio de trabajo.

C.- ANEXO 3: CÓDIGOS DE PROGRAMACIÓN

Código archivo: “**cinematica_exoesqueleto_HELIUM.urdf**”

```
<?xml version="1.0"?>
<robot name="cinematica_exoesqueleto_HELIUM">

 <material name="aluminum">
  <color rgba="0.5 0.5 0.5 1.0"/>
 </material>

  <material name="black_plastic">
  <color rgba="0.12 0.12 0.12 1.0"/>
 </material>

 <link name="base_link">
  <visual>
   <origin rpy="0 0 0" xyz="0 0 0"/>
   <geometry>
    <sphere radius="0.002"/>
   </geometry>
   <material name="black_plastic"/>
  </visual>
 </link>

 <joint name="base_link_to_orthosis_link" type="fixed">
  <parent link="base_link"/>
  <child link="orthosis_link"/>
  <origin xyz="0 0.040892 0.091666" rpy="0 3.14 0"/>
 </joint>

 <link name="orthosis_link">
  <visual>
   <origin rpy="0 0 0" xyz="0 0 0"/>
   <geometry>
    <mesh filename="package://helium_robot/meshes/base.stl" scale="0.001 0.001 0.001"/>
```

```
    </geometry>
    <material name="black_plastic"/>
   </visual>
  </link>

  <joint name="orthosis_link_to_finger1_link1" type="fixed">
   <parent link="orthosis_link"/>
   <child link="finger1_link1"/>
   <origin xyz="0 0 0" rpy="0 0 0"/>
  </joint>

  <link name="finger1_link1">
   <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
     <mesh filename="package://helium_robot/meshes/indice_1.stl" scale="0.001 0.001
0.001"/>
    </geometry>
    <material name="aluminum"/>
   </visual>
  </link>

  <joint name="finger1_link1_to_finger1_link2" type="revolute">
   <parent link="finger1_link1"/>
   <child link="finger1_link2"/>
   <origin xyz="0.057191 -0.047503 -0.032499" rpy="0 1.75 0"/>
   <limit lower="0" upper="0.75" effort="10" velocity="3"/>
  </joint>

  <link name="finger1_link2">
   <visual>
    <origin  xyz="0 0 0" rpy="0 -1.75 0"/>
    <geometry>
     <mesh filename="package://helium_robot/meshes/indice_2.stl" scale="0.001 0.001
0.001"/>
    </geometry>
    <material name="aluminum"/>
```

```
  </visual>
 </link>

 <joint name="finger1_link2_to_finger1_link3" type="revolute">
  <parent link="finger1_link2"/>
  <child link="finger1_link3"/>
  <origin xyz="0 -0.005052 0.016564" rpy="0 0 0"/>
  <limit lower="0" upper="0.75" effort="10" velocity="3"/>
 </joint>

 <link name="finger1_link3">
  <visual>
   <origin  xyz="0 0 0" rpy="0 -1.75 0"/>
   <geometry>
    <mesh filename="package://helium_robot/meshes/indice_3.stl" scale="0.001 0.001
0.001"/>
   </geometry>
   <material name="black_plastic"/>
  </visual>
 </link>

 <joint name="orthosis_link_to_finger2_link1" type="fixed">
  <parent link="orthosis_link"/>
  <child link="finger2_link1"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
 </joint>

 <link name="finger2_link1">
  <visual>
   <origin rpy="0 0 0" xyz="0 0 0"/>
   <geometry>
    <mesh filename="package://helium_robot/meshes/corazon_1.stl" scale="0.001 0.001
0.001"/>
   </geometry>
   <material name="aluminum"/>
  </visual>
 </link>
```

```
 <joint name="finger2_link1_to_finger2_link2" type="revolute">
   <parent link="finger2_link1"/>
   <child link="finger2_link2"/>
   <origin xyz="0.060416 -0.044759 -0.00379" rpy="0 1.57 0"/>
   <limit lower="0" upper="0.75" effort="10" velocity="3"/>
 </joint>

 <link name="finger2_link2">
   <visual>
     <origin  xyz="0 0 0" rpy="0 -1.57 0"/>
     <geometry>
      <mesh filename="package://helium_robot/meshes/corazon_2.stl" scale="0.001 0.001
0.001"/>
     </geometry>
     <material name="aluminum"/>
   </visual>
 </link>

 <joint name="finger2_link2_to_finger2_link3" type="revolute">
   <parent link="finger2_link2"/>
   <child link="finger2_link3"/>
   <origin xyz="-0.00015 -0.004536 0.016902" rpy="0 0 0"/>
   <limit lower="0" upper="0.75" effort="10" velocity="3"/>
 </joint>

 <link name="finger2_link3">
   <visual>
     <origin  xyz="0 0 0" rpy="0 -1.57 0"/>
     <geometry>
      <mesh filename="package://helium_robot/meshes/corazon_3.stl" scale="0.001 0.001
0.001"/>
     </geometry>
     <material name="black_plastic"/>
   </visual>
 </link>
```

```
 <joint name="orthosis_link_to_finger3_link1" type="fixed">
  <parent link="orthosis_link"/>
  <child link="finger3_link1"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
 </joint>

 <link name="finger3_link1">
  <visual>
   <origin rpy="0 0 0" xyz="0 0 0"/>
   <geometry>
    <mesh filename="package://helium_robot/meshes/anular_1.stl" scale="0.001 0.001
0.001"/>
   </geometry>
   <material name="aluminum"/>
  </visual>
 </link>

 <joint name="finger3_link1_to_finger3_link2" type="revolute">
  <parent link="finger3_link1"/>
  <child link="finger3_link2"/>
  <origin xyz="0.054729 -0.047992 0.025801" rpy="0 1.4 0"/>
  <limit lower="0" upper="0.75" effort="10" velocity="3"/>
 </joint>

 <link name="finger3_link2">
  <visual>
   <origin  xyz="0 0 0" rpy="0 -1.4 0"/>
   <geometry>
    <mesh filename="package://helium_robot/meshes/anular_2.stl" scale="0.001 0.001
0.001"/>
   </geometry>
   <material name="aluminum"/>
  </visual>
 </link>

 <joint name="finger3_link2_to_finger3_link3" type="revolute">
  <parent link="finger3_link2"/>
```

```
    <child link="finger3_link3"/>
    <origin xyz="0 -0.005252 0.016287" rpy="0 0 0"/>
    <limit lower="0" upper="0.75" effort="10" velocity="3"/>
  </joint>

  <link name="finger3_link3">
    <visual>
      <origin  xyz="0 0 0" rpy="0 -1.4 0"/>
      <geometry>
       <mesh filename="package://helium_robot/meshes/anular_3.stl" scale="0.001 0.001
0.001"/>
      </geometry>
      <material name="black_plastic"/>
    </visual>
  </link>

  <joint name="orthosis_link_to_thumb1" type="fixed">
    <parent link="orthosis_link"/>
    <child link="thumb1"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </joint>

  <link name="thumb1">
    <visual>
      <origin  xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
       <mesh filename="package://helium_robot/meshes/pulgar_1.stl" scale="0.001 0.001
0.001"/>
      </geometry>
      <material name="aluminum"/>
    </visual>
  </link>

  <joint name="thumb1_to_thumb2" type="revolute">
    <parent link="thumb1"/>
    <child link="thumb2"/>
    <origin xyz="0.025781 -0.005759 -0.077734" rpy="0 1.57 0"/>
```

```
    <limit lower="-2.4" upper="0.75" effort="10" velocity="3"/>
  </joint>

  <link name="thumb2">
    <visual>
      <origin  xyz="0 0 0" rpy="0 -1.57 0"/>
      <geometry>
       <mesh filename="package://helium_robot/meshes/pulgar_2.stl" scale="0.001 0.001
0.001"/>
      </geometry>
      <material name="black_plastic"/>
    </visual>
  </link>

</robot>
```

Código archivo: “**view_helium.launch**”

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

  <arg name="gui" default="false" />
  <arg name="rvizconfig" default="$(find helium_robot)/cfg/view_robot.rviz" />

  <param name="robot_description" textfile="$(find
helium_robot)/urdf/cinematica_exoesqueleto_HELIUM.urdf" />

  <node if="$(arg gui)" name="joint_state_publisher" pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" />
  <node unless="$(arg gui)" name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <rosparam param="source_list">["joint_states_calculado"]</rosparam>
  </node>
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" />

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

</launch>
```

Código archivo: “**helium.py**”

```
#! /usr/bin/env python
import controlPanel
import rospy
from std_msgs.msg import Float64
from sensor_msgs.msg import JointState
import time

class robot:
    def __init__(self):
        self.joints = JointState()
        self.joints.position=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        self.joints.name = ['thumb1_to_thumb2',
                            'finger1_link1_to_finger1_link2','finger1_link2_to_finger1_link3',
                            'finger2_link1_to_finger2_link2','finger2_link2_to_finger2_link3',
                            'finger3_link1_to_finger3_link2','finger3_link2_to_finger3_link3']
        self.pub = rospy.Publisher('joint_states_calculado',JointState,queue_size=1)
    def cinematica_directa(self,s):
        q_thumb = -s[0]/100.0
        q1_f1 = (((1.4 * s[1] + 14.9) * 3.14)/180)/5
        q2_f1 = (((1.19 * s[1] + 32.32) * 3.14)/180)/10
        q1_f2 = (((1.4 * s[2] + 14.9) * 3.14) / 180)/5
        q2_f2 = (((1.19 * s[2] + 32.32) * 3.14) / 180)/10
        q1_f3 = (((1.4 * s[3] + 14.9) * 3.14) / 180)/5
        q2_f3 = (((1.19 * s[3] + 32.32) * 3.14) / 180)/10

        self.joints.header.stamp = rospy.Time.now()
        self.joints.position[0] = q_thumb
        self.joints.position[1] = q1_f1
        self.joints.position[2] = q2_f1
        self.joints.position[3] = q1_f2
        self.joints.position[4] = q2_f2
        self.joints.position[5] = q1_f3
        self.joints.position[6] = q2_f3
        self.joints.velocity = []
        self.joints.effort = []
```

```
            self.pub.publish(self.joints)

if __name__ == '__main__':

    rospy.init_node('helium', anonymous = True)

    r = robot()

    # Conectar al HELIUM
    helium = controlPanel.robot()
    helium.connect('ttyACM0')
    helium.streaming()

    # Variables que voy a usar para almacenar las posiciones para la cinematica directa
    thumb = 0.0
    f1 = 0.0
    f2 = 0.0
    f3 = 0.0

    # Este comando es para mover a una posicion (5 - 95)
    # Posicion cerrado
    # Bucle infinito hasta que se cumpla la condición  posterior con la sentencia break
    while  True:
            # Declaración e inicialización de todas las posiciones del vector a cero
             pos=[0,0,0,0]


             pos[0]= input("Introduce la posicion para cerrar el dedo pulgar [5-95] o [0] para
salir:")
             pos[1]= input("Introduce la posicion para cerrar el dedo indice [5-95] o [0] para
salir:")
             pos[2]= input("Introduce la posicion para cerrar el dedo corazon [5-95] o [0] para
salir:")
             pos[3]= input("Introduce la posicion para cerrar el par formado por los dedos anular
y menique [5-95] o [0] para salir:")
# Condición para romper el bucle: cuando cualquier posición toma el valor 0
             if (pos[0] == 0 or pos[1] == 0 or pos[2] == 0 or pos[3] == 0):
                    break
```

```
            #pos = [85, 85, 85, 85]
            helium.move2position(pos)

            print("[Helium_robot] Closing...")

            # rate = rospy.Rate(10) #frecuencia con la que va a trabajar el nodo
            # while not rospy.is_shutdown():
            while f2 <= pos[2]-2:

                thumb = helium.handExoskeleton.handPacket.pos[0] * 100.0 / 8191.0
                f1 = helium.handExoskeleton.handPacket.pos[1] * 100.0 / 8191.0
                f2 = helium.handExoskeleton.handPacket.pos[2] * 100.0 / 8191.0
                f3 = helium.handExoskeleton.handPacket.pos[3] * 100.0 / 8191.0

                # print([thumb, f1, f2, f3])

                r.cinematica_directa([thumb, f1, f2, f3])
                # rate.sleep()

                # Para que vaya un poco más lento
                time.sleep(0.1)

            time.sleep(2.0)

            print("[Helium_robot] Opening...")

            # Posicion abierto
            pos[0]= input(("Introduce la posicion para abrir el dedo pulgar [5-95] o [0] para
salir:")
            pos[1]= input("Introduce la posicion para abrir el dedo indice [5-95] o [0] para
salir:")
            pos[2]= input("Introduce la posicion para abrir el dedo corazon [5-95] o [0] para
salir:")
            pos[3]= input("Introduce la posicion para abrir el par formado por los dedos anular y
menique [5-95] o [0] para salir:")

        if (pos[0] == 0 or pos[1] == 0 or pos[2] == 0 or pos[3] == 0):
```

```
                    break
            #pos = [5, 5, 5, 5]
            helium.move2position(pos)

            while f2 >= pos[2]+2:

                    thumb = helium.handExoskeleton.handPacket.pos[0] * 100.0 / 8191.0
                    f1 = helium.handExoskeleton.handPacket.pos[1] * 100.0 / 8191.0
                    f2 = helium.handExoskeleton.handPacket.pos[2] * 100.0 / 8191.0
                    f3 = helium.handExoskeleton.handPacket.pos[3] * 100.0 / 8191.0

                    # print([thumb, f1, f2, f3])

                    r.cinematica_directa([thumb, f1, f2, f3])

                    # Para que vaya un poco mas lento
                    time.sleep(0.1)

            # Ya se ha llegado a la posicion abierto, asi que cierro
helium.close()
print("[Helium_robot] Finish!")
```