Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería en Tecnologías Industriales

Control de un robot Niryo desde Matlab y Simulink usando diferentes tipos de comunicaciones

Autor:

Beneyto Rodríguez, Alicia

Tutor(es):

Herreros López, Alberto

Departamento de Ingeniería de Sistemas y Automática

Valladolid, Junio 2022

Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

AGRADECIMIENTOS

Agradecer, principalmente, a mi tutor Alberto Herreros López por su ayuda, dedicación y por haberme dado la oportunidad de poder realizar este trabajo.

Mostrar también el agradecimiento a mi familia, destacando a mi madre, por su constante esfuerzo, paciencia y motivación para que haya llegado hasta aquí.

RESUMEN

Este proyecto tiene como finalidad realizar la comunicación con el robot colaborativo, Niryo Ned, a través de la herramienta de software matemático Matlab. Para conseguir este objetivo, se estudian y se describen las distintas aplicaciones que hacen posible la comunicación entre Matlab y el ROS (Robot Operating System) del robot.

Además de comentar todas las especificaciones técnicas del robot, se han creado distintas estaciones de trabajo con el robot viendo sus diferencias entre un modelo dinámico o uno cinemático.

Por último, se ha diseñado una aplicación en App Designer capaz de comunicarse con el robot Niryo Ned mediante la conexión de tipo TCP/IP. Este diseño tiene un fin educativo para que los alumnos demuestren sus conocimientos de robótica mediante el uso de este robot.

PALABRAS CLAVE

Matlab, App Designer, robótica, ROS, TCP/IP.

ABSTRACT

The project consists of communicating with the collaborative robot, Niryo Ned, using the mathematical software tool Matlab. To achieve this objective, the different applications that make communication between Matlab and the robot's ROS (Robot Operating System) possible are studied and described.

In addition to discussing all the technical specifications of the robot, different workstations have been created with the robot, showing the differences between a dynamic and a kinematic model.

Finally, an application has been designed in App Designer capable of communicating with the Niryo Ned robot via TCP/IP connection. This design has an educational purpose so that students can demonstrate their knowledge of robotics by using this robot.

KEY WORDS

Matlab, App Designer, robotic, ROS, TCP/IP.

ÍNDICE

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

CAPÍTULO 1. INTRODUCCIÓN Y OBJETIVOS

1.1. INTRODUCCIÓN

Los robots comenzaron a utilizarse en cadenas de montaje desempeñando tareas muy limitadas desde hace varias décadas. Eran máquinas de gran tamaño programadas por un humano, por lo que no eran conscientes de lo que les rodeaba. Por este motivo se solían instalar en recintos cerrados para mantener la seguridad de los que le rodean [1].

Desde hace varios años, está surgiendo una nueva generación de robots, denominados robots colaborativos, capaces de trabajar de forma coordinada con los humanos. Llevan incorporados sensores de fuerza para garantizar la seguridad a su alrededor [1].

Los robots colaborativos se caracterizan por ser ligeros, flexibles y fáciles de instalar, han sido diseñados para interactuar con los humanos compartiendo el mismo espacio de trabajo.

En cuanto al ámbito educativo, varías compañías han desarrollado diversos robots colaborativos con el fin de aprender robótica directamente con un robot similar a los industriales. De esta forma, permite al alumno crear sus propios programas y a la vez prototipar escenarios industriales donde se incorpore este robot.

A lo largo de los años se han desarrollado diversas herramientas de simulación en Matlab y Simulink, para recrear estaciones de trabajo donde se le ordene al robot unos movimientos concretos y así profundizar en el estudio de la robótica.

En este proyecto se ha querido dar un paso más estableciendo la comunicación con un robot colaborativo real desde Matlab. De esta forma se le proporcionan órdenes al robot desde una interfaz creada en esta aplicación. Para la creación de las estaciones, se dispone de una librería de Simulink denominada *Simscape*, con la que los alumnos podrán interactuar y gracias a la aplicación creada comprobarán el uso de esta librería en la realidad.

1.1.1. ESTRUCTURA DEL PROYECTO

El proyecto consta de seis capítulos, más dos apartados referidos a los anexos y a la bibliografía.

- **Capítulo 1. Introducción y objetivos**. Se hace una breve introducción sobre el desarrollo de los robots colaborativos y los objetivos principales y secundarios de este proyecto.
- **Capítulo 2. Estado del arte**. Se recopila toda la información necesaria como base para el desarrollo del proyecto, como son los fundamentos de robótica, los antecedentes, las librerías de Matlab y Simulink, y el ROS.
- **Capítulo 3. Niryo Ned.** Se exponen las especificaciones técnicas, las diferentes formas de comunicarse con este robot y la aplicación que incorpora, Niryo Studio.
- **Capítulo 4. Desarrollo del proyecto**. Se explican las diversas estaciones de trabajo que se han creado mediante *SimRob*, las aplicaciones necesarias para establecer la comunicación entre Matlab y ROS, y la aplicación creada en App Designer para Niryo Ned.
- **Capítulo 5. Resultados**. Se realiza un estudio dinámico de una estación, creada en el capítulo anterior, mediante diferentes ensayos y se verifica el uso de la aplicación creada para Niryo Ned mediante imágenes que muestran las respuestas a las instrucciones dadas.
- **Capítulo 6. Conclusiones y líneas futuras**. Se exponen las conclusiones obtenidas del proyecto y líneas de trabajo futuras para posteriores proyectos.
- **Anexos**. Incluye el código perteneciente a todos los componentes que forman la interfaz creada en App Designer, además de los archivos *pyModule.m, pyLines.m* y *Live_pyNiryo.* También se han añadido las abreviaturas que aparecen en el proyecto para facilitar su lectura.
- **Bibliografía**. Se presenta toda la documentación consultada para el desarrollo de este proyecto.

1.2. OBJETIVOS

Partiendo del robot colaborativo que ha integrado la escuela de Ingenierías Industriales de la Universidad de Valladolid, denominado Niryo Ned, se ha querido demostrar el uso didáctico de este robot, con el fin de utilizarse en diversas clases con los alumnos y probar su posible manipulación desde distintos programas a los habitualmente usados en el control de Niyo ned, como es Niryo Studio. La manipulación se realiza desde Matlab y para ello es necesario conocer todas las comunicaciones entre esta aplicación y el ROS del robot.

El desarrollo del proyecto tiene principalmente un fin didáctico en el área de la robótica y las distintas comunicaciones, además de los diversos ensayos realizados con este robot para su futuro uso en las aulas. Se ha incorporado una librería denominada *SimRob*, basada en Matlab y Simulink, para poder realizar las clases y se ha querido incluir el robot Niryo Ned en esta librería.

Para lograr este objetivo principal, se han marcado distintos objetivos intermedios, que son:

- Conocer los detalles técnicos de Niryo Ned y sus distintas formas de comunicación.

- Describir y justificar cada aplicación necesaria para conseguir la comunicación entre ROS y Matlab. Esto conlleva compaginar dos sistemas operativos distintos, como son Windows y Linux, en un mismo ordenador.

- Enumerar las instrucciones a seguir para una correcta instalación de cada programa necesario para la comunicación con el robot.

- Obtener las diferencias existentes entre una estación de trabajo dinámica y otra cinemática.

- Diseñar una aplicación mediante App Designer que controle tanto el robot simulado en la estación de Matlab, como el real.

- Analizar la influencia que supone incorporar un sistema de control y saturación de los actuadores sobre el modelo.

CAPÍTULO 2. ESTADO DEL ARTE

2.1. FUNDAMENTOS DE ROBÓTICA

2.1.1. DEFINICIÓN Y CONCEPTOS BÁSICOS

Un manipulador es un mecanismo formado generalmente por elementos en serie, articulados entre sí, destinado al agarre y desplazamiento de objetos. Es multifuncional y puede ser gobernado directamente por un operador humano o mediante un dispositivo lógico [2].

Teniendo en cuenta la descripción anterior, un robot se define como un manipulador multifuncional reprogramable con varios grados de libertad, capaz de manipular materias, piezas, herramientas o dispositivos especiales según trayectorias variables programadas para realizar diversas tareas. Está formado por una serie de eslabones unidos mediante articulaciones que permiten el movimiento relativo entre cada dos eslabones consecutivos, así se muestra en la siguiente figura [2].

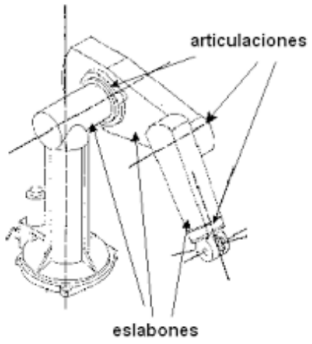


Ilustración 1. Estructura de un robot

Algunos conceptos básicos que se deben conocer de los robots se definen a continuación [2]:

- Eslabón: componente físico que existe entre articulaciones. Comúnmente se denomina brazo.
- Articulación: puntos que ponen en contacto los eslabones. Pueden ser de tipo prismático, de revolución o esféricas. Estos elementos están accionados por actuadores.
- Tool Center Point (TCP): punto central de la herramienta del robot, define el alcance del robot.

- Grado de libertad (GDL): indica cada uno de los movimientos independientes que puede realizar una articulación de un robot respecto a la anterior. Determina la accesibilidad de un robot y su capacidad para orientar sus herramientas, soliendo coincidir con el número de articulaciones del robot.

La robótica se ocupa del estudio de la programación y la mecánica de los robots, relacionándose con campos tan variados como la informática, la electrónica o la ingeniería. Con el paso de los años, esta ciencia ha permitido el desarrollo de distintas herramientas matemáticas para la descripción de un robot, como pueden ser el algoritmo de Denavit–Hartenberg o mediante los archivos URDF (United Robotics Description Format) que se describirán en el apartado *Definición matemática de un robot.*

2.1.2. CINEMÁTICA DEL ROBOT

La cinemática estudia el movimiento del robot con respecto a un sistema de referencia. Realiza una descripción analítica del movimiento espacial del robot en función del tiempo, y en particular las relaciones entre la posición y orientación del extremo final del robot con los valores que toman sus coordenadas articulares [2].

Existen dos tipos dependiendo de la forma en la que se resuelva el problema:

- Cinemática directa: consiste en determinar cuál es la posición y orientación del extremo del robot, con respecto a un sistema de referencia, conocidos los valores de las articulaciones y los parámetros geométricos de los elementos.
- Cinemática inversa: resuelve la configuración que debe adoptar el robot para una posición y orientación del extremo conocidas, es decir, determina el valor de cada articulación.

En la siguiente figura se muestra la relación entre los dos tipos de cinemática:

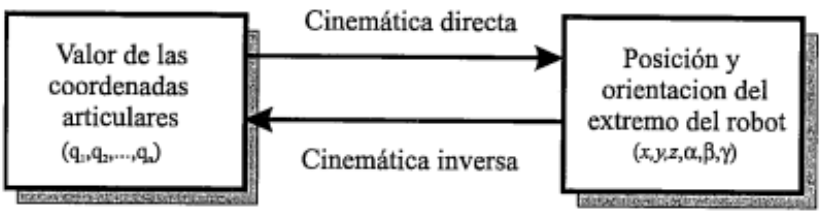


Ilustración 2. Diagrama de relación entre cinemática directa e inversa

En cuanto a la cinemática inversa, existen tres formas de realizar la trayectoria del extremo del robot desde la posición actual hasta la posición deseada:

- Trayectoria libre: el extremo del robot se desplaza a las coordenadas cartesianas deseadas por el camino más sencillo, es decir, no tiene por qué ser una línea recta.
- Trayectoria lineal: el extremo del robot se desplaza desde la posición actual hasta la deseada en línea recta.
- Trayectoria circular: realiza el movimiento generando un arco de semi-circunferencia desde la posición actual hasta la deseada pasando por un punto definido.

2.1.3. DINÁMICA DEL ROBOT

La cinemática estudia los tipos de movimientos sin atender a las causas que lo producen, mientras que la dinámica estudia las causas que originan los movimientos en los cuerpos [2].

El modelo dinámico de un robot pretende conocer la relación entre el movimiento del robot y las fuerzas implicadas en el mismo. Por lo tanto, debe relacionar matemáticamente [2]:

- La localización del robot definida por sus variables articulares o por las coordenadas de localización del TCP, y sus derivadas: velocidad y aceleración.
- Las fuerzas y pares aplicados en las articulaciones.
- Los parámetros dimensionales del robot, siendo estos la longitud, las masas y las inercias de cada elemento.

Es preciso señalar que en la mayor parte de aplicaciones reales de la robótica, las cargas e inercias manejadas no son suficientes para originar deformaciones en los eslabones del robot. Por lo tanto, la siguiente obtención del modelo dinámico considera a éste como un sólido rígido.

Este modelo se basa en el planteamiento del equilibrio de fuerzas establecido en la segunda ley de Newton, o su equivalente para movimientos de rotación, denominada ley de Euler [2], descrita por las ecuaciones (1) y (2):

$$\sum F = m\dot{v} \tag{1}$$

$$\sum T = I\dot{\omega} + \omega \times (I\omega) \tag{2}$$

En el caso más simple se tiene un robot monoarticular con toda la masa concentrada en el centro de gravedad donde no existe rozamiento y no se manipula ninguna carga. El equilibrio de fuerzas-pares daría como resultado la ecuación (3), [2]:

$$\tau = ML^2\ddot{\theta} + MgLcos\theta \tag{3}$$

Del planteamiento del equilibrio de fuerzas y pares del robot se obtienen dos modelos dinámicos:

- Modelo dinámico directo: expresa la evolución temporal de las coordenadas articulares del robot en función de las fuerzas y pares que intervienen.
- Modelo dinámico inverso: expresa las fuerzas y pares que intervienen en función de la evolución de las coordenadas articulares.

El planteamiento del equilibrio de fuerzas para un robot real que tenga seis grados de libertad es mucho más complicado, por lo tanto, se obtiene mediante la formulación de Lagrange-Euler.

La formulación de Lagrange adopta la siguiente forma [3]:

$$\tau = \mathrm{D}(\mathrm{q})\ddot{\mathrm{q}} + \mathrm{H}(\mathrm{q},\dot{\mathrm{q}})\mathrm{q} + \mathrm{g}(\mathrm{q}) \tag{4}$$

Donde *D* es la matriz de masa o inercia, *H* representa las fuerzas centrífugas y de Coriolis y *g* es el vector que incluye los términos gravitatorios.

La ecuación (5) de Lagrange-Euler que incluye tanto los movimientos rotacionales como lineales, es la siguiente [4]:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \tau_i \tag{5}$$

Siendo la función de Lagrange:

$$L = K(\mathrm{q},\dot{\mathrm{q}}) - P(\mathrm{q}) \tag{6}$$

Donde *K* representa la energía cinética y *P* la energía potencial del sistema.

Se debe considerar que la mayoría de las simulaciones se realizan mediante un modelo cinemático debido a la complejidad de todos los factores que intervienen en el modelo dinámico, como se acaba de explicar. De esta forma, se obtiene una simulación con una mejor estética.

2.1.3.1. CONTROL DINÁMICO

El control dinámico tiene por misión procurar que las trayectorias realmente seguidas por el robot sean lo más parecidas posibles a las propuestas por el control cinemático [2].

El sistema de control es el encargado de la realización de un movimiento concreto por parte del robot, proporcionando a los actuadores las ordenes necesarias para conseguir dicho movimiento [2].

Este sistema de control incluye reguladores. Existen distintos tipos con diferentes combinaciones de las siguientes acciones:

- Acción proporcional (P): su señal de salida es proporcional a la entrada.

$$u(t) = k_p e(t) \tag{7}$$

  Donde k_p es la constante de proporcionalidad.

- Acción derivativa (D): su señal de salida es proporcional a la derivada de la señal del error.

$$u(t) = k_d \frac{de(t)}{dt} \tag{8}$$

- Acción integral (I): su señal de salida es proporcional a la integral del error.

$$u(t) = k_i \int_0^t e(t)dt \tag{9}$$

Combinado estas acciones se obtienen los reguladores: P, PI, PD y PID. En la siguiente figura se muestra un sistema de control con un regulador PID:

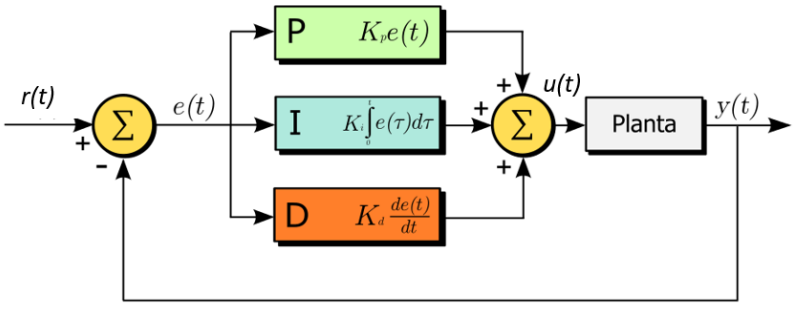


Ilustración 3. Regulador PID

Para simular el comportamiento dinámico de un robot se suele emplear un sistema en lazo cerrado con un controlador, con el objetivo de conseguir una

señal de salida y(t) igual a la de referencia r(t) corrigiendo la señal de salida gracias a la disminución del error e(t) por medio de la realimentación.

Existen diversas arquitecturas de la estructura de control con distintos objetivos cada una. La estructura de control básica tiene la siguiente forma:

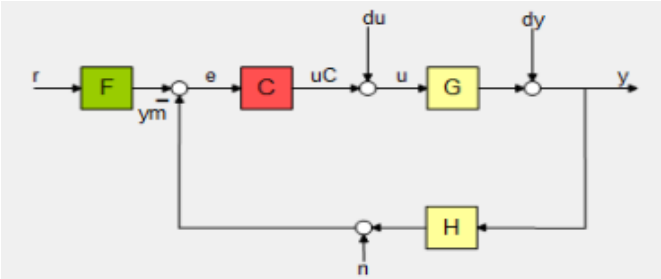


Ilustración 4. Estructura de control básica

Se compone de un filtro (F) por el que pasa la señal de referencia para eliminar el ruido que contenga, un controlador (C) necesario para reducir el error estacionario, la planta (G) y un lazo que permite realizar la realimentación de la señal de salida. Este lazo contiene otro filtro (H), siendo normalmente su valor 1.

También existen otras estructuras de control con una arquitectura de mayor complejidad, formada por varios lazos, como se muestra a continuación:

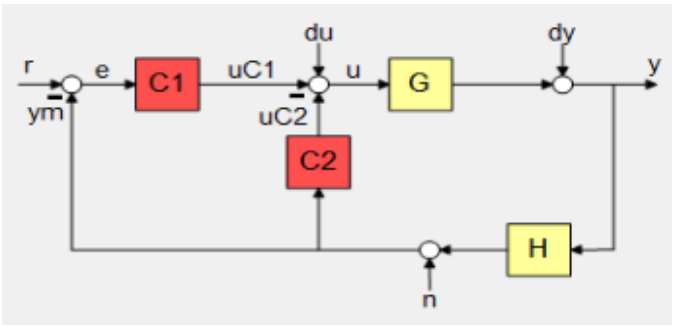


Ilustración 5. Estructura de control con varios lazos

En este caso, se ha incluido el controlador C2 en un lazo interno con el fin de conseguir la estabilidad del sistema. Se suele incorporar en sistemas inestables o que tengan una elevada oscilación. Al igual que en la estructura básica, el controlador C1 disminuye el error estacionario. Lo más común es que el controlador C1 sea de tipo proporcional o PI, y que C2 sea derivativo.

2.1.4. DEFINICIÓN MATEMÁTICA DE UN ROBOT

2.1.4.1. ALGORITMO DE DENAVIT-HARTENBERG

En 1955, Denavit y Hartenberg propusieron un método matricial que permite establecer de manera sistemática un sistema de coordenadas $\{S_i\}$ ligado a cada eslabón *i* de una cadena articulada, pudiéndose determinar seguidamente las ecuaciones cinemáticas de la cadena completa. Este método utiliza una matriz de transformación homogénea, que se obtendrá a continuación, para describir la relación espacial entre dos elementos rígidos adyacentes, reduciéndose el problema cinemático directo a encontrar una sola matriz que relacione la localización espacial del extremo del robot con su base [2].

Según esta representación, escogiéndose adecuadamente los sistemas de coordenadas asociados a cada eslabón, será posible pasar de uno al siguiente mediante 4 trasformaciones básicas que dependen exclusivamente de las características geométricas [2].

El conjunto de transformaciones que permiten relacionar el sistema de referencia del elemento *i* con el sistema del elemento *i-1* es el siguiente:

1. Rotación alrededor del eje z_{i-1} un ángulo θ_i.
2. Traslación a lo largo de z_{i-1} una distancia d_i.
3. Traslación a lo largo de x_i una distancia a_i.
4. Rotación alrededor del eje x_i un ángulo α_i.

Las transformaciones se deben realizar en el orden indicado ya que el producto de matrices no es conmutativo. De esta forma se tiene [2]:

$$A_i = T(z,\theta_i)T(0,0,d_i)T(a_i,0,0)T(x,\alpha_i) \tag{10}$$

Y realizando el producto entre matrices, se obtiene el resultado presentado en la ecuación (11):

$$A_i^{i-1} = \begin{bmatrix} C\theta_i & -C\alpha_i S\theta_i & S\alpha_i S\theta_i & a_i C\theta_i \\ S\theta_i & C\alpha_i C\theta_i & -S\alpha_i C\theta_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

Donde θ_i, a_i, d_i, α_i son los parámetros de D-H.

En la siguiente figura se puede ver un ejemplo gráfico donde se muestran estos parámetros:

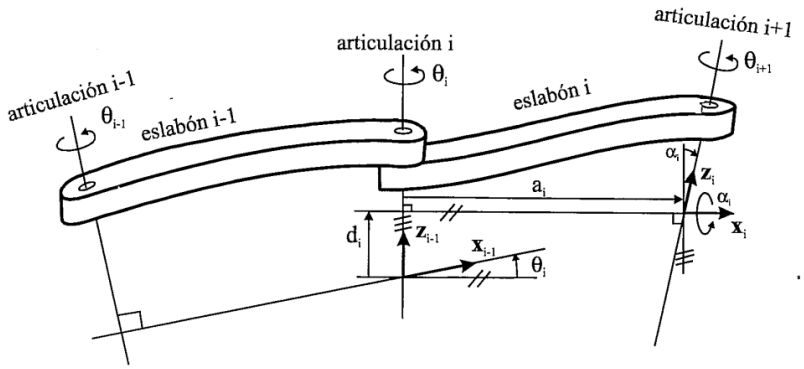


Ilustración 6. Parámetros de D-H

Estos parámetros dependen únicamente de las características geométricas de cada eslabón y de las articulaciones que le unen con el anterior y el siguiente, siendo estos [2]:

- θ_i Ángulo que forman los ejes x_{i-1} y x_i en un plano perpendicular al eje z_{i-1}. Es un parámetro variable en articulaciones giratorias.
- d_i Distancia a lo largo del eje z_{i-1} desde el origen del sistema de coordenadas (i-1)-ésimo hasta la intersección del eje z_{i-1} con el eje x_i. Es un parámetro variable en articulaciones prismáticas.
- a_i Distancia a lo largo el eje x_i que va desde la intersección del eje z_{i-1} con el eje x_i hasta el origen del sistema i-ésimo, en el caso de articulaciones giratorias. Si son articulaciones prismáticas, se calcula como la distancia más corta entre los ejes z_{i-1} y z_i.
- α_i Ángulo de separación del eje z_{i-1} y el eje z_i medido en un plano perpendicular al eje x_i.

2.1.4.1. ARCHIVO URDF (United Robotics Description Format)

Hasta hace poco tiempo, era muy común el uso del algoritmo de Denavit-Hartenberg pero poco a poco ha ido aumentando el uso de los archivos URDF como alternativa a este algoritmo ya descrito. Actualmente, los fabricantes de robots diseñan estos archivos URDF para poder realizar su descarga y producir la simulación.

Este archivo consiste en una descripción del robot en formato *xml*. Es ampliamente usado en los sistemas ROS, modelando el robot a través de URDF y realizando una simulación y un análisis mediante ROS. Una gran ventaja para

este proyecto es que se puede obtener un modelo *Simscape Multibody* importando un archivo URDF, de esta forma se puede realizar el análisis de simulación o diseño del controlador en Simulink. Este formato permite definir tanto la descripción cinemática de un robot como la dinámica, su representación visual y su modelo de colisiones [3].

Los archivos permiten describir a los robots como estructuras de árbol, figura 7, mediante elementos de eslabones (*links*) y de uniones (*joints*), que conectan los eslabones juntos y definen sus características cinemáticas y dinámicas de la articulación, limitando así los posibles giros o movimientos. Los robots se deben componer de estructuras rígidas conectadas por articulaciones.

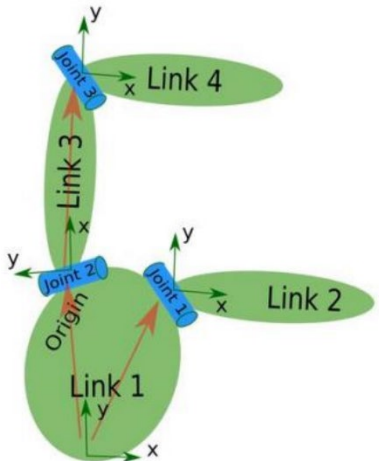


Ilustración 7. Estructura URDF

La descripción más común en este formato es:

```
<robot name= “ejemplo”>

   <link> … </link>
   <link> … </link>
   <joint> … </joint>

</robot>
```

Siendo el elemento raíz <robot>. A continuación se describen las propiedades de los elementos *link* y *joint*.

- Elemento *Link*

Los eslabones del robot se definen mediante el nombre <*link name*>. Cuando se utiliza una determinada estructura como sistema de coordenadas base el nombre que recibe es *base-link*.

A continuación se muestra una figura de la estructura de este elemento:

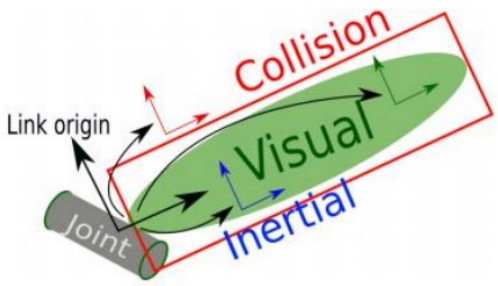


Ilustración 8. Estructura del elemento link

El código que presenta, necesario para su descripción cinemática y dinámica, se divide en los siguientes bloques que tienen una serie de atributos cada uno:

- ***<inertial>:*** Define las propiedades inerciales del eslabón.
    - ***<origin>:*** define el origen del elemento inercial con respecto al origen del link.
    - ***<mass>:*** determina la masa del eslabón.
    - ***<inertia>:*** es la matriz de rotación inercial 3x3.

- ***<visual>:*** Describe la parte visual del eslabón.
    - ***<origin>:*** especifica el origen de coordenadas del elemento visual con respecto al origen del *link*. El desplazamiento viene representado por x,y,z y los ángulos de giro por r,p,y.
    - ***<geometry>:*** especifica la forma visual de la estructura.
    - ***<material>:*** define un nombre del material del elemento visual.
    - ***<color rgba>:*** define el valor del color del material mediante cuatro números que representan el rojo, verde, azul y opaco, en un rango entre 0 y 1.

- ***<collision>:*** Especifica las propiedades de colisión del eslabón. Generalmente es más grande que el elemento de visión de tal manera que detecta que el link ha colisionado antes de que suceda realmente.

  También se pueden definir propiedades como *<origin>, <geometry>, <material>* o *<color>.*

Se muestra un ejemplo del código y su estructura:

```
 <link name="shoulder_link">

     <inertial>
       <origin rpy="0 0 0" xyz="-0.0307310771918448
0.00581074732334248 0.0302277607274943"/>
       <mass value="0.474558968160783"/>
       <inertia ixx="0.0015" ixy="0.0" ixz="0.0" iyy="0.0015"
iyz="0.0" izz="0.0015"/>
     </inertial>

     <visual>
       <origin rpy="0 0 0" xyz="0 0 0"/>
       <geometry>
         <mesh
filename="package://niryo_robot_description/meshes/ned/collada/shoul
der_link.dae"/>
       </geometry>
     </visual>

     <collision>
       <origin rpy="0 0 0" xyz="0 0 0"/>
       <geometry>
         <mesh
filename="package://niryo_robot_description/meshes/ned/stl/shoulder_
link.stl"/>
       </geometry>
     </collision>

   </link>
```

- Elemento *Joint*

Este elemento de unión necesita una descripción de sus propiedades dinámicas y cinemáticas que se plasmará en el código. A continuación se muestra su estructura:

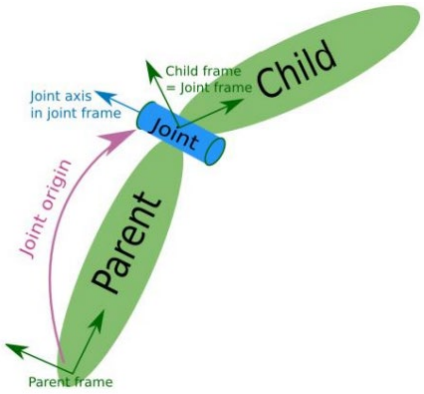


Ilustración 9. Estructura del elemento Joint

La estructura de este elemento puede contener los siguientes atributos:

- ***<parent link>:*** define el eslabón padre, es decir, la estructura principal conectada por esta unión.
- ***<child link>:*** define el eslabón hijo, es decir, la estructura secundaria conectada por esta unión.
- ***<origin>:*** define el origen de coordenadas del elemento *joint* con respecto al origen de coordenadas del link padre.
- ***<axis>:*** establece la dirección del movimiento de la articulación. Por defecto, el valor predeterminado es el eje x (1, 0, 0).
- ***<dynamics>:*** especifica las propiedades físicas de la articulación: *damping* indica el valor de amortiguación física de la articulación y *friction* el valor de fricción estática física de la articulación.
- ***<limit>:*** define los límites de movimiento de la articulación. Se puede definir únicamente para articulaciones prismáticas y de revolución.

Se muestra un ejemplo de la estructura del código:

```
 <joint name="joint_2" type="revolute">

     <origin rpy="1.5707963268 -1.5707963268 0" xyz="0 0 0.065"/>
     <parent link="shoulder_link"/>
     <child link="arm_link"/>
     <axis xyz="0 0 1"/>
     <limit effort="10.0" lower="-2.09003177926"
upper="0.610167106497" velocity="2.2"/>

   </joint>
```

2.2. ANTECEDENTES

San José Miguel Ángel Mato [5] llevó a cabo su proyecto final de carrera en 2014, en el cual desarrolló una interfaz gráfica desde Matlab para controlar el robot ABB IRB 120 simulado. Lo realizó empleando la librería *Robotic ToolBox*. Además consiguió comunicarse con el robot real mediante la comunicación OPC.

Posteriormente, en 2015, Juan Antonio Ávila Herrero [6] basó su proyecto en el modelado de la célula robótica utilizando el programa *RobotStudio*. Seguidamente, en 2016, Álvaro Galindo de los Santos [7], teniendo como base el proyecto anterior, se centró en el control remoto del robot mediante tecnología WIFI. Ambos trabajaron con fines docentes dando lugar a la estación robótica actual con la que cuenta la escuela.

En 2019, Carlos Jiménez Jiménez [8] desarrolló su trabajo fin de master con el objetivo de crear un sistema robótico educativo capaz de jugar al ajedrez, empleando la comunicación TCP/IP entre el robot y Matlab. Más tarde en 2020, Víctor Lobo Granado [9] realizó su trabajo fin de master en el que planteaba el diseño, la simulación y la fabricación de entornos de trabajo para una estación de trabajo, y en el cual conseguía una comunicación entre el robot y un entorno gráfico permitiendo así la simulación de la estación con Simulink.

Por último, Sandra Arévalo Fernández [3] llevó a cada su trabajo de fin de grado en 2021, que consistía principalmente en crear una librería de iconos en Simulink para facilitar el diseño de una estación robótica a los alumnos. Además, realizó la conexión entre el robot real ABB y Matlab mediante la comunicación OPC. Junto al profesor Alberto Herreros López, publicaron un artículo [10] explicando la herramienta *SimRob,* de uso didáctico, que habían creado para diseñar y simular estaciones robóticas.

Por otra parte, diversos profesores del Departamento de Automática de la escuela de Ingenierías Industriales han llevado a cabo varios proyectos educativos sobre esta plataforma. El profesor y tutor de este proyecto, Alberto Herreros López dotó al robot la capacidad de escribir sobre un papel el texto leído de un fichero empleando distintos tipos de letra y de planos de escritura.

2.3. MATLAB Y SIMULINK

Matlab y Simulink son dos marcos de cálculo numérico y visualización de datos pertenecientes a la compañía MathWorks.

Matlab (Matrix Laboratory) es una plataforma de programación y cálculo numérico utilizada por millones de ingenieros y científicos para analizar datos,

desarrollar algoritmos y crear modelos. Ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio, lenguaje M. Sus prestaciones básicas son la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware [11].

Ilustración 10. Icono de Matlab

Esta plataforma dispone de varias herramientas adicionales que expanden sus prestaciones, denominadas *toolboxes*, siendo una de estas Simulink.

Simulink es un entorno de programación visual mediante diagramas de bloque que se utiliza para diseñar sistemas con modelos multidominio, simular antes de implementar en hardware y desplegar sin necesidad de escribir código. Este entorno recoge la programación de más alto nivel de abstracción. Mientras que Matlab emplea la extensión *.m*, Simulink genera archivos con extensión *.mdl* [12].

Cuando se utilizan Matlab y Simulink conjuntamente, se está combinando la programación textual y la gráfica para diseñar un entorno de simulación, esto proporciona una gran ventaja como herramienta para el desarrollo de aplicaciones robóticas.

Durante los últimos años se han desarrollado *toolboxes* destinadas para el ámbito de la robótica, como es *Robotic Toolbox* de Matlab.

2.3.1. *Robotic System Toolbox* DE MATLAB

Coker desarrolló la aplicación *Robotic Toolbox* [13] en Matlab, con el objetivo de crear un entorno universal para el análisis y el control de robots basado en la parametrización de Denavit-Hartenberg. Esta parametrización se encarga de definir los ejes del robot mediante algoritmos de optimización local obteniendo la cinemática inversa. Además, para acelerar las simulaciones, emplea una representación basada en los ejes del robot, un diagrama de hilos.

A continuación se muestra una figura de la estructura robótica definida mediante *Robotic Toolbox:*

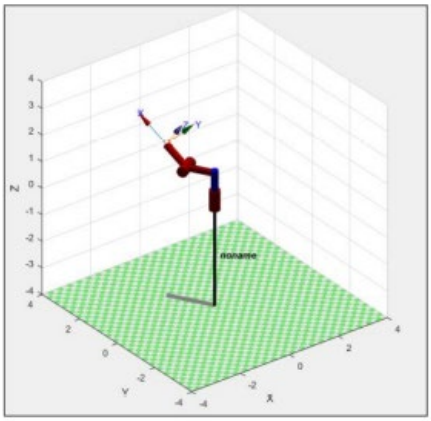


Ilustración 11. Estructura robótica definida mediante Robotic Toolbox

La herramienta más reciente de Matlab que se emplea para la robótica es *Robotic System Toolbox* [14], incorporada en las versiones posteriores a la 2015a. Se encarga de guardar la configuración de un robot, mediante el objeto *RigidBody*, en formato Denavit-Hartenberg (DH) o leyendo un fichero URDF. La simulación de este objeto se realiza a través de hilos o con ayuda de ficheros gráficos *stl* en Matlab. Esta librería proporciona herramientas y algoritmos que permiten el estudio de la cinemática y dinámica del robot, pudiendo comprobar las colisiones y la generación de trayectorias. El inconveniente es que las gráficas de Matlab no están preparadas para simulaciones tan pesadas y provocan una mayor lentitud.

Esta herramienta incluye una biblioteca de modelos de robots industriales disponibles comercialmente que pueden ser importados, visualizados y simulados gracias a la posibilidad de leer los archivos URDF.

A continuación se muestra una imagen con la estructura del robot *RigidBody*:

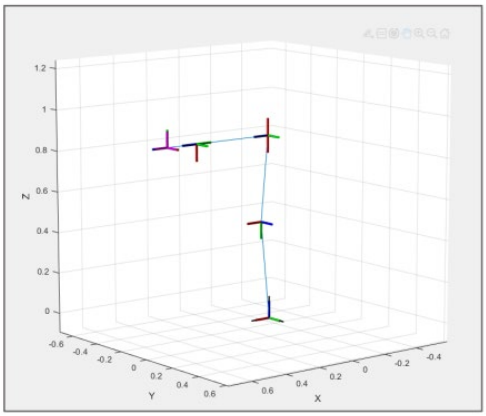


Ilustración 12. Estructura de un robot RigidBody

2.3.2. *Simscape Multibody* DE SIMULINK

Simscape Multibody [15] es una herramienta de simulación de mecanismos mecánicos de Simulink. Puede leer ficheros URDF y traducir la información de éstos en iconos de inercia y gráficos para cada brazo e iconos de revolución o traslación para los ejes rotacional o prismático. Un fichero de Simulink que contenga estos elementos puede ser leído como un objeto *RigidBody,* objeto de la librería *Robotic System Toolbox*. De esta forma se podrán controlar los movimientos del sistema, es decir, la cinemática inversa.

Con toda la información dada en estos ficheros, la aplicación genera una animación 3D, representada con la herramienta *Mechanics Explorer*, aplicación de Matlab utilizada para la simulación, visualizando así la dinámica del sistema. El resultado de la simulación se muestra en la siguiente figura:

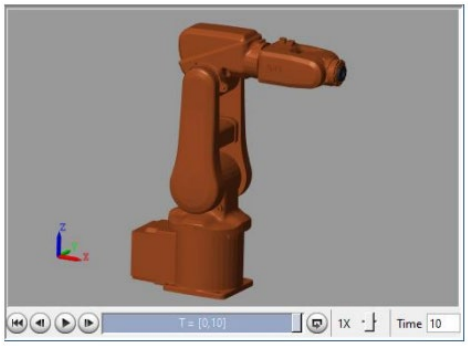


Ilustración 13. Animación de un robot obtenido con Simscape Multibody

A continuación se van a describir los iconos de *Simscape* utilizados para la simulación de los robots:

Tabla 1. Iconos empleados de Simscape Mutibody

Icono	Descripción
 World Frame	Representa el marco de referencia global en un modelo. Este marco es inercial y está en reposo absoluto. Los ejes son ortogonales y están dispuestos según la regla de la mano derecha.
 Mechanism Configuration	Proporciona parámetros mecánicos y de simulación a un mecanismo, es decir, un grupo autónomo de bloques *Simscape Multibody* interconectados. Los parámetros incluyen la gravedad y un delta de linealización para calcular las derivadas parciales numéricas durante la linealización.

 Solver Configuration	Cada red física representada por un diagrama de bloques de *Simscape* conectado requiere información de configuración del *solver* para la simulación. El modelo necesita esta información para comenzar la simulación.
 Infinite Plane	Exporta un plano infinito para modelar problemas de contacto. Puede modelar el contacto entre el plano y muchos tipos de geometrías, como todos los tipos de sólidos y cascos convexos de la biblioteca de elementos de cuerpo y la geometría de puntos de la biblioteca de curvas y superficies.
 Spline	Crea una curva *spline* continua basada en la interpolación cúbica entre los puntos especificados. La curva puede ser bidimensional, como un perfil de leva plano, o tridimensional, como la pista de una montaña rusa.
 Reference Frame	Este bloque representa un marco de referencia con respecto al cual se pueden definir otros marcos. El marco de referencia es generalmente no inercial. Puede acelerar con respecto al marco mundial. Este bloque es opcional en un modelo.
 Connection Port	Transfiere una conexión física o una señal a través de los límites del subsistema.
 Rigid Transform	Aplica una transformación invariable en el tiempo entre dos fotogramas. La transformación gira y traslada el marco del puerto seguidor (F) con respecto al marco del puerto base (B).
 Weld Joint	Representa una articulación con cero grados de libertad. Los marcos de la base y del seguidor, cada uno conectado a un cuerpo rígido separado, son coincidentes todo el tiempo. El cuadro de diálogo del bloque ofrece opciones de detección de fuerzas y pares totales y de restricción.
 Revolute Joint	Modela una articulación que tiene un grado de libertad rotacional.
 Inertia	Añade a la estructura adjunta una masa puntual o distribuida con propiedades inerciales fijas.

Modela un elemento sólido con geometría, inercia, color y marco de referencia derivados de un archivo externo.

Estos iconos se utilizarán para la creación de las estaciones robóticas y la de sus propios componentes, así se verá en el siguiente apartado.

2.3.3. LIBRERÍA *SimRob*

Con el objetivo de simplificar todas las herramientas explicadas anteriormente y que tenga un uso más sencillo, se ha propuesto una aplicación denominada *SimRob* [10] para definir una estación robótica de forma sencilla mediante *SimScape Multibody* utilizando iconos definidos de robots y de herramientas. Puede realizar modelos tanto dinámicos como cinemáticos. Para la cinemática directa e inversa, la estación se convierte en un objeto *RigidBody*. Finalmente, se produce la simulación en el entorno *Mechanics Explores.*

Con el objetivo explicado anteriormente, se han diseñado iconos de robots, herramientas y piezas para poder construir una estación de trabajo.

2.3.3.1. ICONOS DE ROBOTS

Se ha puesto una imagen de los robots en la máscara de los iconos para que su uso sea más sencillo. En la figura 14 aparecen los iconos de robots en el lenguaje de *Simscape Multibody*.

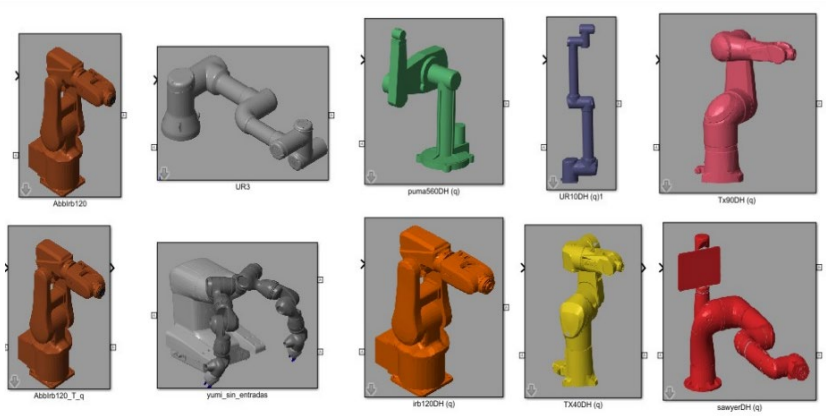


Ilustración 14. Iconos de robots

Se dispone de iconos que definen las articulaciones y además, de iconos que definen componentes gráficos y dinámicos de cada brazo del robot.

Se puede observar la forma que tiene la estructura interna de un robot en la figura 15, y la de un brazo del robot en la figura 16.

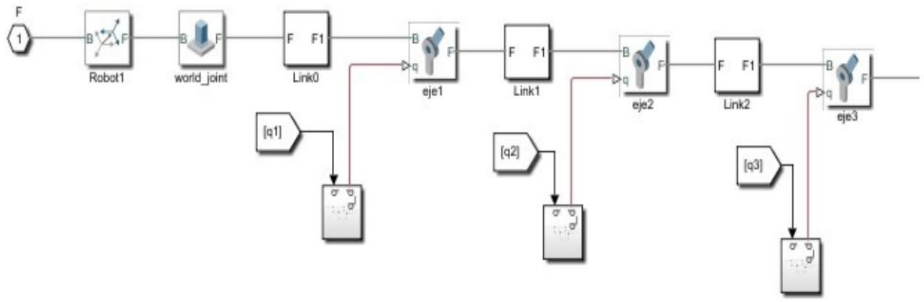


Ilustración 15.Estructura interna de un robot

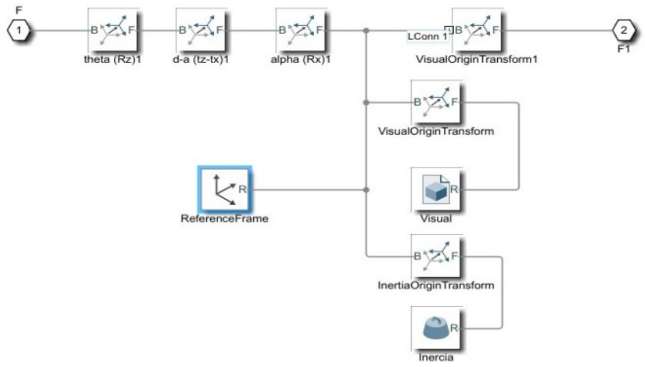


Ilustración 16. Estructura interna de un brazo del robot.

Un método para obtener los iconos es importando un fichero URDF del robot que contiene toda la información de éste.

El otro método consiste en modelar un robot estándar con el formato Denavit-Hartenberg (DH) y cambiar sus parámetros y los ficheros *stl* de las gráficas. En la figura 17 se observa la máscara de estos iconos donde se define la matriz DH y las propiedades dinámicas del robot.

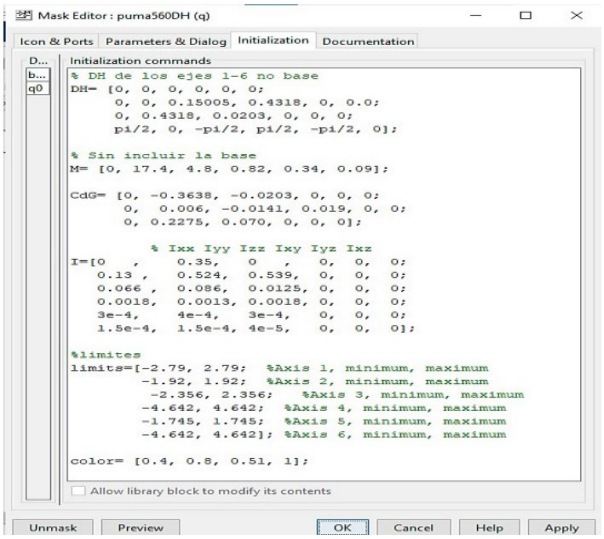


Ilustración 17. Máscara de un robot con el formato DH

Los robots tienen la posición de cada eje como la entrada de Simulink, y una entrada y salida de *Simscape* para el acople del icono a la estación.

Se deben definir dos variables en cada icono: la posición de la base del robot y la posición inicial de los ejes del robot.

2.3.3.2. ICONOS DE PIEZAS Y HERRAMIENTAS

Las características principales del icono deben quedar definidas en una variable tipo estructura que se encuentra en la máscara del icono. Para que se definan por defecto de una forma más sencilla, se dispone de la función *Piezas*.

Estas características son: la posición de la base, la posición del TCP, el color y las características dinámicas.

En las siguientes ilustraciones, 18 y 19, aparecen algunos iconos de piezas y estructuras que han sido creados.

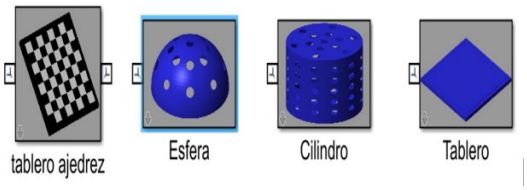


Ilustración 18. Iconos de piezas

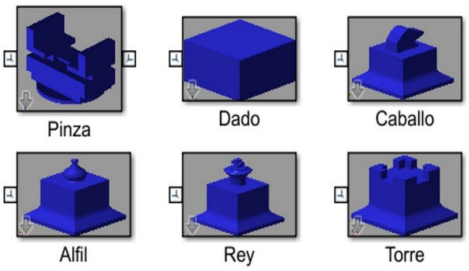


Ilustración 19. Icono de una herramienta y cargas

2.3.3.3. COMPOSICIÓN DE UNA ESTACIÓN CON UN ROBOT

La estación debe contener unos iconos bases de *Simscape*: referencia total, propiedades de gravedad y el *solver* de las ecuaciones diferenciales.

En la figura 20 se ha diseñado una estación con el robot ABB irb120 que tiene como herramienta una pinza y como carga un lápiz. La entrada de Simulink son los ejes del robot y las salidas son las conexiones de *Simscape*.

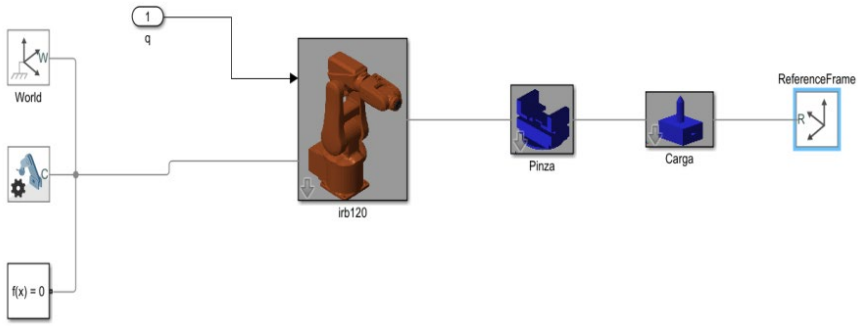


Ilustración 20. Estación con el robot ABB irb120

En la figura 21 se muestra el robot simulado con la pinza y la carga sujetada por ésta. El robot se ha girado 30 grados en el eje 1 y 90 grados en el eje 5 y solo se ha simulado los cambios en los ejes.

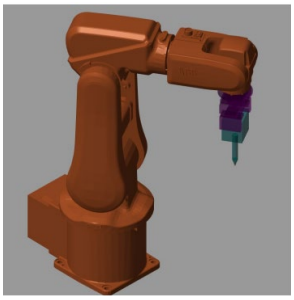


Ilustración 21. Simulación de la estación con el robot ABB irb120

2.3.4. OBJETO *Kin*

El proyecto se ha basado en el uso del objeto *Kin*. Esta herramienta ha sido creada por Alberto Herreros López con el objetivo de realizar un simulador de robot relacionando *MultiBody* de Simulink y *RigidBody* de Matlab.

Los métodos principales de los que consta la clase *Kin* son los movimientos típicos de cualquier lenguaje de programación de un robot: movimiento en arco, en línea y por coordenadas tanto articulares como cartesianas.

Esta clase parte de un modelo de robot definido en *MultiBody* y su equivalente en *RigidBody* que se define a partir del primero. Se usa *MultiBody* para la simulación de movimientos debido a que es más continua. En este entorno se pueden añadir las herramientas y los objetos de trabajo que se deseen, además de grabar las simulaciones para su posterior reproducción y definir la velocidad de simulación. Para poder obtener las cinemáticas inversas y directas, es necesario el modelo equivalente de *RigidBody* que puede constar solo del robot, ya que no se va a visualizar. Otra forma de obtener estos puntos es con la interface creada en App Designer.

Introduciendo el objeto *RigidBody* y el fichero Simulink de *MultiBody* como argumentos, las salidas por pantalla se convierten en simulaciones a la velocidad deseada.

2.3.4.1. FUNCIONES

Métodos de Rigid-Body

- **Tool():** realiza cambios de herramienta, o entre carga y herramienta.
- **Wobj(), Body(), DelBody():** permiten incluir o quitar cuerpos en el robot *RigidBody*. Todos los cuerpos se definen con *MultiBody* por lo que no será útil usarlos conjuntamente con esta aplicación.

Métodos de visualización

- **Show([joint], [pose]):** el argumento *joint* muestra el robot con la posición articular dada y cambia la posición interna del robot. El argumento *pose* permite dibujar los puntos dados interpolados en Simulink como un rastreo de la posición del TCP, usado principalmente en los métodos de movimiento. Se puede mostrar el robot en la posición actual cuando no se introducen argumentos.

- **Rec(value):** graba todos lo movimiento y tiempos de simulación para la futura reproducción. Entre intervalos de grabación tiene un tiempo de espera de 1 segundo. Cuando el valor del argumento es 1, empieza a grabar respetando los movimientos en memoria, mientras que para dejar de grabar, el valor debe ser 0. Si no se introduce ningún argumento, la información grabada se resetea y comienza a grabar.

- **Rep():** reproduce lo grabado hasta el momento.

Métodos de movimientos básicos

- **MoveAbsJ(Joint, [nPts]):** permite el movimiento introduciendo coordenadas articulares, es decir, el robot se mueve desde la posición inicial hasta una o varias posiciones articulares deseadas.
  El primer argumento, *Joint*, es un matriz (N x 6) en la cual cada fila corresponde a una posición articular dada en radianes.
  nPts define los puntos de interpolación que se desean entra la posición actual y la dada siempre que *pose* sea solo un punto. Por defecto se tienen 10 puntos.

- **MoveJ(pose, [Hwobj], [nPts]):** realiza el movimiento tras introducir unas coordenadas cartesianas. Desplaza la herramienta del robot desde la posición inicial hasta las posiciones cartesianas *pose* por el camino más sencillo, es decir, no tiene por qué ser una línea recta. Por defecto se tienen 10 puntos.

  Pose es una matriz (N x 6) donde cada fila tiene la forma $[x,y,z,r_z,r_y,r_x]$. Si la entrada es [x, y, z], se entiende que la orientación se mantiene igual que la inicial. La posición se da en milímetros y la orientación en radianes.

  Hwobj consiste en una matriz homogénea que defina el eje de referencia deseado de los puntos.

- **MoveL(pose, [Hwobj], [nPts]):** permite realizar un movimiento en línea recta desde la posición actual hasta la posición pose tipo $[x,y,z,r_z,r_y,r_x]$ interpolando el número de puntos introducidos en el tercer argumento. Si la entrada de *pose* es [x, y, z], se entiende que la orientación se mantiene constante. Si se incorporan los ejes de referencia mediante el segundo argumento, el punto *pose* queda definido con respecto a ellos.

- **MoveC(pos1, pos2, [Hwobj], [nPts]):** permite la realización de movimientos generando un arco de semi-circunferencia desde la posición actual hasta *pos2* pasando por *pos1*. Estos dos primeros argumentos son del tipo [x, y, z]. El ángulo es perpendicular al plano definido por los tres puntos.

  Hwobj es la matriz homogénea del objeto, por defecto el eje global.

  nPts son los puntos deseados para la interpolación, por defecto son 10 puntos.

Métodos internos

- **Pose(joint, [Hwobj]):** este método emplea la cinemática directa, tiene como entrada *joint* las posiciones articulares N x 6 y devuelve las posiciones cartesianas tipo $[x,y,z,r_z,r_y,r_x]$ correspondiente.

  Hwobj es la matriz homogénea de los ejes de referencia deseados. Por defecto, el eje es la base del robot.

  Sin argumentos devuelve la posición actual de la herramienta del robot.

- **Joint(pose, [w]):** este método emplea la cinemática inversa punto a punto, tiene como entrada las posiciones cartesianas tipo $[x,y,z,r_z,r_y,r_x]$ y devuelve las posiciones articulares N x 6.

  El vector w = $[w_{Rz},w_{Ry},w_{Rx},w_x,w_y,w_z]$ es el peso en la optimización local de las rotaciones y traslaciones.

- **Robot():** devuelve una propiedad del robot que se haya podido modificar.

- **Sim(mdl, type, value):** envía a Simulink los parámetros requeridos para la simulación.

  mdl es el fichero de Simulink donde se encuentra el sistema.

  Value es el valor de la velocidad o del tiempo.

  Type es el tipo de medida elegido, puede ser 'T' de tiempo o 'V' de velocidad.

Para finalizar este apartado que engloba todos los conceptos básicos necesarios para conocer Matlab y Simulink, se ha creado un esquema para facilitar la comprensión de la conexión que existe entre todas las librerías comentadas anteriormente:

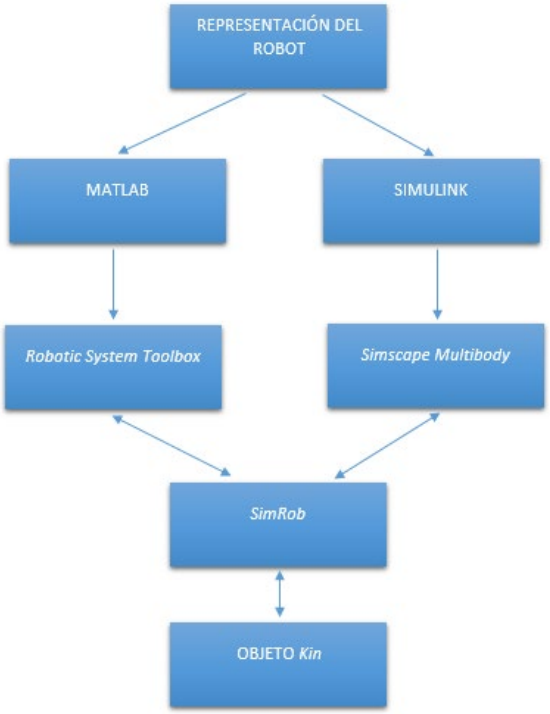


Ilustración 22. Conexión entre las librerías de Matlab y Simulink

2.3.5. APP DESIGNER

App Designer es un entorno de desarrollo interactivo con el fin de diseñar una aplicación y poder programar su comportamiento. La creación de la aplicación consta de dos partes, por un lado, el código, encargado de ejecutar todas las sentencias y hacer que funcione la aplicación, y por otro lado se encuentra la parte visual, encargada del aspecto visual [16].

El creador tiene una gran libertad a la hora de crear el aspecto visual de la aplicación ya que tiene a su disposición un lienzo donde colocar todos los elementos que desee y configurarlos a su gusto.

Una gran ventaja de este software es la comprobación automática de los errores del código gracias el uso de *Code Analyzer*, el cual muestra mensajes de advertencia y errores mientras se está escribiendo el código.

Otra ventaja es la gran variedad de componentes que ofrece, ya sean botones, listas desplegables, áreas de texto, etc, los cuales poseen las llamadas

funciones *Callback*. Estas funciones son ejecutadas cuando se interactúa con los componentes que se encuentran en el área de diseño, y se encargan de recoger la información que el usuario introduce en ellas, la cual se empleará en ocasiones en la ejecución de otras funciones. Además, cambiando los valores de las propiedades de cada elemento, se puede modificar su apariencia y comportamiento [16].

Los componentes se dividen en varios grupos: comunes, contenedores, herramientas de figura e instrumentación. En este proyecto solo se van a usar algunos componentes del primer grupo mencionado. A continuación se pueden observar los componentes comunes:

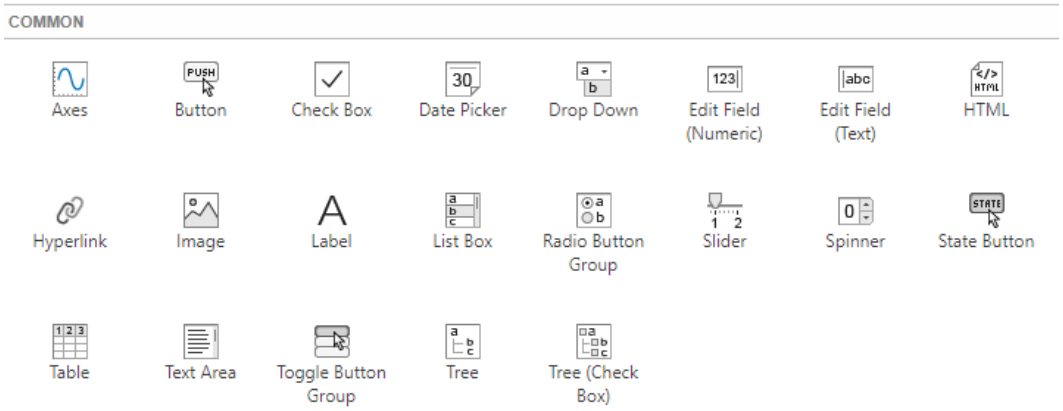


Ilustración 23. Componentes comunes de App Designer

Se van a explicar las propiedades que caracterizan a algunos componentes de la imagen anterior, los cuales se han usado en este proyecto, en la siguiente tabla:

Tabla 2. Iconos empleados en App Designer

Icono	Descripción
Button	Responden cuando el usuario pulsa y suelta.
Edit Field (Text)	Permiten introducir texto.
Label	Contienen texto estático para etiquetar partes de una aplicación.

	Permiten al usuario seleccionar un valor a lo largo de la cremallera.
	Gestionan la selección de botones de radio o de conmutación.

2.4. ROS (Robot Operating System)

A lo largo de las últimas décadas, la comunidad de la robótica ha realizado grandes avances desarrollando algoritmos que ayudan a los robots a funcionar con una mayor autonomía.

A pesar de los importantes progresos que se han tenido en el hardware, se siguen presentando grandes retos para los desarrolladores de software.

Se ha creado un sistema operativo robótico denominado ROS con el objetivo de apoyar la reutilización de código en la investigación y el desarrollo de robótica.

2.4.1. DEFINICIÓN

ROS es un sistema meta-operativo de código abierto. Proporciona los servicios esperados de cualquier sistema operativo, incluyendo la abstracción del hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el intercambio de mensajes entre procesos (nodos) y la gestión de paquetes. También proporciona herramientas y bibliotecas para obtener, construir, escribir y ejecutar el código en varios ordenadores [17].

2.4.2. CONCEPTOS BÁSICO

- **Paquete**: unidad principal para organizar el software en ROS. Un paquete puede contener nodos, una biblioteca independiente de ROS, un conjunto de datos, archivos de configuración, o cualquier otra cosa que se organice de forma útil. El objetivo de estos paquetes es proporcionar esta funcionalidad útil de una manera fácil de consumir para que el software pueda ser fácilmente reutilizado [17].

- Nodo: proceso que realiza cálculos. Los nodos se combinan en un gráfico y se comunican entre sí mediante temas de transmisión, servicios RPC y el servidor de parámetros. Un sistema de control de robots suele estar compuesto por muchos nodos. Se recomienda crear un único nodo para cada propósito y desarrollarlo para facilitar su reutilización. Un nodo ROS se escribe con el uso de una biblioteca de cliente ROS, como *roscpp* o *rospy* [17].

  En el caso de los robots móviles, el programa para operar el robot se divide en funciones especializadas. Se utiliza un nodo especializado para cada función, como el accionamiento del sensor, la conversión de datos del sensor, el reconocimiento de obstáculos, el accionamiento del motor, la entrada del codificador y la navegación.

- Master: proporciona servicios de nombramiento y registro al resto de los nodos del sistema ROS. Hace un seguimiento de los editores y suscriptores de los temas, así como de los servicios. Permite que los nodos individuales de ROS se localicen entre sí. Una vez que estos nodos se han localizado, se comunican entre sí de igual a igual [17].

- Mensajes: los nodos se comunican entre sí mediante el paso de mensajes. Un mensaje es una estructura de datos, que comprende campos tipificados. Se admiten los tipos primitivos estándar (entero, coma flotante, booleano, etc.), así como las matrices de tipos primitivos. Los mensajes pueden incluir estructuras y matrices anidadas de forma arbitraria. Los mensajes se enrutan a través de un sistema de transporte con semántica de publicación/suscripción [17].

- Topic (tema): Un nodo envía un mensaje publicándolo en un tema determinado. El tema es un nombre que se utiliza para identificar el contenido de un mensaje. Un nodo que esté interesado en un determinado tipo de datos se suscribirá al tema correspondiente. Puede haber varios editores y suscriptores simultáneos para un mismo tema, y un mismo nodo puede publicar y/o suscribirse a varios temas [17].

- Servicio: el modelo de publicación/suscripción es un paradigma de comunicación muy flexible, pero su transporte unidireccional de muchos a muchos no es apropiado para las interacciones de solicitud/respuesta, que a menudo se requieren en un sistema distribuido. La solicitud/respuesta se realiza a través de servicios, que están definidos por un par de estructuras de mensajes: una para la solicitud y otra para la respuesta [17].

2.4.3. VERSIONES

ROS sufre constantes modificaciones y mejoras por tratarse de un código abierto, por lo que existen diversas versiones desde la creación inicial del software.

A continuación se muestra una tabla [17] donde aparecen las distintas versiones, distinguiéndose las disponibles para su uso en verde y las descatalogadas en gris.

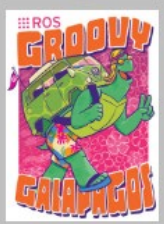
Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015
ROS Groovy Galapagos	December 31, 2012			July, 2014

Ilustración 24. Versiones de ROS

Se debe tener en cuenta que la versión de ROS puede variar dependiendo del robot que se utilice. En el caso del robot Niryo Ned, emplea la versión ROS Melodic.

2.4.4. COMUNICACIONES

El ROS Master actúa como un servicio de nombres en el gráfico de cálculo de ROS. Almacena información de registro de temas y servicios para nodos ROS. Los nodos se comunican con el Master para proporcionar su información de registro. A medida que estos nodos se comunican con el Master, pueden recibir información sobre otros nodos registrados y realizar conexiones según corresponda. El Master también realizará devoluciones de llamada a estos nodos cuando cambie esta información de registro, lo que permite que los nodos creen conexiones dinámicamente a medida que se ejecutan nuevos nodos [17].

Los temas actúan como bus que permiten el intercambio de información entre dos nodos. Un nodo puede publicar información o estar subscrito a varios temas.

En la figura 25 se puede observar el intercambio de información entre tres nodos mediante temas. Uno de los nodos publica información que es recibida por nodos subscriptores. Puede establecerse, por tanto, una comunicación multipunto unidireccional [17].

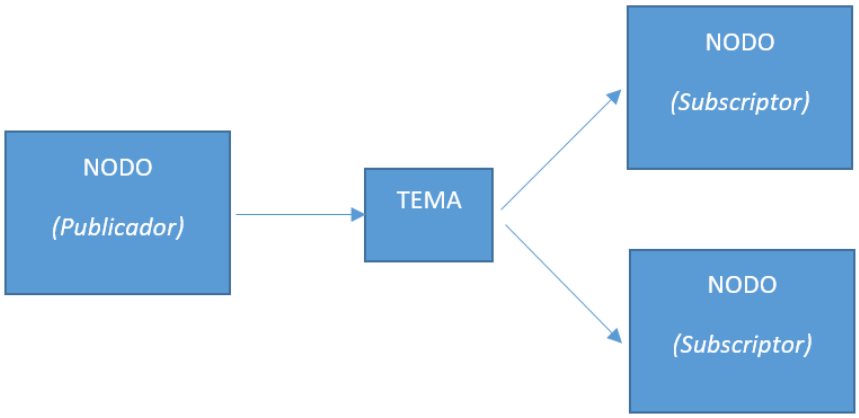


Ilustración 25. Diagrama de comunicación entre nodos mediante temas

Otra forma de comunicación es mediante los servicios. Un nodo actúa como servidor y los demás como clientes. Los nodos clientes realizan peticiones que debe atender el nodo servidor. La información de los servicios consta por tanto de los argumentos de petición y de la respuesta [17].

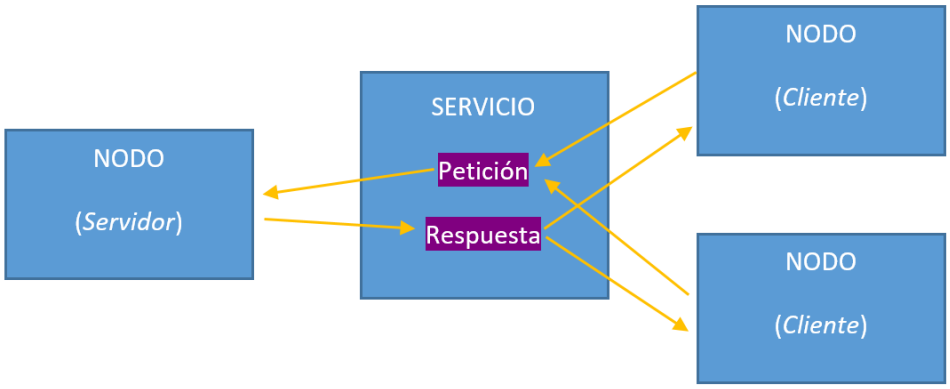


Ilustración 26. Diagrama de comunicación entre nodos mediante servicios

La última forma de comunicación entre los nodos, y la más compleja, es mediante acciones, construidas por servicios y temas. De igual forma que las comunicaciones anteriores, se puede realizar mediante varios nodos.

Por último, se debe tener en cuenta que los nodos de ROS utilizan una biblioteca cliente, como *rospy* (librería cliente de Python), para comunicarse con otros nodos, como se muestra a continuación:

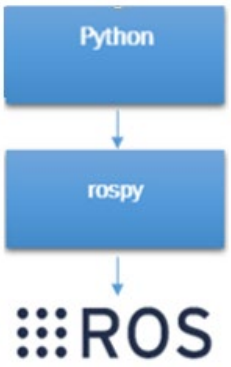


Ilustración 27. Comunicación entre Python y ROS

CAPÍTULO 3. NIRYO NED

3.1. ESPECIFICACIONES TÉCNICAS

Niryo Ned es un brazo robótico colaborativo de seis ejes diseñado para la educación y la investigación que está basado en Ubuntu 18.04 y en ROS Melodic. Éste sistema operativo permite que el robot cuente con múltiples librerías para concebir muchos programas, desde los más sencillos a los más complejos, respondiendo a sus necesidades [18].

En cuanto a su estructura externa, se compone de aluminio que le confiere la suficiente robustez para realizar con fluidez los movimientos necesarios en los proyectos de robótica [18].

Para realizar este proyecto se han tenido en cuenta algunas especificaciones técnicas de Niryo Ned para el diseño de la pantalla en App Designer, ya que será necesario conocer los límites del propio robot.

Este robot cuenta con 6 grados de libertad, pesa 6,5 kg, permite manejar una carga útil de 300 g y tiene un alcance de 440 mm.

En la siguiente figura se muestra un plano con las dimensiones del robot Niryo Ned [18]:

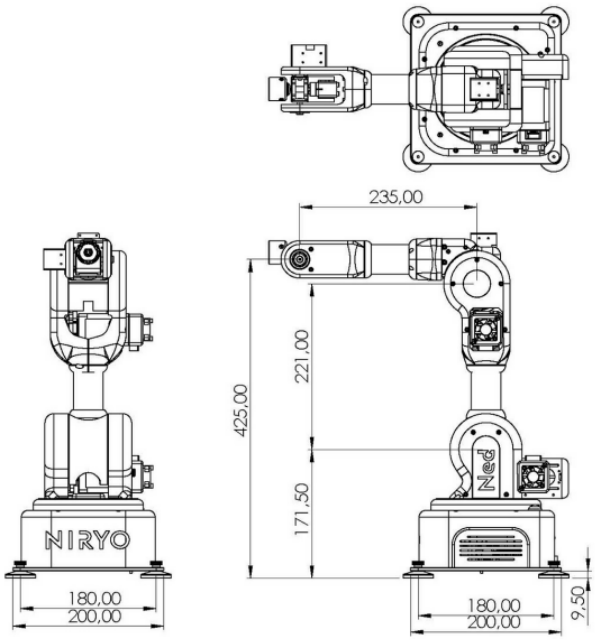


Ilustración 28. Planos de Niryo Ned

Niryo Ned tiene unos límites máximos de rotación correspondiente a cada articulación. En la siguiente tabla se presentan estos límites establecidos por el fabricante y una figura que muestra dónde se encuentra cada articulación:

Tabla 3. Límites de máxima rotación de Niryo Ned

	Mínimo	Máximo
J1	-170º	170º
J2	-120º	35º
J3	-77º	90º
J4	-120º	120º
J5	-100º	55º
J6	-145º	145º

Ilustración 29. Numeración de las articulaciones de Niryo Ned

A continuación se pueden observar dos imágenes donde se aprecia el rango de trabajo de éste robot [18]:

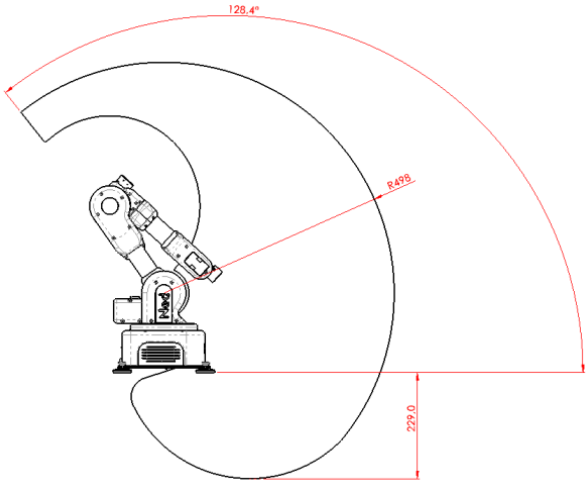


Ilustración 30. Perfil del área de trabajo de Niryo Ned

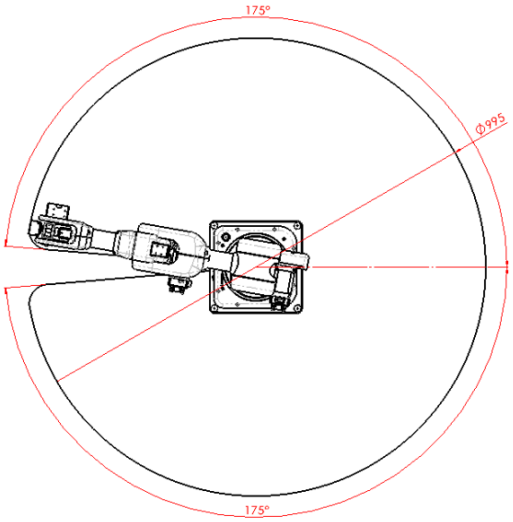


Ilustración 31. Alzado del área de trabajo de Niryo Ned

3.2. SIMULACIÓN Y COMUNIACIÓN CON ROS DE NIRYO

Para realizar la comprobación de los distintos movimientos programados para el robot Niryo Ned, se puede hacer mediante el robot real o con distintas herramientas de visualización 3D, como son *Rviz* y *Gazebo*. Esto se organiza según el siguiente diagrama [19]:

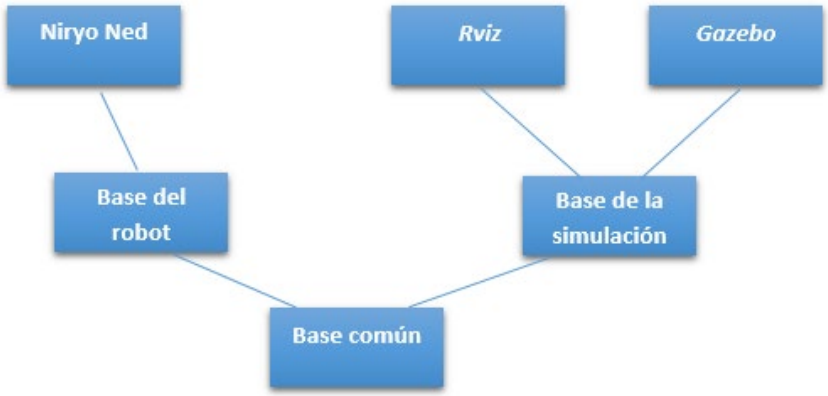


Ilustración 32. Diagrama de control de Niryo Ned, real o simulado

Rviz es una herramienta de visualización 2D y 3D para aplicaciones de ROS. Su objetivo es proporcionar una vista del modelo del robot, capturar la información de los sensores en el sistema interno del robot y reproducir los datos capturados.

La instalación se realiza al descargar el paquete completo de ROS [17]. Además, cuando se descargan los archivos pertenecientes al paquete Niryo, éste incluye un archivo propio de esta aplicación especificando las condiciones

del entorno que se quiere crear, el sistema de referencia principal y algunas características propias de los objetos a representar, como es el robot Niryo Ned. En la siguiente figura se muestra la simulación mediante esta herramienta:

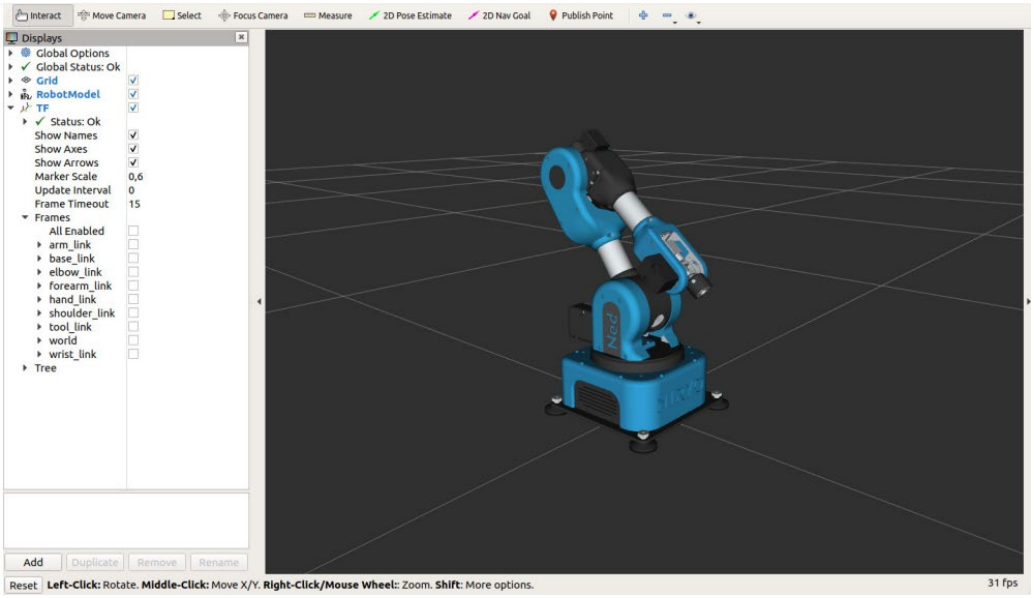


Ilustración 33. Simulación de Niryo Ned en Rviz

La otra herramienta de visualización es *Gazebo*, un simulador de entornos 3D que posibilita la evaluación del comportamiento de varios robots de forma virtual. Permite diseñar robots personalizados, crear mundos virtuales usando herramientas sencillas como CAD o importar modelos ya creados. Además, es posible sincronizarlo con ROS para que los robots publiquen la información de sus sensores en nodos, y así implementar una forma lógica y un control que de órdenes al robot.

De igual forma que en *Rviz*, los paquetes de Niryo incluyen un archivo del robot Niryo Ned para posibilitar su simulación en *Gazebo*, como se muestra a continuación:

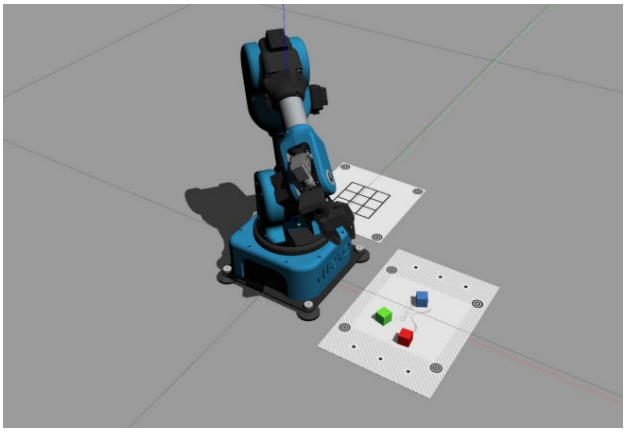


Ilustración 34. Simulación de Niryo Ned en Gazebo

Por otro lado, el ROS de Niryo cuenta con tres formas distintas de comunicarse con el usuario, mediante PyNiryo, Niryo Studio o Modbus Client, así se muestra a continuación:

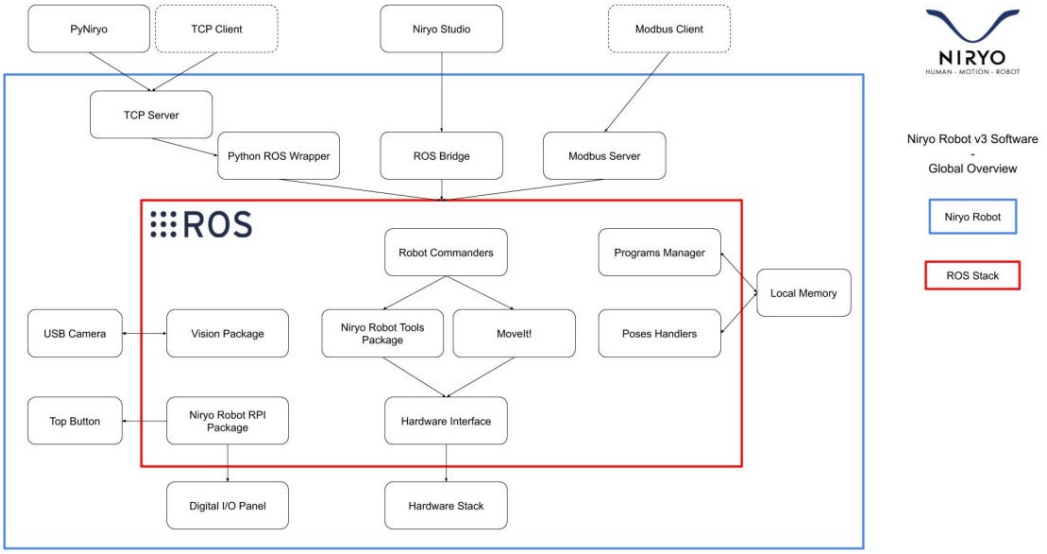


Ilustración 35. Estructura de comunicación con el ROS de Niryo

Python ROS Wrapper obliga al usuario a escribir su código directamente en el robot o, al menos, a copiar el código en el robot mediante un comando de terminal. Como se quiere ejecutar el código desde el ordenador, se debe utilizar el paquete de Python, PyNiryo [19]. Sin embargo, es necesaria la librería *rospy* para establecer la comunicación entre PyNiryo y ROS.

Rospy es una biblioteca de Python dirigida a ROS que permite a los programadores de Python interactuar rápidamente con temas, servicios y parámetros de ROS. Esta biblioteca favorece la velocidad de implementación para que los algoritmos puedan ser fácilmente prototipados y probados en ROS [17].

En la siguiente figura se muestra un esquema de esta primera forma de comunicación con ROS:

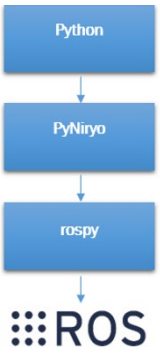


Ilustración 36. Diagrama de comunicación Python-ROS

Otra forma de comunicarse es mediante Niryo Studio, HMI gráfica que permite un control rápido y directo del robot a través de un ordenador externo. Su objetivo es proporcionar a los usuarios una interfaz completa y sencilla para el movimiento de Niryo Ned, los entornos de programación y el estado actual de Ned. Más adelante se podrá encontrar más información sobre esta aplicación.

La última forma de comunicación es mediante Modbus Client. Este protocolo se usa frecuentemente para transmitir señales de los dispositivos de instrumentación y control, como un PLC (Programmable Logic Controller), a un controlador principal.

3.3. CONEXIONES

Se puede realizar la conexión con Niryo Ned de tres formas distintas:

- Modbus TCP (master)
- TCP/IP
- CAN (esclavo, compatible con los motores NiryoStepper)

En este proyecto se ha establecido la conexión con el robot mediante el modelo TCP/IP.

Este modelo hace referencia a dos protocolos de red que permiten la transferencia de datos entre redes de ordenadores. El primer protocolo que interviene en este modelo, denominado Protocolo de Control de Transmisión (TCP), proporciona un transporte fiable de datos dentro del nivel de transporte de referencia OSI (Open System Interconexion). Mientras que el segundo, designado Protocolo de Internet (IP), ofrece la posibilidad de dirigir los datos a otras máquinas como sería el robot Niryo Ned [20].

Este modelo está constituido por distintas capas para conseguir un intercambio fiable de datos entre dos equipos de una forma más sencilla. Cada capa se construye sobre su predecesora, siendo los servicios y funciones de cada una variables dependiendo del tipo de red. Sin embargo, la misión de cada capa en cualquier red es proveer servicios a las capas superiores. De esta forma se tienen los siguientes niveles [20]:

- Nivel de aplicación. Es el más alto y en él se encuentran diversas aplicaciones que tienen la capacidad de acceder a servicios vía Internet.
- Nivel de transporte. Se encarga de ofrecer una comunicación entre los extremos de programas de aplicación.

- Nivel de red. Realiza una serie de acciones sobre la información que recibe del nivel anterior para luego acometer el envío un nivel inferior.
- Nivel de enlace. Su misión es transmitir la información recibida del hardware.

En la siguiente figura se muestra un esquema con estos niveles:

Ilustración 37. Diagrama de niveles del modelo TCP/IP

Este modelo emplea las relaciones cliente/servidor. El servidor se trata de una aplicación que ofrece un servicio a los usuarios de Internet mientras que el cliente es quien lo solicita.

En el segundo protocolo que interviene en este modelo aparecen las direcciones IP, un identificador numérico asignado a cualquier dispositivo conectado a Internet. Este mecanismo permite encontrar la información que se busca al conectarse a los dispositivos y sitios web.

Existen dos tipos de direcciones: pública y privada. La dirección IP pública permite identificarte en Internet, de modo que toda la información buscada pueda llegarte. Mientras que la dirección IP privada o local es utilizada dentro de una red privada para conectarte de forma segura a otros dispositivos de la misma red [20].

Las direcciones IP locales tienen reservado una serie de intervalos específicos y nunca deberían de aparecer en Internet. Estos intervalos son los siguientes:

- Clase A: 10.0.0.0 – 10.255.255.255
- Clase B: 172.16.0.0 – 172.31.255.255
- Clase C: 192.168.0.0 – 192.168.255.255

En la siguiente imagen se muestra un ejemplo de las conexiones existentes entre los distintos dispositivos usados a diario y sus correspondientes direcciones:

Ilustración 38. Ejemplo de comunicación entre dispositivos cotidianos

A partir del modelo que se ha explicado anteriormente basado en TCP/IP, se tienen tres formas distintas de conectarse con el robot Niryo Ned real y una cuarta forma de conectarse con Niryo Ned simulado [21]:

1. **Modo Hotspot**. La dirección IP correspondiente es 10.10.10.10. Este modo será el empleado en este proyecto para realizar la conexión con el robot real.

2. **Mediante red Wi-fi**. Para emplear este modo debe estar conectado el ordenador a la misma WIFI que Ned.

3. **Cable de Internet**. Se conecta al robot mediante el puerto *Lan*.

4. **Simulación con Niryo Studio**. El simulador *Rviz* o *Gazebo* se inicia en el mismo ordenador que se ejecuta Niryo Studio. En este modo se conecta al *Localhost* correspondiente con la dirección IP 127.0.0.1.

3.4. NIRYO STUDIO

Niryo Studio es una HMI (Human-Machine Interface) gráfica que permite un control rápido y directo del robot mediante un ordenador externo. Su finalidad es proporcionar a los usuarios una interfaz completa y sencilla para realizar los movimientos de Niryo Ned, los entornos de programación y el estado del mismo [21].

A través de esta aplicación, los usuarios pueden:

- Administrar y configurar los parámetros de Ned.
- Controlar el movimiento de Ned.
- Programar a Ned usando *Blockly*, un lenguaje de programación de alto nivel que emplea bloques visuales los cuales representan un fragmento de código. De esta forma, se pueden crear secuencias complejas de forma más fácil, sin necesidad de saber el comando necesario de cada lenguaje de programación. La siguiente figura muestra esta función:

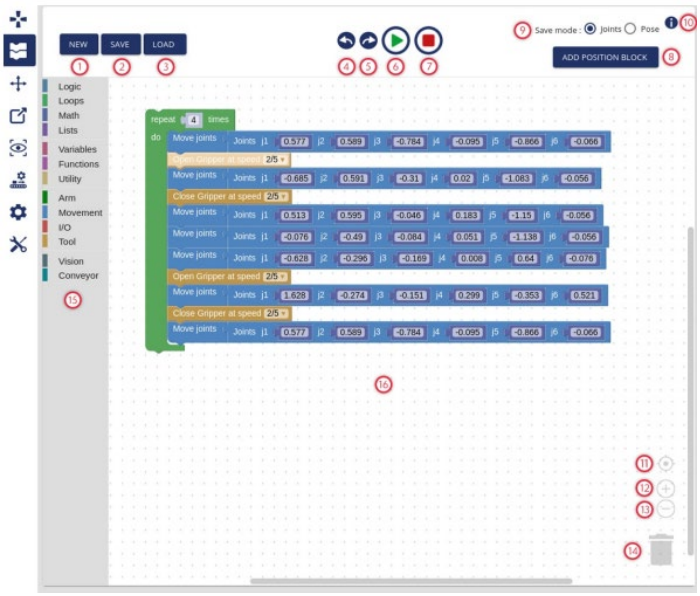


Ilustración 39. Función Blockly de Niryo Studio

- Ejecutar un programa.
- Gestionar, configurar, controlar y programar las herramientas de Ned.
- Administrar, configurar, controlar y programar los complementos de Ned.

Niryo Studio presenta la siguiente interfaz para realizar las operaciones:

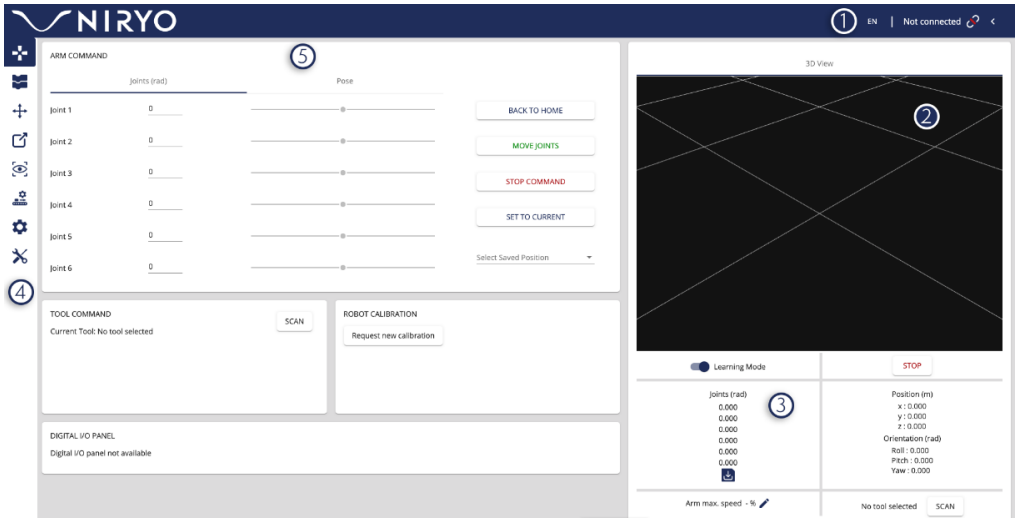


Ilustración 40. Interfaz de Niryo Studio

1. Barra de herramientas que permite seleccionar el idioma y conectarse con el robot.
2. Visualización 3D del robot o transmisión de video de la cámara.
3. Estado del robot.
4. Distintas secciones que existen en la aplicación, entre las que se incluye el modo Blockly.
5. La ventana principal.

En la pantalla principal se encuentran distintas ventanas con cremalleras que permiten controlar el robot por cinemática directa e inversa.

También incluye el control mediante la pulsación de flechas para realizar el movimiento del TCP o el control de cada articulación según se desee y establecer la velocidad de este movimiento.

CAPÍTULO 4. DESARROLLO DEL PROYECTO

4.1. *SimRob* DE NIRYO NED

A la hora de crear las estaciones de trabajo en Simulink en las cuales aparece un robot, entre otros componentes, se debe considerar que los brazos y las articulaciones del robot quedan definidos en un formato universal denominado URDF (Universal Robot Defined Format), que usa ficheros *xml*. Simulink (*Simscape*) es capaz de leer los ficheros con este formato y traducirlo a un icono de este entorno.

El archivo URDF que representa al robot, es distinto para un modelo cinemático que para uno dinámico, ya que cada uno tiene diferentes características.

En ambos modelos, el robot se define en un subsistema con máscara con dos variables de entrada, las coordenadas de la base del robot [x,y,z,r_z,r_y,r_x] y los valores iniciales de los ángulos de las articulaciones, como se muestra en la siguiente figura:

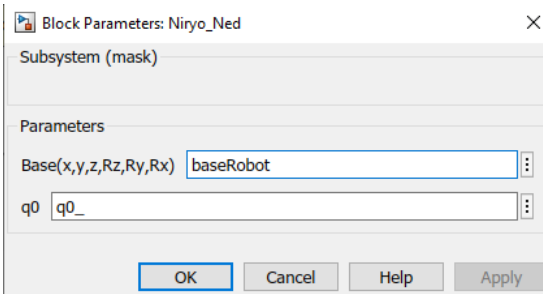


Ilustración 41. Variables de entrada a la máscara del robot

Para el estudio y desarrollo de este proyecto, se han creado tres tipos de estaciones con diferentes características: modelo cinemático, modelo dinámico con un lazo y modelo dinámico con dos lazos

4.1.1. MODELO CINEMÁTICO

Se ha diseñado una estación robótica a partir de la librería *Simscape Multibody* de Simulink para estudiar el modelo cinemático. Este modelo incorpora el robot Niryo Ned, una pinza, una carga que en un principio es *LapExt* (carga nula) necesaria para intercambiarla por el *DadoP* y así simular que se está cogiendo el dado, otro dado que tiene como referencia el tablero de ajedrez, y distintos

elementos necesarios para crear la estación que han sido explicados en el *Capítulo 2*. En la siguiente figura se puede ver el resultado de la estación:

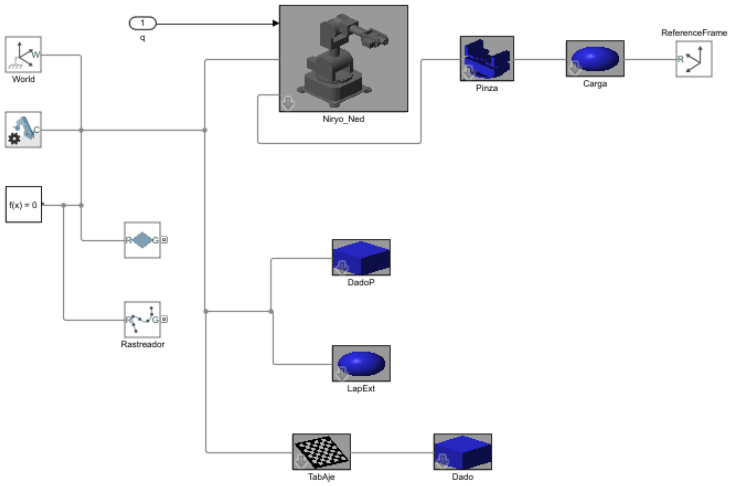


Ilustración 42. Estación para el modelo cinemático

El icono del robot, figura 43, presenta dos señales de entrada, una de *Simscape* para asociarlo al sistema mecánico y otra de Simulink que introduce los ángulos de las articulaciones del robot. Además, tiene una variable de salida de *Simscape* para asociarlo a otros componentes mecánicos.

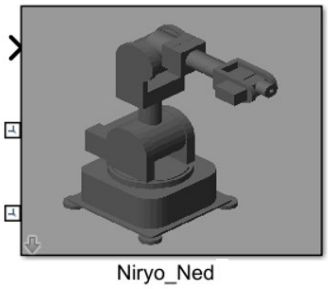


Ilustración 43. Icono del robot del modelo cinemático

En este modelo, las entradas en las articulaciones son los ángulos de las respectivas articulaciones, sin embargo, no existen salidas. Este esquema URDF se muestra en la figura 44.

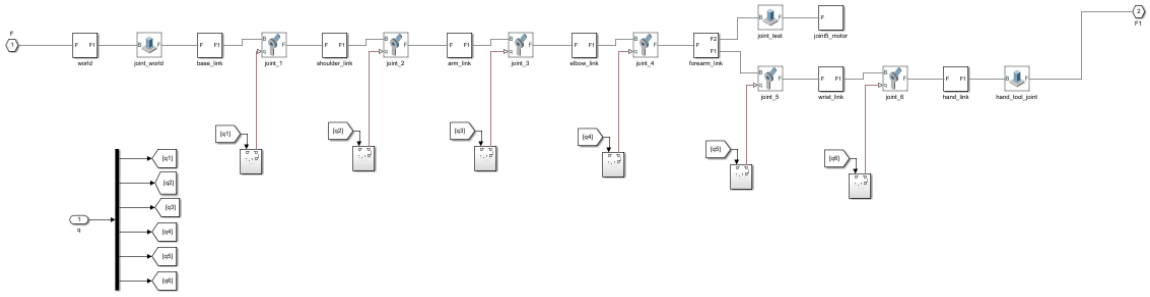


Ilustración 44. URDF del robot del modelo cinemático

4.2.2. MODELO DINÁMICO CON UN LAZO

Para el diseño de un modelo dinámico es necesario realizar un sistema en lazo cerrado para conseguir mover al robot a la posición deseada, es decir, que la señal de salida siga a la de referencia.

La estación que se muestra en la figura 45, está compuesta por una pinza, una carga (nula), un dado, el robot Niryo Ned, el lazo cerrado y los elementos ya explicados que son imprescindibles para construir una estación.

El lazo cerrado se compone de un controlador proporcional, en el cual se introduce el valor de *kcontrol* que se desee, necesario para convertir la señal que proporciona los ángulos en una señal con los pares de las articulaciones, y un elemento de saturación para representar una limitación que tendrían los actuadores en la realidad, ya que limita los pares que pueden ejercer.

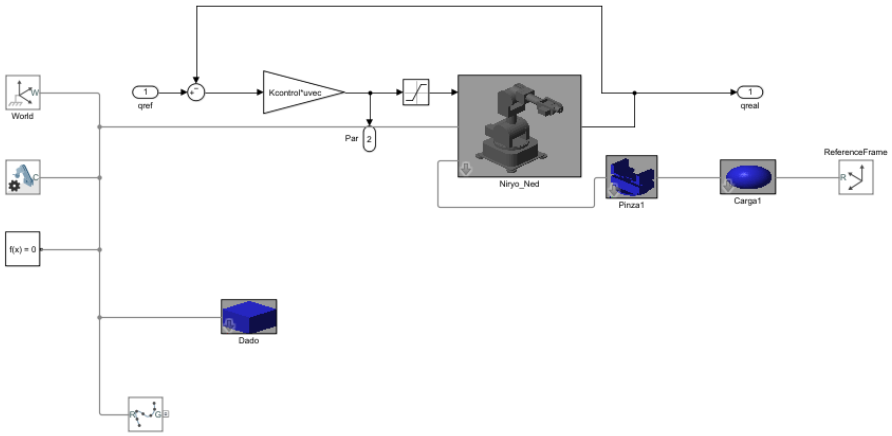


Ilustración 45. Estación para el modelo dinámico con un lazo

El icono del robot, figura 46, contiene dos señales de entrada, una de *Simscape* para asociarlo al sistema mecánico y otra de Simulink que introduce los ángulos de las articulaciones del robot, al igual que se ha explicado en el modelo cinemático. Sin embargo, a diferencia que el modelo anterior, tiene dos salidas, una de *Simscape* para asociarlo a otros componentes mecánicos y otra necesaria para realizar la realimentación de las posiciones angulares.

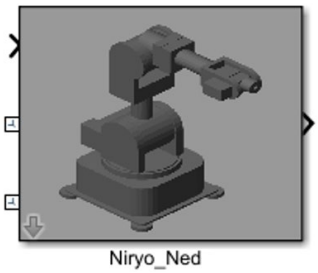


Ilustración 46. Icono del robot del modelo dinámico con un lazo

Las entradas en las articulaciones son los pares correspondientes a cada una, no los ángulos. En este caso, sí existen salidas en las articulaciones, las cuales miden el ángulo de cada una.

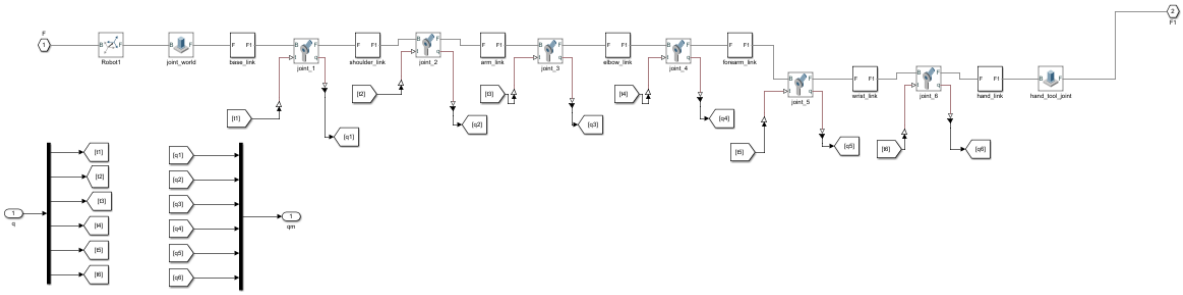


Ilustración 47. URDF del robot del modelo dinámico con un lazo

4.2.3. MODELO DINÁMICO CON DOS LAZOS

La estación que se presenta en la figura 46 es similar a la descrita en el apartado anterior, excepcionando la adición de un nuevo lazo. El controlador del sistema en lazo cerrado determina las oscilaciones que tiene el robot en movimiento, por lo tanto, se ha introducido un nuevo lazo interno que realimenta la velocidad para reducir las oscilaciones del sistema.

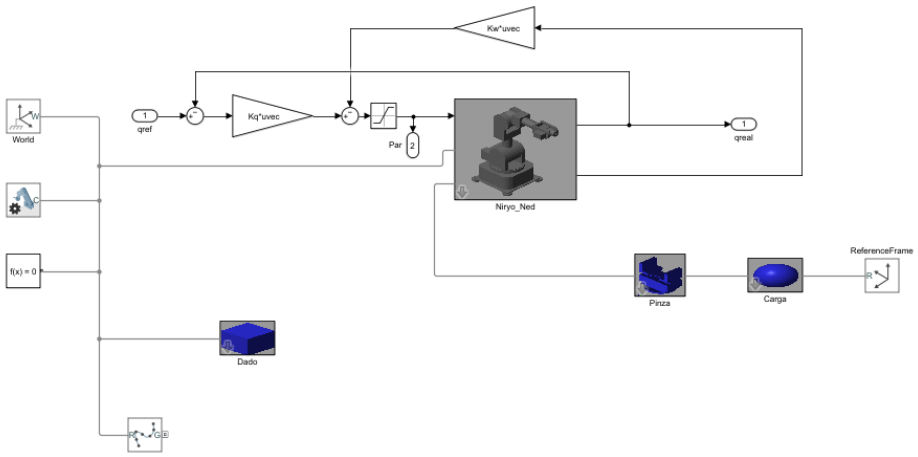


Ilustración 48. Estación para el modelo dinámico con dos lazos

Al haber incluido este lazo, se ha modificado el icono del robot, como se muestra en la figura 49, añadiendo una salida necesaria para realizar la realimentación de la velocidad.

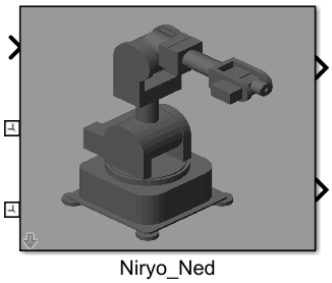


Ilustración 49. Icono del robot del modelo dinámico con dos lazos

Como consecuencia, se ha incluido una nueva salida en las articulaciones con el objetivo de medir la velocidad de éstas, como se muestra en la siguiente figura:

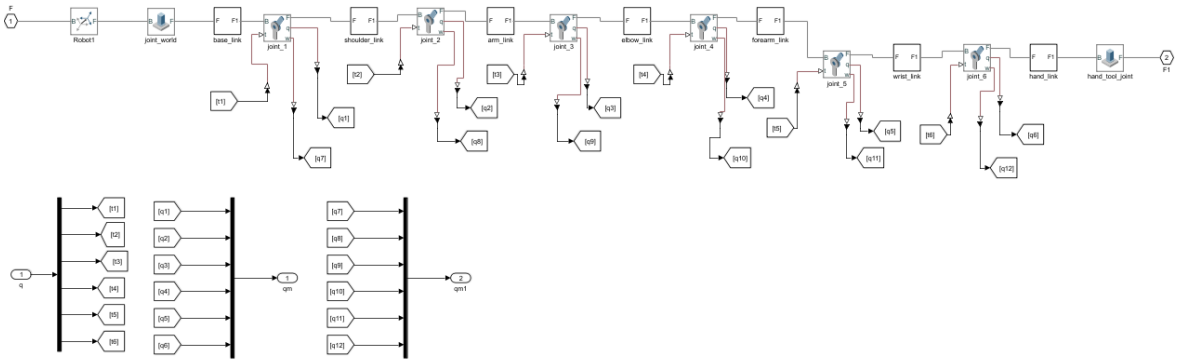


Ilustración 50. URDF del robot del modelo dinámico con dos lazos

4.2. COMUNICACIONES

Niryo Ned es un robot colaborativo basado en Ubuntu 18.04 y ROS Melodic, una solución de código abierto creada para la robótica. Por lo tanto, para comenzar a usar este robot se deben instalar diversas aplicaciones, como Ubuntu, que se describirán en este capítulo, además de las correspondientes instrucciones para su instalación.

La última versión del *kernel* de Linux permite ejecutar operaciones complejas como la pila completa de ROS. Por lo tanto, es capaz de efectuar simulaciones para los robots Ned, Niryo One o Ned2 en una máquina Windows.

Ubuntu es una distribución de Linux basada en *Debian* que incluye software libre y código abierto. La versión necesaria para Niryo Ned es la 18.04, lanzada el 26 de Abril de 2018 y con un soporte de 5 años.

Se debe tener en cuenta que esta pantalla solo es capaz de abrir comandos, por lo que será necesario la descarga de una pantalla gráfica para poder

realizar las simulaciones lanzadas desde Ubuntu. Por lo tanto, se debe descargar *XLaunch*, un programa gráfico para crear ventanas gráficas.

Una vez se hayan instalado estas aplicaciones, ya será posible la descarga de ROS Melodic.

ROS dispone de una herramienta de visualización 3D, llamada *Rviz*, donde se pueden representar los robots. La finalidad de esta herramienta en el proyecto es simular el robot Niryo Ned con una determinada IP a la cual se le mandan distintas instrucciones, y así comprobar que las instrucciones funcionarían en el robot real [17].

Debido a la complejidad de las comunicaciones entre el usuario y el ROS de Niryo Ned, explicadas en el *Capítulo 2*, se ha decidido crear un interface en Matlab para facilitar al usuario el control del robot. Para ello se establecerá una conexión entre Matlab y Python mediante varios archivos, indicados en las instrucciones, que permiten crear un enlace entre ambas aplicaciones. Finalmente, el diagrama de comunicaciones entre aplicaciones que se obtiene es el siguiente:

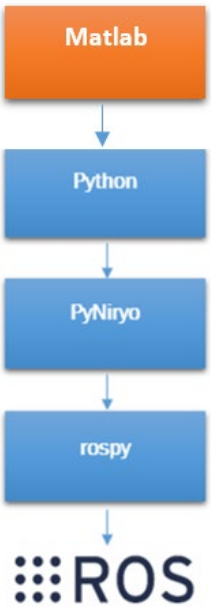


Ilustración 51. Diagrama de comunicación Matlab-ROS

4.2.1. INSTALACIÓN DE LAS APLICACIONES

4.2.1.1 Instalación de la pantalla virtual WSL2

Se debe abrir una ventana denominada “Windows PowerShell” para introducir los comandos que se mostrarán a continuación.

1. Habilitar el subsistema de Windows para Linux

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-
Subsystem-Linux /all /norestart
```

2. Actualizar WSL para utilizar la versión 2

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform
/all /norestart
```

3. Reiniciar el ordenador para finalizar la instalación de WSL y la actualización a WSL2.

4. Establecer la versión 2 de WSL por defecto.

```
wsl --set-default-version 2
```

5. Instalar Ubuntu 18.04 LTS utilizando Microsoft Store.

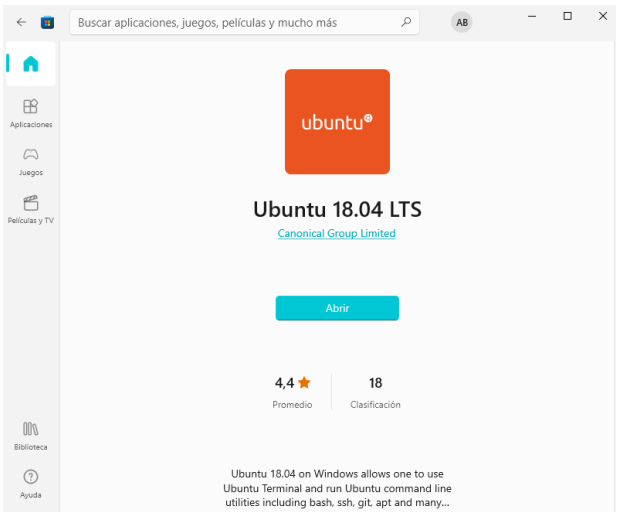


Ilustración 52. Ubuntu en Microsoft Store

6. Iniciar la aplicación y esperar a que se termine la instalación. Una vez terminada, se debe introducir un nombre de usuario y una contraseña para crear una cuenta.

4.2.1.2. Programa gráfico para crear ventanas gráficas

WSL no tiene un servidor X por lo que no podrá lanzar ninguna ventana gráfica.

Para descargar un servidor X para Windows 10, se debe acceder a siguiente enlace: https://sourceforge.net/projects/vcxsrv/

1. Abrir XLaunch configurándolo como se muestra a continuación:

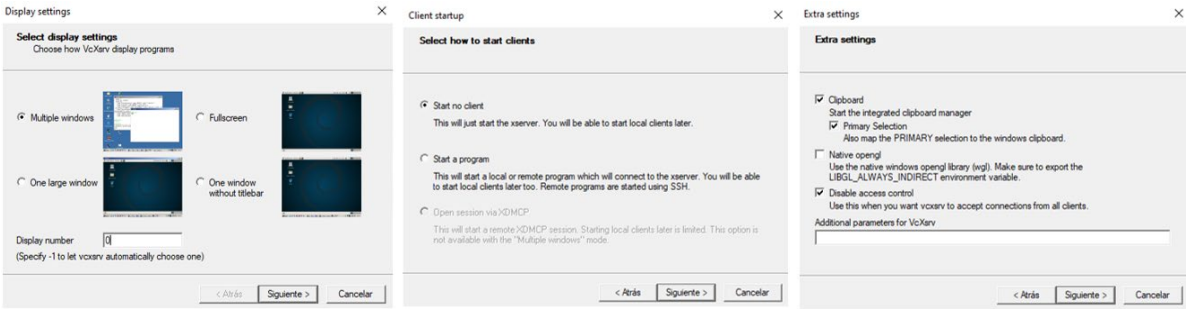


Ilustración 53.Configuración de XLaunch

Una vez finalizadas estas ventanas, el servidor se mantendrá en segundo plano.

Se debe abrir el terminal de Ubuntu para introducir el siguiente código.

2. Instalar el paquete de comunicación con la pantalla.

```
$ sudo apt update

$ sudo apt install x11-apps
```

3. Exportar la dirección de XLaunch en Ubuntu.

```
$ export DISPLAY=$(cat /etc/resolv.conf | grep nameserver | awk '{print $2}'):0
```

Introducirlo en el fichero de entrada ~/.bashrc para que se active de forma automática cada vez que abrimos una pantalla.

```
$ echo export DISPLAY=$(cat /etc/resolv.conf | grep nameserver | awk
'{print $2}'):0 >> ~/.bashrc
```

Volver a lanzar el inicio.

```
$ source ~/.bashrc
```

4.2.1.3. Introducción de ROS Melodic en la pantalla wsl2

ROS Melodic está principalmente dirigido a la versión Ubuntu 18.04, aunque también es compatible en otros sistemas Linux, Mac OS X, Android y Windows.

Introducir el código en el terminal de Ubuntu.

1. Instalar el editor de texto (gedit).

```
$ sudo apt-get update

$ sudo apt-get install gedit
```

2. Configurar el ordenador para que acepte el 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'
```

3. Configurar tus claves.

```
$ sudo apt install curl # if you haven't already installed curl

$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc
| sudo apt-key add -
```

4. Instalar.

Se produce la instalación completa de ROS, rqt, Rviz, bibliotecas genéricas de robots, simuladores 2D/3D y percepción 2D/3D.

```
$ sudo apt update

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

5. Comprobar qué paquetes se han instalado.

```
$ apt search ros-melodic
```

4.2.1.4. Instalación del paquete Niryo

Es preferible realizar la descargar de los paquetes de Niryo en un directorio que se pueda ver desde Windows, usando sus discos montados, y así crear una carpeta llamada “Niryo”. Por ejemplo: "C:\Usuarios\alicia\Niryo"

Introducir el siguiente código en el terminal de Ubuntu.

1. Dirigir las descargas a la carpeta que se ha creado.

```
$ cd /mnt/c/users/alici/Niryo
```

2. Espacio en disco y copia.

```
$ mkdir -p catkin_ws_niryo_ned/src
$ cd catkin_ws_niryo_ned
```

3. Instalar Git.

```
$ sudo apt install git
```

4. Descargar los ficheros de Github.

```
$ git clone https://github.com/NiryoRobotics/ned_ros src
```

5. Instalar paquetes propios para la aplicación de Niryo.

```
$ sudo apt install build-essential
$ sudo apt install catkin
$ sudo apt install python-catkin-pkg
$ sudo apt install python-pymodbus
$ sudo apt install python-rosdistro
```

```
$ sudo apt install python-rospkg
$ sudo apt install python-rosdep-modules
$ sudo apt install python-rosinstall python-rosinstall-generator
$ sudo apt install python-wstool
$ sudo apt install ros-melodic-moveit
$ sudo apt install ros-melodic-control
$ sudo apt install ros-melodic-controllers
$ sudo apt install ros-melodic-tf2-web-republisher
$ sudo apt install ros-melodic-rosbridge-server
$ sudo apt install ros-melodic-joint-state-publisher-gui
$ sudo apt install ros-melodic-joint-trajectory-controller
```

6. Dirigirse al directorio de *catkin_ws_niryo_ned*

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

Para que recuerde siempre esta dirección, introducir:

```
$ echo "source $(pwd)/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

7. Modificar la pantalla virtual wsl2

Después de lanzar ifconfig, se cambia el fichero ~/.bashrc para modificar ROS_MASTER_URI y ROS_IP de forma automática y así no tener que cambiarlo manualmente.

Abrir una ventana gráfica.

```
$ gedit ~/.bashrc
```

Copiar el siguiente código en las últimas líneas de la ventana.

```
$ export DISPLAY=$(cat /etc/resolv.conf | grep nameserver | awk
'{print2}'):0
$ ip=$(ip -o -4 addr list eth0 |awk '{print $4}' |cut -d / -f1)
$ export ROS_MASTER_URI=http://$ip:11311
```

```
$ export ROS_IP=$ip
```

Debe quedar como se muestra en la siguiente imagen:

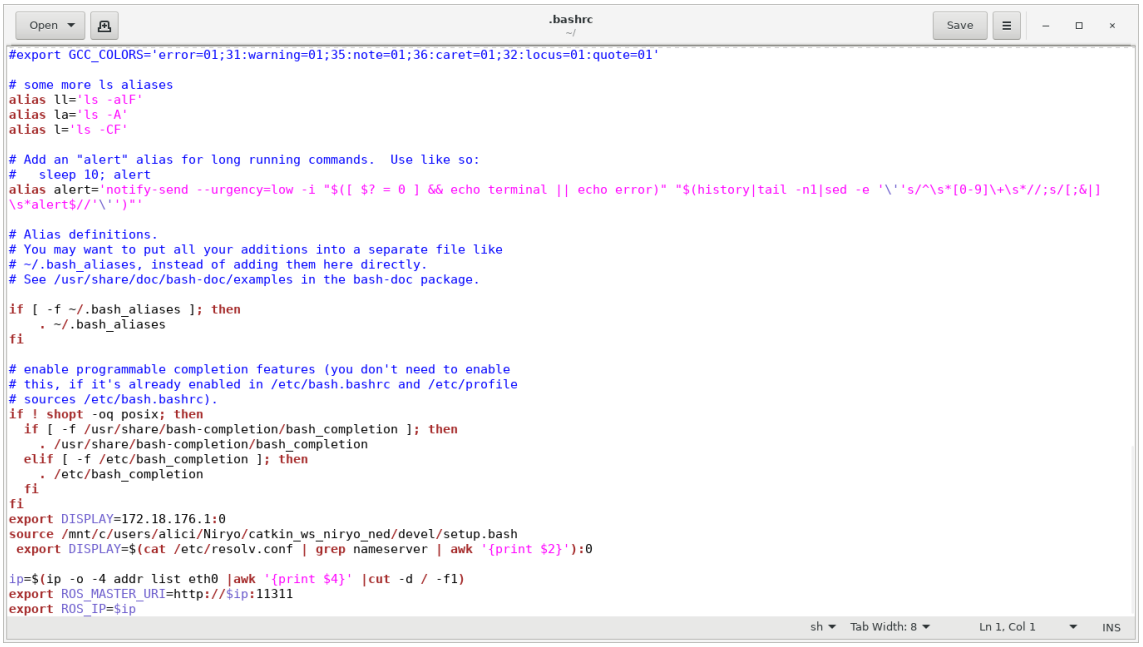


Ilustración 54. Modificación de ~/.bashrc

8. Para abrir Niryo con control externo tipo *teacher*.

```
$ roslaunch niryo_robot_description display.launch
```

9. Para abrir Niryo con *Rviz* con control por ROS.

```
$ roslaunch niryo_robot_bringup desktop_rviz_simulation.launch
```

10. Para abrir Niryo con simulación por *Gazebo*.

```
$ roslaunch niryo_robot_bringup desktop_gazebo_simulation.launch
```

11. Para facilitar el inicio de las simulaciones, crear una ventana gráfica llamada “inicio”.

```
$ gedit inicio
```

De esta forma, cada vez que se introduzca el código:

```
$ cat inicio
```

Se mostrarán las 3 formas de abrir Niryo y se podrá elegir el modo de simulación que se prefiera.

4.2.1.4. Instalación Niryo Studio

Se descarga la carpeta que contiene el programa *NiryoStudio.exe* que permite realizar la conexión con nuestro robot simulado y controlarlo.

4.2.1.5. Instalación de *Anaconda*

1. Descargar el software Anaconda

Este software será necesario para poder usar Python. Se descarga en el siguiente enlace: https://www.anaconda.com/products/individual

Hay que tener en cuenta a la hora de la descarga, incluir el paquete PATH, necesario para realizar la conexión entre Matlab y Python.

La instalación de Anaconda se introduce en una carpeta creada en el directorio "C:/Usuarios/alicia/anaconda3".

Para la comunicación entre Matlab y Python, son necesarios unos archivos de Matlab, denominados *pyLines.m* y *pyModule.m* incluidos en los Anexos, que se guardan en el directorio "C:/Usuarios/alicia/Niryo/programas". La conexión entre ambos programas se comprueba introduciendo el siguiente código en Matlab2022a que es la compatible con Python3.9 y se debe obtener este resultado:

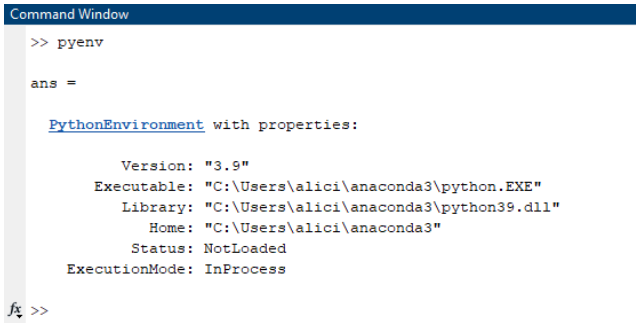


Ilustración 55. Comprobación de la conexión Matlab-Python

2. Abrir la ventana Anaconda Powershell Prompt.

Introducir el siguiente código:

```
$ pip install pyniryo
```

De esta forma se instalan los archivos necesarios para conectar Python con Niryo dentro de la carpeta anaconda3.

Ejecutando en Matlab el archivo *Live_PyNiryo.mlx*, adjunto en los Anexos, se comprobará si se puede controlar el robot Niryo desde Matlab.

4.3. APLICACIÓN PARA NIRYO NED

En este apartado se aborda el tema relacionado con la creación de la aplicación, explicando detalladamente cada una de las funciones que ofrece. El código que posee cada botón se adjunta en los *Anexos*.

La aplicación se encarga de establecer la comunicación entre el usuario y el robot, en este caso, Niryo Ned, aunque con pequeñas variaciones podría ser útil para cualquier otro robot. Se ha diseñado con un objetivo educacional y además, demostrar cómo se puede manipular este robot desde una aplicación distinta a las existentes hasta ahora, como es Niryo Studio.

La interfaz creada permite mandar instrucciones tanto a una estación de Matlab que contenga el robot que se quiere controlar, siendo esta la explicada anteriormente *SimRob* de Niryo Ned, como al robot real o a la simulación del robot en *Rviz*, usando en estos dos últimos su IP. La figura 56 muestra un esquema de la comunicación que permite realizar la aplicación creada:

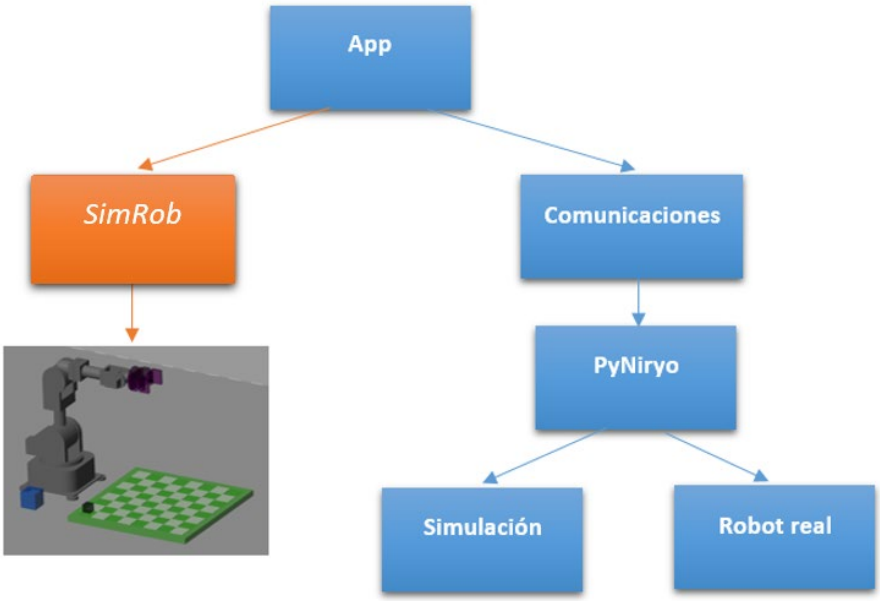


Ilustración 56. Diagrama de comunicaciones de la app

En la siguiente imagen se tiene una visión global de la pantalla al iniciarla:

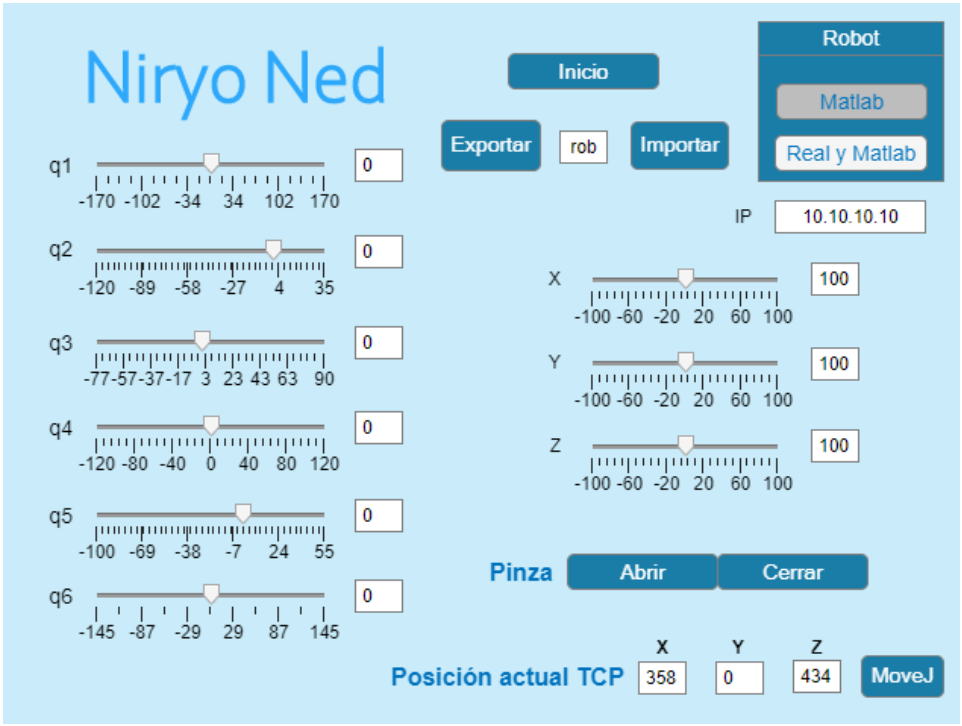


Ilustración 57. Aplicación para Niryo Ned

A continuación se van a describir detalladamente las distintas funciones que se pueden realizar con esta interfaz.

➢ MOVIMIENTO POR CINEMÁTICA DIRECTA

Se compone de 6 cremalleras, correspondientes a los ángulos de las articulaciones del robot, como se ve en la siguiente figura:

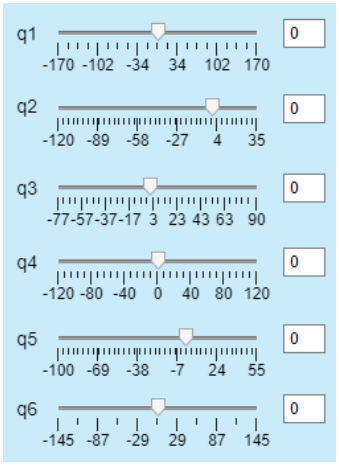


Ilustración 58. Cremalleras para cinemática directa

Estas cremalleras, pertenecientes al componente *Slider*, permiten controlar los movimientos del robot mediante cinemática directa, mientras se introducen los valores de los ángulos, se visualiza las posiciones que el robot va tomando. Además, se han incluido unas casillas, mediante el componente *Edit Field (Text)*, a la derecha de cada cremallera para introducir el valor exacto del ángulo en grados.

Se han definido los ángulos de máxima rotación de cada articulación del robot Niryo Ned como los límites de las cremalleras. Estos límites pueden modificarse en las siguientes líneas de la función *startFcn()* que se encuentra en la sección del código:

```
app.q1Slider.Limits = [-170,170];
app.q2Slider.Limits = [-120,35];
app.q3Slider.Limits = [-77,90];
app.q4Slider.Limits = [-120,120];
app.q5Slider.Limits = [-100,55];
app.q6Slider.Limits = [-145,145];
```

➢ MOVIMIENTO POR CINEMÁTICA INVERSA EN COORDENDAS RELATIVAS AL TCP

Se compone de 3 cremalleras, correspondientes a la distancia que debe avanzar el TCP (punto de trabajo) en cada eje, como se muestra en la figura 59:

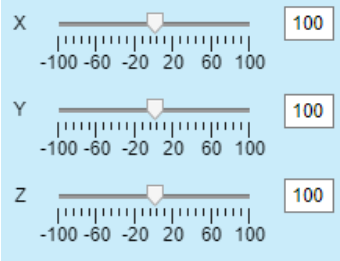


Ilustración 59. Cremalleras para cinemática inversa con coordenadas relativas al TCP

Estas cremalleras permiten el movimiento del robot mediante cinemática inversa, es decir, aportando al robot una coordenada, en mm, a la cual se quiere llegar. En este caso, la coordenada seleccionada tiene como referencia el TCP del robot, por lo que se puede conseguir una mayor precisión en el movimiento.

Se ha incorporado una casilla a la derecha de cada cremallera con el objetivo de variar los límites de ésta según desee el usuario.

➢ MOVIMIENTO POR CINEMÁTICA INVERSA EN COORDENDAS ABSOLUTAS

Esta parte de la pantalla se compone de 3 casillas en las cuales se introduce la coordenada, en mm, a la que se quiere desplazar, y del botón *MoveJ* para realizar el movimiento. En la siguiente imagen se muestra lo descrito anteriormente:

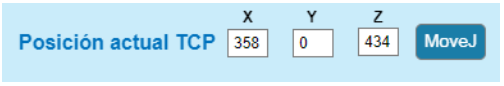


Ilustración 60. Casillas de coordenadas absolutas para la cinemática inversa

MoveJ desplaza la herramienta del robot desde la posición actual hasta la posición introducida en las casillas por el camino más corto, no tiene por qué ser una línea recta. Al tener como entrada [x, y, z], la orientación se mantiene igual que la inicial.

Además de poder introducir la coordenada deseada, tras utilizar otras partes de la interfaz para mover el robot, se muestra la posición donde ha quedado el TCP.

➢ POSICIÓN DE INICIO

El botón de inicio que se muestra en la figura 61 permite colocar al robot en la posición definida como *Inicio*, q = [0, 0, 0, 0, 0, 0].

Ilustración 61. Botón de posición de inicio

Realiza el movimiento mediante la función *MoveAbsJ*, es decir, el robot se mueve desde la posición actual hasta la posición articular introducida como *Inicio*.

Una gran ventaja de este botón es permitir la coordinación entre el robot de la estación simulada en Matlab y el robot real comunicado mediante una IP, ya que, de esta forma, parten ambos de la misma posición inicial.

➢ IMPORTAR – EXPORTAR ROBOT

Los botones que se muestran en la figura 62, permiten importar y exportar el objeto *Kin* del robot simulado en Matlab que se quiere mover.

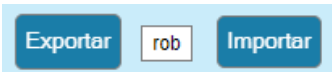


Ilustración 62. Botones de exportar - importar robot

La casilla central es editable, por lo que el usuario puede variar el nombre del robot dependiendo de cómo lo haya definido en su estación de trabajo.

➢ ELECCIÓN DEL ROBOT DE MATLAB (*SimRob*) O EL REAL

En la figura 63 se muestra el diseño de un grupo de botones excluyentes, es decir, si pulsas uno el otro no se mantiene pulsado, con el fin de que el usuario decida simular únicamente el robot de la estación de Matlab (*SimRob*), o el real y el de Matlab. Además, se ha añadido una casilla para introducir la IP del robot que se desee mover.

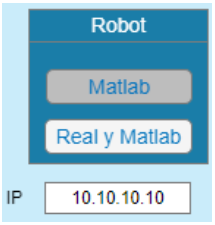


Ilustración 63. Botones para elegir el robot a mover

Cuando se pulsa el botón inferior, se establece la comunicación con PyNiryo con el fin de conectarse al ROS de Niryo Ned y ejecutar las órdenes dadas desde Matlab. Para verificar la conexión, una vez pulsado este botón, el color

de las letras debe aparecer verde, como se muestra en la imagen 64. Una vez establecida la conexión, el robot se autocalibra para desempeñar un correcto funcionamiento.

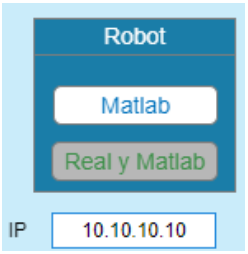


Ilustración 64. Verificación de la conexión con el robot real

Por el contrario, al pulsar de nuevo el botón superior, se verifica la desconexión del robot real cuando el color vuelve a ser azul, el del inicio de la pantalla.

- **ABRIR – CERRAR PINZA**

Estos botones, figura 65, solo están disponibles para el robot real, ya que la simulación de Matlab no es capaz de desempeñar esta tarea porque su pinza se mantiene fija.

Ilustración 65. Botones abrir - cerrar pinza

Permiten abrir y cerrar cualquiera de las 3 pinzas que incorpore el robot Niryo Ned, siendo también excluyentes a la hora de presionarlos.

CAPÍTULO 5. RESULTADOS

5.1. ESTUDIO DINÁMICO DEL COMPORTAMIENTO DEL SISTEMA

5.1.1. SEGUIMIENTO DE UNA TRAYECTORIA

Se van a realizar distintos ensayos dinámicos con la tercera estación que se ha diseñado, modelo dinámico con dos lazos, variando los controladores de ambos lazos y modificando la saturación de los pares de entrada. El robot debe realizar un movimiento definido por una entrada que varía las coordenadas articulares desde 0 a -30º cada una.

Se mostrarán dos diagramas, el primero hace referencia a los ángulos correspondientes a cada articulación, y el segundo a los pares que ejercen cada articulación en un periodo de tiempo.

La constante de proporcionalidad del lazo externo se denomina *Kq*, la del lazo interno *Kw* y la saturación de los actuadores *sat*.

- Ensayo 1: K_q = 100, K_w = 0, *sat* = 10000 Nm

Se comienza estudiando el caso más sencillo, solo interviene el controlador del lazo externo.

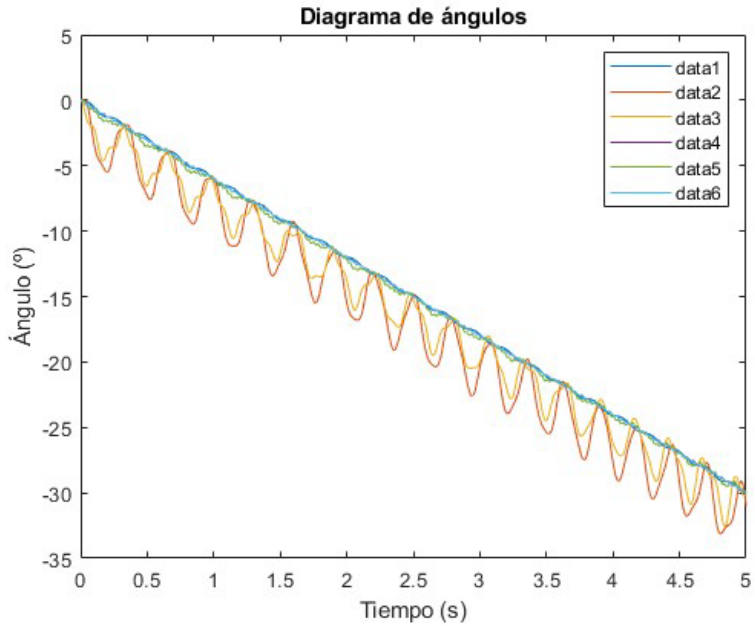


Ilustración 66. Diagrama de ángulos para Kq = 100, Kw = 0, sat = 10000 Nm

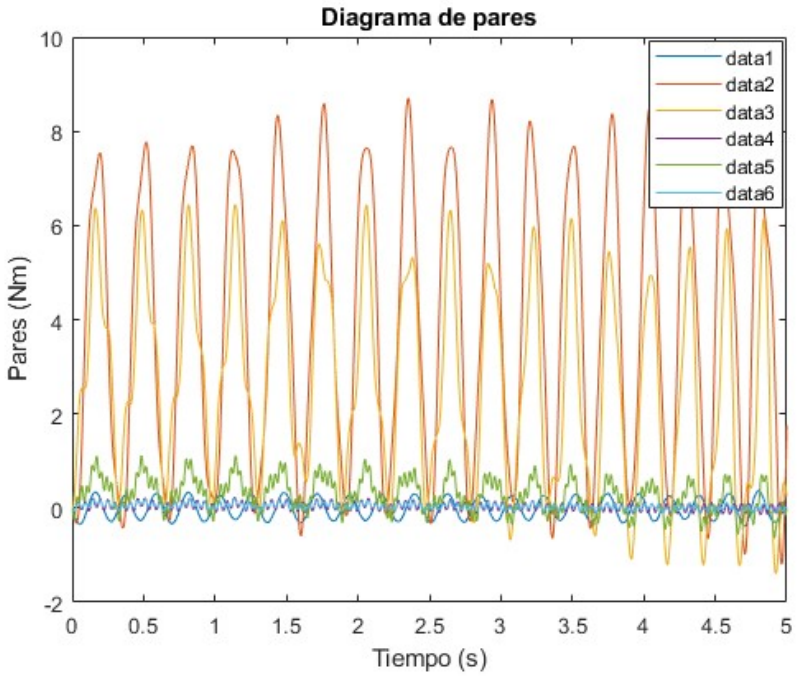


Ilustración 67. Diagrama de pares para Kq = 100, Kw = 0, sat = 10000 Nm

El robot no es capaz de seguir la trayectoria definida sin oscilaciones debido al bajo valor del controlador del lazo externo y la inexistencia del controlador en el lazo interno.

Se puede observar en el segundo diagrama como los actuadores que se encuentran en la posición 2 y 3 ejercen un mayor par para lograr llegar a la posición final.

- Ensayo 2: K_q = 10000, K_w = 0, *sat* = 10000 Nm

En este ensayo se ha incorporado una constante del lazo externo muy elevada para examinar el comportamiento del sistema.

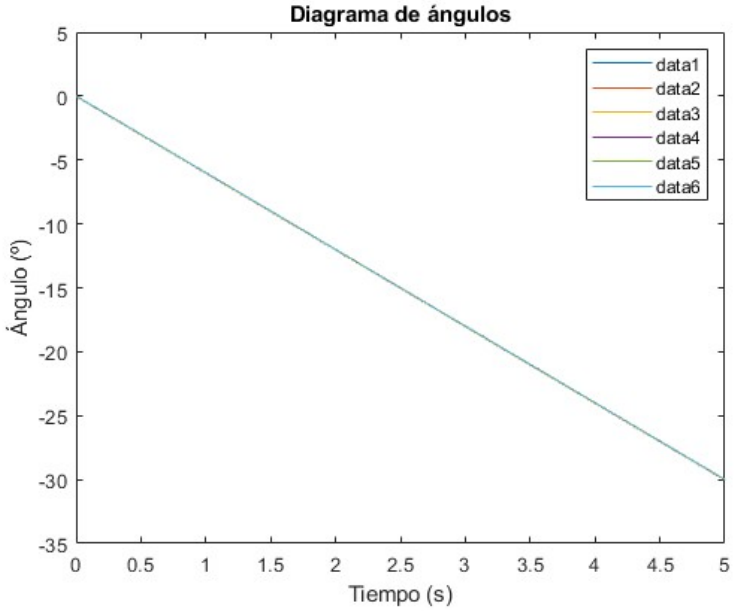


Ilustración 68. Diagrama de ángulos para Kq = 10000, Kw = 0, sat = 10000 Nm

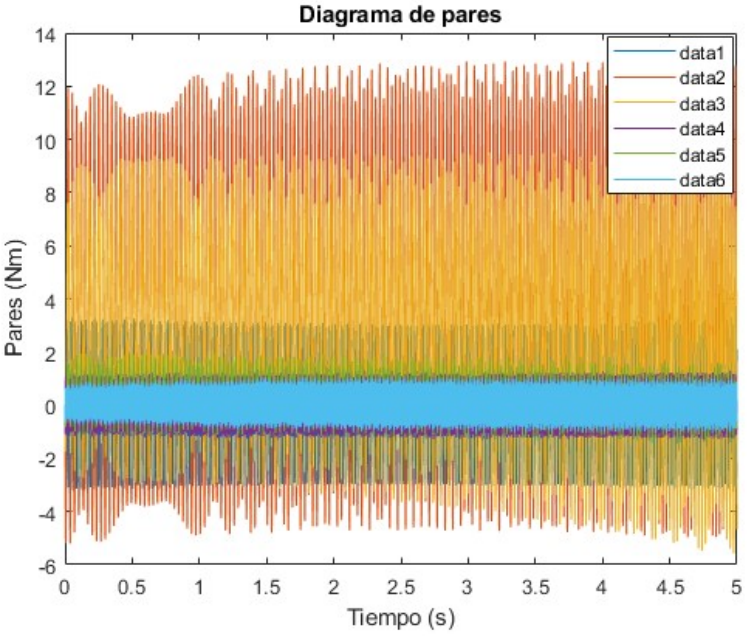


Ilustración 69. Diagrama de pares para Kq = 10000, Kw = 0, sat = 10000 Nm

Gracias al aumento de esta constante, el robot ha conseguido alcanzar las coordenadas articulares deseadas. Sin embargo, las oscilaciones en el diagrama de pares han aumentado considerablemente.

- Ensayo 3: K_q = 100, K_w = 100, *sat* = 10000 Nm

Se ha decidido aumentar la constante del controlador del lazo interno para comprobar que realmente elimina las oscilaciones del sistema que se ha visto en el ensayo 1.

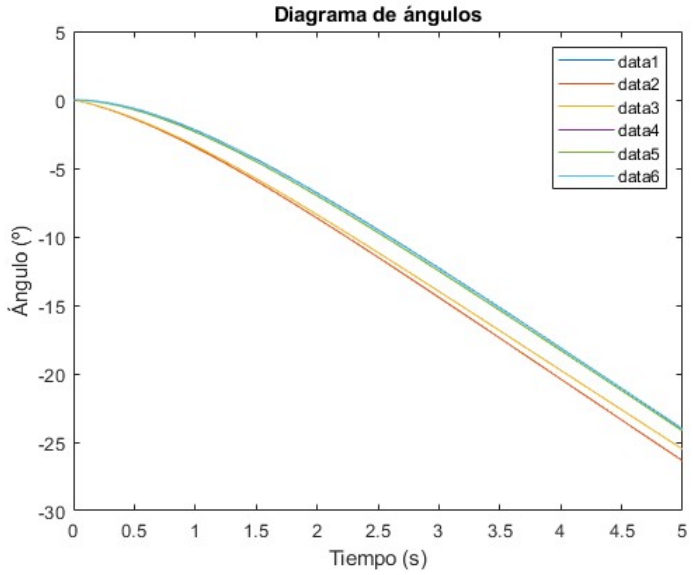


Ilustración 70. Diagrama de ángulos para Kq = 100, Kw = 100, sat = 10000 Nm

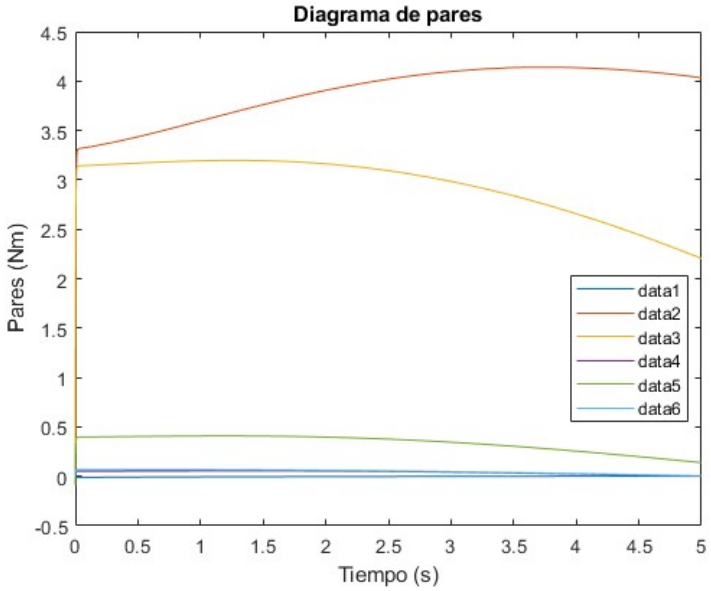


Ilustración 71. Diagrama de pares para Kq = 100, Kw = 100, sat = 10000 Nm

Según se esperaba, el controlador del lazo interno ha conseguido reducir las oscilaciones del sistema, sin embargo, ha desencadenado una posición final distinta a la deseada. También se ha visto como en el diagrama de pares ya no aparecen oscilaciones, y además, ha reducido el par que debe ejercer cada actuador.

- Ensayo 4: K_q = 100, K_w = 10000, *sat* = 10000 Nm

En este ensayo se observará lo que ocurre cuando se incorpora una constante demasiado elevada en el controlador del lazo interno.

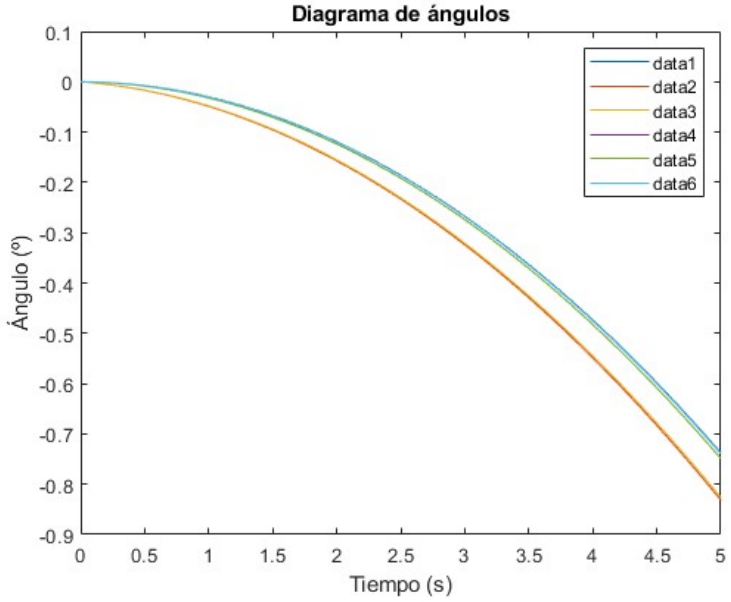


Ilustración 72. Diagrama de ángulos para Kq = 100, Kw = 10000, sat = 10000 Nm

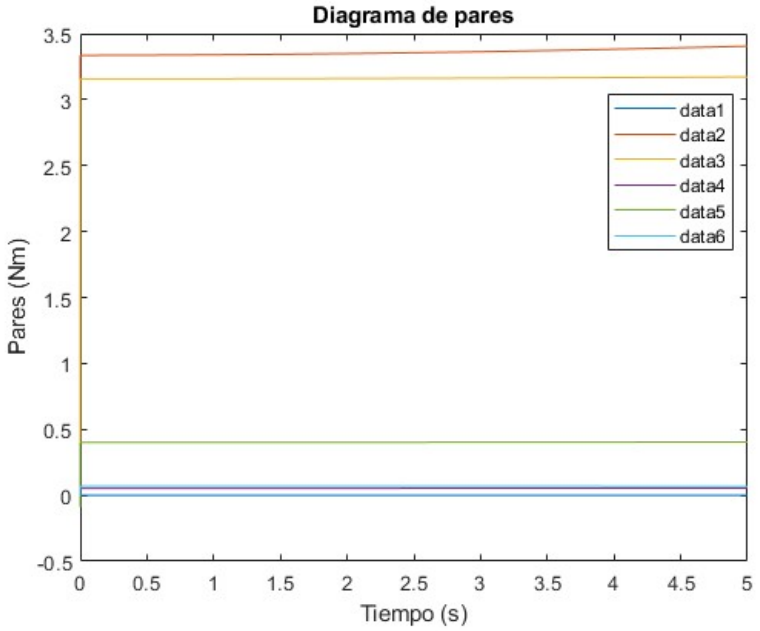


Ilustración 73. Diagrama de pares para Kq = 100, Kw = 10000, sat = 10000 Nm

De nuevo se observa como elimina las oscilaciones en ambos diagramas completamente, pero se queda muy lejos de alcanzar las coordenadas articulares deseadas.

- Ensayo 5: K_q = 1000, K_w = 10, *sat* = 4 Nm

En los ensayos anteriores se ha tenido un valor de la saturación muy elevado, por lo que no ha influido en el sistema. A continuación se va a comprobar la influencia de una saturación menor.

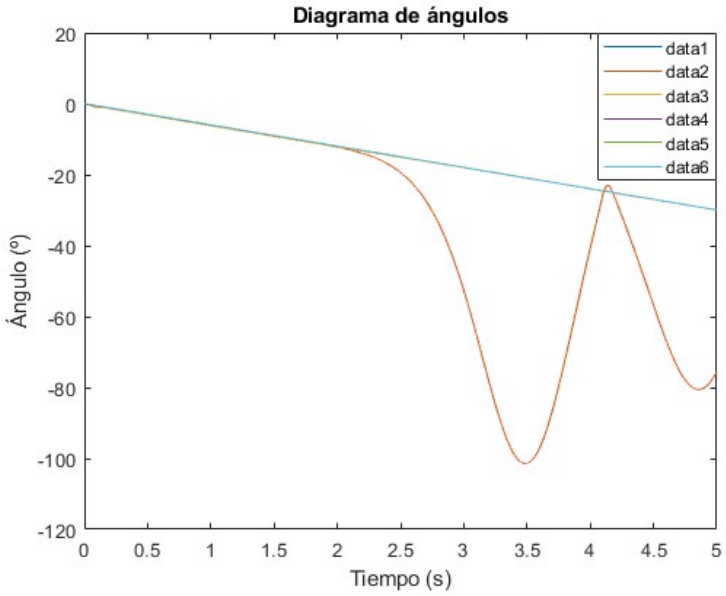


Ilustración 74. Diagrama de ángulos para Kq = 1000, Kw = 10, sat = 4 Nm

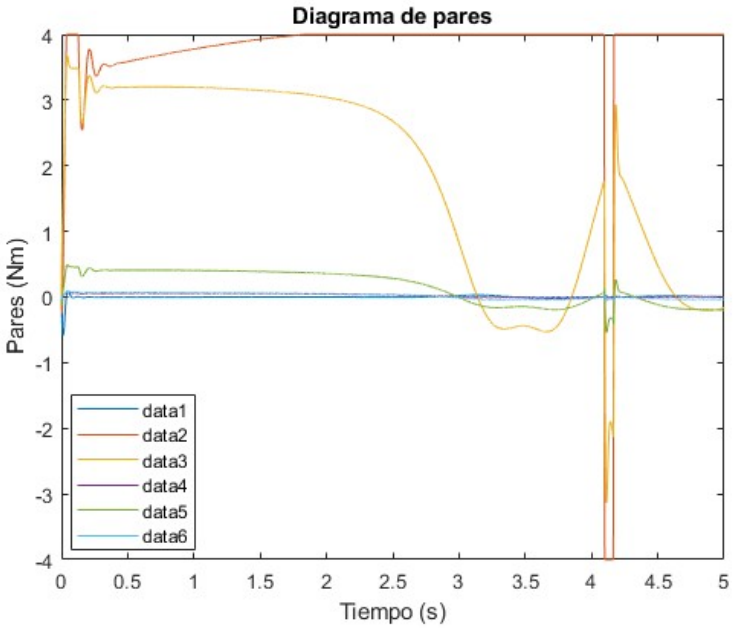


Ilustración 75. Diagrama de pares para Kq = 1000, Kw = 10, sat = 4 Nm

Los actuadores del robot se han visto limitados a la hora de ejercer el par suficiente para realizar la trayectoria deseada. El sistema necesita ejercer un valor de par específico pero la saturación no lo permite, por lo que no se

consigue la suficiente fuerza para elevar las articulaciones del robot y como consecuencia no realiza la trayectoria deseada.

5.1.2. ELEVACIÓN DE UNA CARGA

A partir de la estación que incorpora un modelo dinámico con dos lazos, se ha establecido una trayectoria, con la ayuda del objeto *Kin,* con la finalidad de que el robot coja una carga de masa *m*. Los parámetros usados que caracterizan al sistema son: K_q = 10000, K_w = 0, *sat* = 10000 Nm.

Se van a realizar dos ensayos, el primero con una carga soportable para el robot, y el segundo con una masa demasiado elevada, y así examinar su comportamiento.

Para ambos ensayos el robot comienza en la siguiente posición:

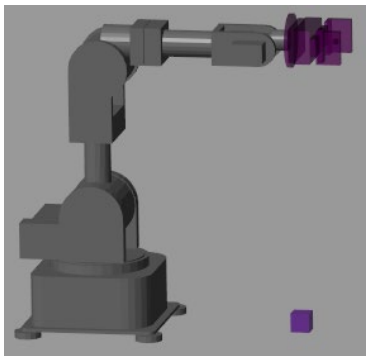


Ilustración 76. Posición inicial del ensayo

- **Ensayo 1: m = 10 kg**

El robot vuelve a la misma posición de inicio ya que tiene suficiente fuerza para elevar la carga.

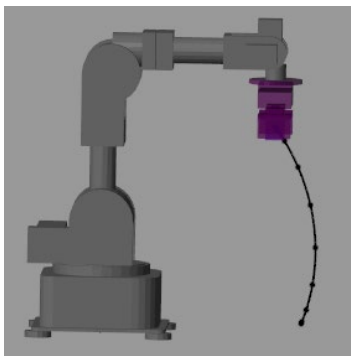


Ilustración 77. Posición final del ensayo con m = 10 kg

- **Ensayo 2: m = 1000 kg**

El robot no consigue volver a su posición de inicio debido a que la carga es demasiado elevada y los actuadores no tienen suficiente fuerza para levantarla.

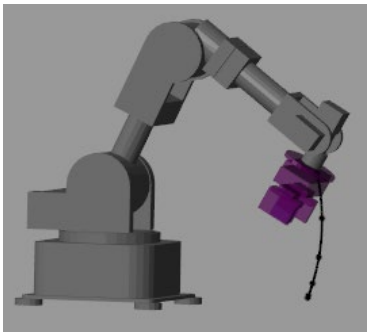


Ilustración 78. Posición final del ensayo con m = 1000 kg

5.2. ESTUDIO DE COMUNICACIONES DE LA APLICACIÓN

Se van a ejecutar distintas acciones con la aplicación diseñada para Niryo Ned y comprobar su funcionamiento mediante la herramienta de simulación *Rviz*. Esta herramienta nos permite verificar que la conexión mediante IP con un robot es correcta, por lo tanto, para el robot real también será correcta.

En primer lugar, se realizará la conexión con el robot simulado mediante la IP 127.0.0.1.

Antes de realizar la conexión, la lista de botones se encuentra así:

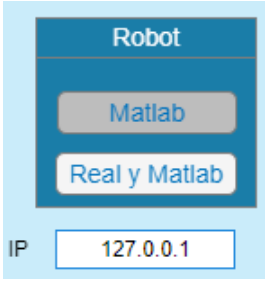


Ilustración 79. Selección de la simulación en Matlab

Tras pulsar el botón *Real y Matlab*, debe de aparecer en color verde para confirmar que la conexión ha sido correcta, como en la figura 80.

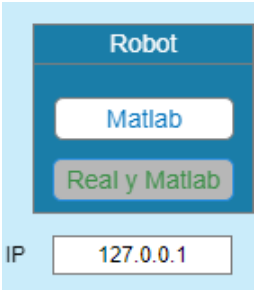


Ilustración 80. Selección de la simulación en Matlab y del robot real

Para comprobar el correcto funcionamiento de las cremalleras diseñadas para la cinemática directa, se van a introducir las coordenadas articulares correspondientes al vector q = [2,-50, 35, -24, 50] en grados. Además, también se puede comprobar cómo nos indica la posición actual que ha tomado el TCP.

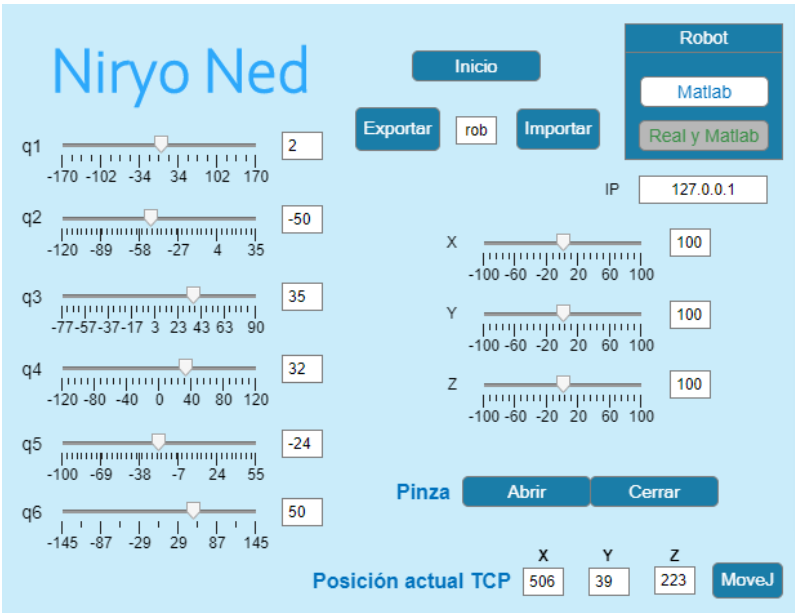


Ilustración 81. Comprobación de las cremalleras de cinemática directa

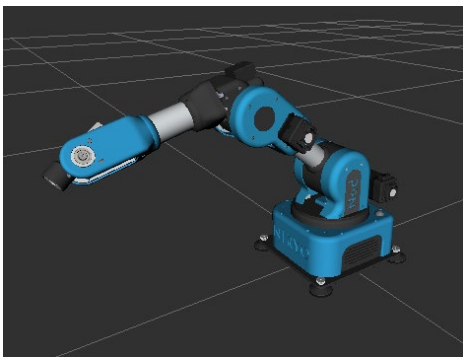


Ilustración 82. Niryo Ned en el entorno Rviz tras moverse por cinemática directa

Pulsando el botón *Inicio* se obtiene la posición que tiene como coordenadas articulares q = [0, 0, 0, 0, 0, 0]. Tras pulsarlo, las cremalleras y las casillas de la posición del TCP, nos indican las nuevas coordenadas.

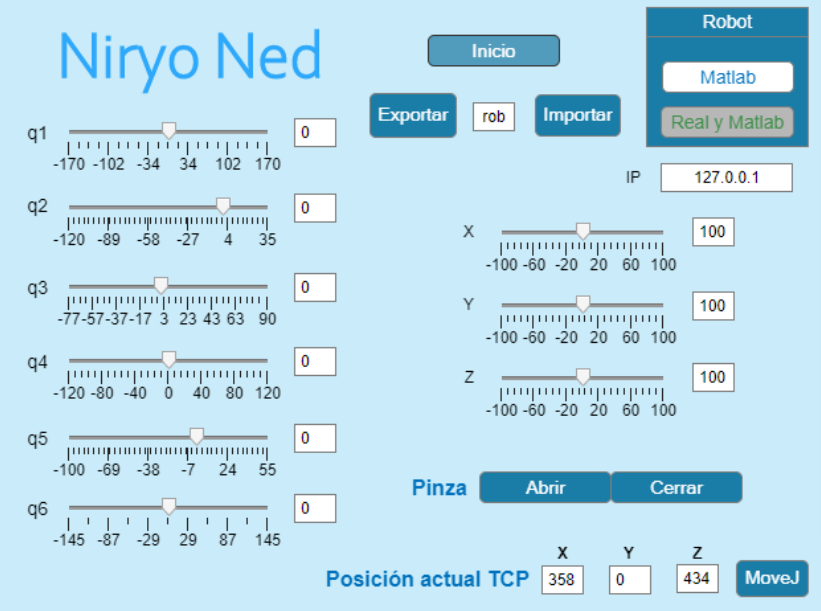


Ilustración 83. Comprobación del botón Inicio

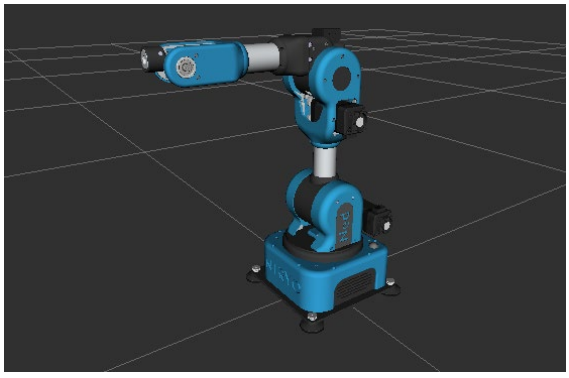


Ilustración 84. . Niryo Ned en el entorno Rviz en la posición de Inicio

Para comprobar el funcionamiento de las cremalleras de cinemática inversa con las coordenadas relativas al TCP y el cambio de límites de estas, se va a ordenar que se desplace al punto [200, 10, -100] relativo al TCP.

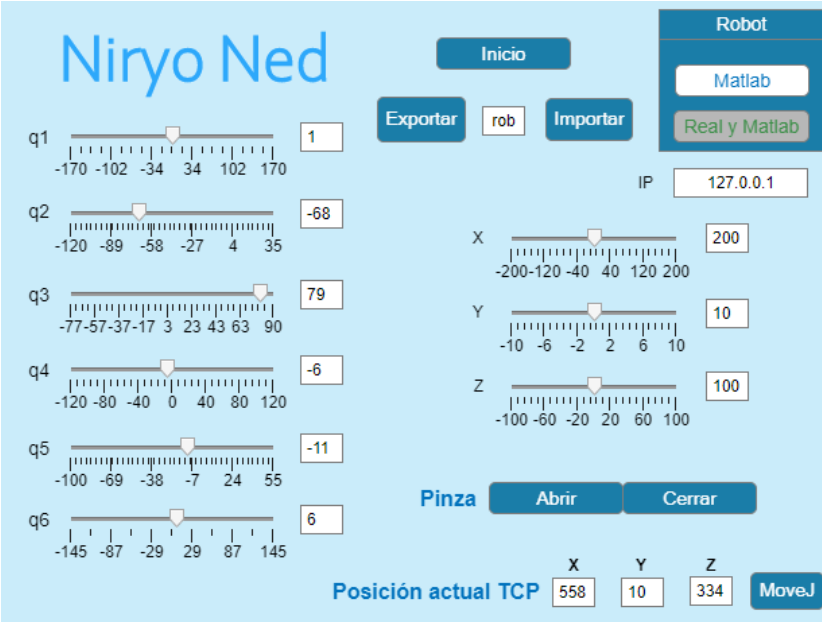


Ilustración 85. Comprobación de las cremalleras de cinemática inversa

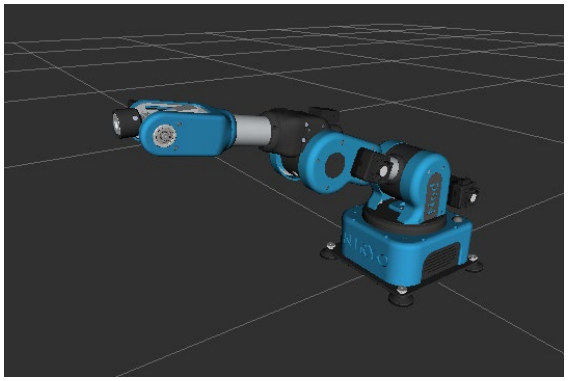


Ilustración 86. Niryo Ned en el entorno Rviz tras moverse por cinemática inversa

Por último, se va a verificar el funcionamiento del botón *MoveJ* introduciendo las coordenadas absolutas [300, 10, 300].

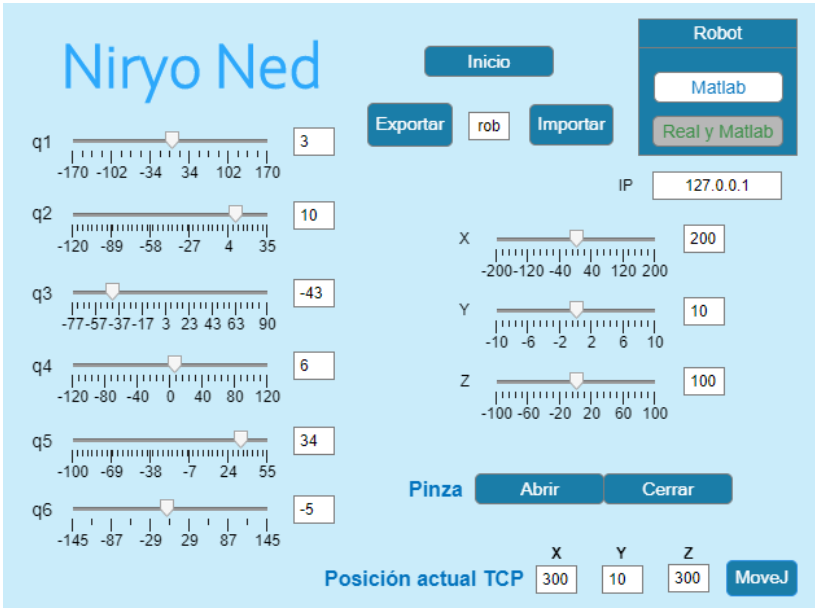


Ilustración 87. Comprobación del botón MoveJ

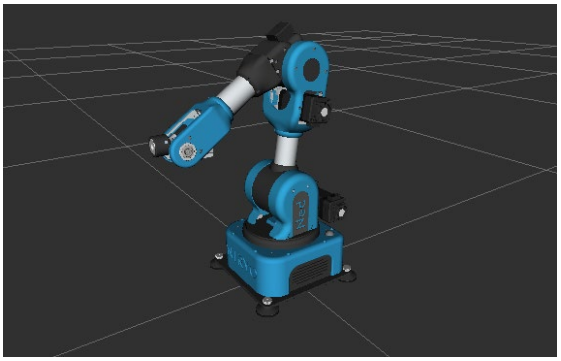


Ilustración 88. Niryo Ned en el entorno Rviz tras pulsar MoveJ

Los botones pertenecientes al accionamiento de la pinza han sido comprobados con el robot real y su funcionamiento es correcto.

CAPÍTULO 6. CONCLUSIONES Y LÍNEAS FUTURAS

6.1. CONCLUSIONES

Tras la realización del proyecto, se puede decir que se han cumplido los objetivos planteados inicialmente.

Aunque la instalación de todas las aplicaciones para la comunicación de Matlab con Niryo Ned es un proceso laborioso, es de gran utilidad didáctica conocer el esquema de comunicaciones completo. De esta forma, el alumno no se rige solo a programar los movimientos del robot, sino que es capaz de entender la forma en la que se comunica.

Gracias al diseño de la interfaz en App Desigener, el alumno podrá poner en práctica sus conocimientos teóricos de robótica, como el movimiento mediante cinemática directa e inversa. Además, un gran aporte de esta aplicación, es que el alumno podrá comprobar cómo el robot real realiza sus trayectorias programadas. En la clase solo podrá conectarse una persona al robot, pero una vez se desconecte, cualquier otro compañero podrá controlar los movimiento del robot.

Respecto al modelo dinámico estudiado, se puede concluir que al incluir el elemento de saturación de los actuadores para que el modelo fuese más realista, conlleva una limitación a la hora de la ejecución de los movimientos.

Por último, entre todas las funciones que desempeña la pantalla creada, se ha incluido el movimiento relativo al TCP para conseguir una mayor precisión a la hora de mover la herramienta de trabajo.

6.2. POSIBLES LÍNEAS DE TRABAJO FUTURAS

Se plantean los posibles trabajos futuros a partir de este proyecto:

- Estudiar el resto de comunicaciones de las que dispone Niryo Ned, como son Modbus TCP (master) y CAN (esclavo, compatible con los motores NiryoStepper). También sería interesante establecer el control mediante un Arduino.
- Modificar el diseño de la aplicación creada en App Designer incluyendo nuevas funciones, como *learning mode,* para modificar manualmente la posición del robot real y que la aplicación lo detecte y lo simule.

- Utilizar las distintas herramientas que incluye Niryo Ned, como el electroimán, la cámara de visión o la bomba aspiradora, para desempeñar una serie de tareas.

- Diseñar nuevas herramientas para la simulación en Matlab, como una pinza que simule el movimiento de abrir y cerrar, el electroimán o la bomba aspiradora.

ARCHIVO pyModule.m

```
function mod= pyModule(modulo)
   if nargin==0
      error('mod= pyModule(modulo)');
   end
   mod = py.importlib.import_module(modulo);
   %py.reload(mod); % versión python 2
   py.importlib.reload(mod); % versión python 3.x
   metodo= fieldnames(mod);
   for i= 1:length(metodo)
      eval("met= mod."+ metodo{i}+ ";");
      assignin('caller', metodo{i}, met);
   end
   % Borra el modulo compilado para no mezclar si se volviera a llamar
   eval(['! del __pycache__\',modulo,'.cpython-37.pyc']);
end
```

ARCHIVO *pyLines.m*

```
function mod= pyLines(varargin)
   fic= fopen('fic.py','w');
   for i= 1:nargin
      fprintf(fic, [varargin{i},'\n']);
   end
   fclose(fic);
   mod = py.importlib.import_module('fic');
   %py.reload(mod); % versión python 2
   py.importlib.reload(mod); % versión python 3.x
   metodo= fieldnames(mod);
   for i= 1:length(metodo)
      eval(['met= mod.', metodo{i}, ';']);
      assignin('caller', metodo{i}, met);
   end
end
```

ARCHIVO *Live_pyNiryo.mlx*

```
pyLines('from pyniryo import *')

robot_ip_address = "127.0.0.1";

robot = NiryoRobot(robot_ip_address)
```

```
Connected to server (127.0.0.1) on port: 40001

Disconnected from robot

robot =

  Python NiryoRobot with properties:

     analog_io_state: [1×0 py.list]
    digital_io_state: [1×8 py.list]
     hardware_status: [1×1 py.pyniryo.api.objects.HardwareStatusObject]
              joints: [1×6 py.list]
       learning_mode: 1
                pose: [1×1 py.pyniryo.api.objects.PoseObject]
                tool: [1×1 py.pyniryo.api.enums_communication.ToolID]

    Connected to server (127.0.0.1) on port: 40001
```

```
robot.calibrate_auto() %Calibra el robot

robot.move_joints(0.0, 0.0, 0.0, 0.0, 0.0, 0.0) %Mueve el robot
```

CÓDIGO DE LA APLICACIÓN DE APP DESIGNER

PROPIEDADES Y FUNCIONES

```
properties (Access = private)
        rob % Objeto Kin
        robot %real
    end


    methods (Access = private)

        function q = LeerJointSlider(app) %Pone en la variable q el valor de la
cremallera
                q= zeros(1,6);
                q(1)= app.q1Slider.Value;
                q(2)= app.q2Slider.Value;
                q(3)= app.q3Slider.Value;
                q(4)= app.q4Slider.Value;
                q(5)= app.q5Slider.Value;
                q(6)= app.q6Slider.Value;
                q= q*pi/180;
        end

        function EscribirJointSlider(app, q) %Escribe en la cremallera el valor
de q
                q= q*180/pi;
                app.q1Slider.Value= q(1);
                app.q2Slider.Value= q(2);
                app.q3Slider.Value= q(3);
                app.q4Slider.Value= q(4);
                app.q5Slider.Value= q(5);
                app.q6Slider.Value= q(6);
        end

        function q= LeerJointEdit(app) %Pone en la variable q el valor puesto en
la casilla qEdit
                q= zeros(1,6);
                q(1)= str2double(app.q1Edit.Value); %Pasa de texto a número
                q(2)= str2double(app.q2Edit.Value);
                q(3)= str2double(app.q3Edit.Value);
                q(4)= str2double(app.q4Edit.Value);
                q(5)= str2double(app.q5Edit.Value);
                q(6)= str2double(app.q6Edit.Value);
                q= q*pi/180;
        end

        function EscribirJointEdit(app, q) %Escribe en qEdit el valor de q
                q= q*180/pi;
                app.q1Edit.Value= sprintf('%.0f', q(1)); %Pasa de número a texto,
con 0 decimales
                app.q2Edit.Value= sprintf('%.0f', q(2));
                app.q3Edit.Value= sprintf('%.0f', q(3));
                app.q4Edit.Value= sprintf('%.0f', q(4));
                app.q5Edit.Value= sprintf('%.0f', q(5));
                app.q6Edit.Value= sprintf('%.0f', q(6));
        end

        function pose= LeerPose(app)  %Lee la posición absoluta de Tcp escrita
en Posxyz y la escribe en pose
```

```
                pose= zeros(1,3);
                pose(1)= str2double(app.PosX.Value); %Pasa de texto a número
                pose(2)= str2double(app.PosY.Value);
                pose(3)= str2double(app.PosZ.Value);
                pose(1:3)= pose(1:3)*1e-3;
        end

        function EscribirPose(app, pose) %Escribe en la casilla Posxyz la
posición del tcp (pose)
               pose(1:3)= pose(1:3)*1e3;
               app.PosX.Value= sprintf('%.0f', pose(1));
               app.PosY.Value= sprintf('%.0f', pose(2));
               app.PosZ.Value= sprintf('%.0f', pose(3));
        end

        function DibujarPunto(app) %Dibuja el punto de la posición del tcp
            pose= LeerPose(app);
            punto= repmat(pose,[3,1])+ randn(3,3)*1e-5;
            app.rob.Show([],punto)
        end

    end
```

FUNCIONES DE INICIALIZACIÓN (startupFcn)

```
             app.rob= rob; % robot Matlab
            app.robot = robot; %real

            q= app.rob.Joint;
            pose= app.rob.Pose;
            app.EscribirJointSlider(q);
            app.EscribirJointEdit(q);
            app.EscribirPose(pose);

            % Establecer los límites de las cremalleras
            app.q1Slider.Limits = [-170,170];
            app.q2Slider.Limits = [-120,35];
            app.q3Slider.Limits = [-77,90];
            app.q4Slider.Limits = [-120,120];
            app.q5Slider.Limits = [-100,55];
            app.q6Slider.Limits = [-145,145];

            app.MatlabButton.Value = 1; %Siempre se inicia con el botón Matlab
pulsado
```

CREMALLERA PARA EL MOVIEMIENTO POR CINEMÁTICA DIRECTA

```
            q= app.LeerJointSlider(); %Lee el número de la cremallera
            app.EscribirJointEdit(q); %Escribe el número en la casilla
            app.rob.MoveAbsJ(q); %Mueve el robot a la posición dada
            pose= app.rob.Pose; % Guarda en pos la posición del robot
            app.EscribirPose(pose); % Escribe la posición en casillas XYZ

            if app.RealyMatlabButton.Value == 1
               app.robot.move_joints(q); %Robot Real

            end
```

CASILLA EDITABLE JUNTO A LA CREMALLERA

```
             q= app.LeerJointEdit(); %Lee el número de la casilla
            app.EscribirJointSlider(q); %Mueve la cremallera al número dado
            app.rob.MoveAbsJ(q); %Mueve el robot a la posición dada
            pose= app.rob.Pose; % Guarda en pos la posición del robot
            app.EscribirPose(pose); % Escribe la posición en casillas XYZ

            if app.RealyMatlabButton.Value == 1
               app.robot.move_joints(q); %Robot Real
            end
```

BOTÓN DE INICIO

```
             q= [0,0,0,0,0,0]*pi/180; %Posición de inicio
            app.rob.MoveAbsJ(q); %Mueve el robot a la posición inicial
            pose= app.rob.Pose; %Escribe la posición en la variable pose
            app.EscribirJointSlider(q); %Ajusta la cremallera
            app.EscribirJointEdit(q); %Escribe en la casilla lateral de la
cremallera
            app.EscribirPose(pose); %Escribe la posición absoluta del tcp en XYZ

            if app.RealyMatlabButton.Value == 1
               app.robot.move_joints(q); %Robot Real
            end
```

BOTÓN DE IMPORTAR

```
            app.rob= evalin('base', app.robEdit.Value); %Importa el robot
introducido en la casilla rob
            pose= app.rob.Pose;
            app.EscribirPose(pose); %Escribe la posición del robot importado
            app.DibujarPunto;
```

BOTÓN DE EXPORTAR

```
            assignin('base', app.robEdit.Value, app.rob); %Exporta el robot
```

CREMALLERA PARA CINEMÁTICA INVERSA EN COORDENADAS RELATIVAS AL TCP

Se va a mostrar como ejemplo la cremallera correspondiente al eje X, pero sería similar para el resto de ejes.

```
            pose=  app.LeerPose(); %Leer la posición actual del tcp
            dpose= (app.XSlider.Value)*1e-3; %Posición relativa escrita en este
eje en la cremallera, sirve tanto si es positiva como negativa
            pose(1)= pose(1)+ dpose; %Posición relativa pedida
            app.EscribirPose(pose); %Ecribe la posición en el vector pose
            app.DibujarPunto; %Dibuja al punto al que va a ir
            app.rob.MoveJ(app.LeerPose); %Se mueve a la posición dada
            app.XSlider.Value = 0; %Vuelve a poner en 0 la cremallera

            %Escribir en las cremalleras q la posición que tiene ahora el
            %robot tras haberlo movido relativamente
            q= app.rob.Joint;
            app.EscribirJointSlider(q);
            app.EscribirJointEdit(q);

            if app.RealyMatlabButton.Value == 1
               q= app.LeerJointSlider(); %Lee la posición del robot de Matlab
               app.robot.move_joints(q); %Mueve al robot real a la posición del
de Matlab
            end
```

CASILLA PARA LOS LÍMITES DE LA CREMALLERA

```
            limX = abs(str2double(app.LimX.Value));
            app.XSlider.Limits = [-limX, limX];
```

BOTÓN *MoveJ*

```
            app.rob.MoveJ(app.LeerPose);
            q= app.rob.Joint;
            app.EscribirJointSlider(q);
            app.EscribirJointEdit(q);

            if app.RealyMatlabButton.Value == 1
               q= app.LeerJointSlider(); %Lee la posición del robot de Matlab
               app.robot.move_joints(q); %Mueve al robot real a la posición del
de Matlab
            end
```

BOTÓN DE ABRIR PINZA

```
            if app.AbrirPinza.Value == 1
               app.CerrarPinza.Value == 0
               app.robot.update_tool(); %Detecta y configura la herramienta
               app.robot.release_with_tool(); %Suelta el objeto
            end
```

BOTÓN DE CERRAR PINZA

```
             if app.CerrarPinza.Value == 1
               app.AbrirPinza.Value == 0
               app.robot.update_tool(); %Detecta y configura la herramienta
               app.robot.grasp_with_tool(); %Agarra el objeto
            end
```

BOTONES PARA SELECCIONAR EL ROBOT

```
            if  app.RealyMatlabButton.Value == 1
                pyLines('from pyniryo import *');
                robot_ip_address = app.IPEditField.Value;
                app.robot = NiryoRobot(robot_ip_address);
                app.robot.calibrate_auto();
                app.RealyMatlabButton.FontColor = [0.23,0.56,0.28]; %Indica
conexión con el robot real
            end
            if app.MatlabButton.Value == 1 %Si se pulsa el botón de Matlab y
además está conectado el robot real
                if app.RealyMatlabButton.FontColor == [0.23,0.56,0.28]
                    app.robot.close_connection(); %desconectar el robot real
                    app.RealyMatlabButton.FontColor = [0.00,0.45,0.74]; %Cambiar
al color azul
                end
            end
```

ABREVIATURAS

CAD	*Computer Aided Design*
CAN	*Controller Area Network*
D-H	Denavit - Hartenberg
GDL	Grado De Libertad
GUI	*Graphical User Interface*
HMI	*Human-Machine Interface*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
OPC	*Open Protocol Communication*
PID	Controlador proporcional, integral y derivativo
PLC	*Programmable Logic Controller*
ROS	*Robot Operating System*
RPC	*Remote Procedure Call*
TCP (comunicación)	*Transmission Control Protocol*
TCP (robot)	*Tool Center Point*
URDF	*United Robotics Description Format*

BIBLIOGRAFÍA

[1] «Robots colaborativos» [En línea]. Available: https://k-robots.es/que-es-un-cobot-robot-colaborativo/. [Último acceso: Junio 2022].

[2] A. Barrientos, L. F. Peñín, C. Balaguer y R. Aracil, Fundamentos de robótica, Madrid: McGraw-Hill, 1997. ISBN: 84-481-0815-9.

[3] S. Arévalo Fernández, Aplicación de Simulink y Matlab para análisis cinemático y dinámico, control y comunicaciones de robots industriales, TFG Univ. Valladolid, 2021.

[4] N. Saeed Benjamin, D. Ph y E. P, Introduction to Robotics (2nd edition), Wiley, 2011. ISBN: 978-0-470-60446-5.

[5] S. J. M. Á. Mato, Simulación, control cinemático y dinámico de robots comerciales usando la herramienta de Matlab, "Robotic Toolbox", TFG Univ. Valladolid, 2014.

[6] J. A. Á. Herrero, Modelado de una célula robótica con fines educativos usando el programa "RobotStudio", TFG Univ. Valladolid, 2015.

[7] Á. G. de los Santos, Control remoto de una célula robotizada con fines educativos mediante tecnología WIFI, TFG Univ. Valladolid, 2016.

[8] C. Jiménez, Diseño de un sistema robótico educativo para jugar al ajedrez con robots industriales, TFM Univ. Valladolid, 2019.

[9] V. Lobo, Diseño, simulación y fabricación de una estación robotizada universitaria, TFM Univ. Valladolid, 2020.

[10] S. Arévalo y A. Herreros, «Aplicación de Matlab y Simulink para estaciones robóticas» de *XLII Jornadas de Automática: libro de actas.*, Castelló, 2021 , doi: 10.17979/spudc.9788497498043.625.

[11] Matlab (versión 2022a). Software Mathworks.

[12] Simulink (versión 2022a). Software Mathworks.

[13] P. Corke, Robotics, Vision & Control, Fundamental Algorithms in MATLAB, 2013.

[14] Robotic System Toolbox (versión 2022a). Software Mathworks.

[15] Simscape Multibody (versión 2022a). Software Mathworks.

[16] App Designer (versión 2022a). Software Mathworks.

[17] «ROS (Robot Operating System)» [En línea]. Available: http://wiki.ros.org/es. [Último acceso: Junio 2022].

[18] Ned User Manual (v3.1.1). Manual del operador: Niryo SAS, 2021.

[19] Ned ROS Documentation (v4.0.1). Manual del operador Niryo SAS, 2022.

[20] L. M. Crespo Martínez y F. A. Candelas Herías, Introducción a TCP/IP. Sistemas de transporte de datos, Alicante, 1998. ISB: 84-7908-435-9.

[21] Niryo Studio User Manual (v4.0.1). Manual del operador: Niryo SAS, 2022.