Universidad de la República
Facultad de Ingeniería

Control MPC basado en PINCs para manipulador articulado

Memoria de proyecto presentada a la Facultad de Ingeniería de la Universidad de la República por

Rodrigo Baliosian, Facundo Gil, Marcos Ibarburu

en cumplimiento parcial de los requerimientos para la finalización de la carrera de Ingeniería Eléctrica.

Tutor Pablo Monzón ..... Universidad de la República

Tribunal

Facundo Benavides ..... Universidad de la República
Christian Díaz ..... Universidad de la República
Álvaro Giusto ..... Universidad de la República

Montevideo
jueves 13 noviembre, 2025

Control MPC basado en PINCs para manipulador articulado, Rodrigo Baliosian, Facundo Gil, Marcos Ibarburu.

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.2).
Contiene un total de 150 páginas.
Compilada el jueves 13 noviembre, 2025.
http://iie.fing.edu.uy/

Agradecimientos

Rodrigo Baliosian: Me gustaría expresar mi agradecimiento, en primer lugar, a nuestro tutor y Jefe de Departamento, Pablo Monzón, quien nos acercó la idea del proyecto y fue un pilar fundamental tanto en su desarrollo como en la construcción del perfil de Control del cual me gradúo. También a mis compañeros de tesis, Facundo y Marcos, por este año de trabajo compartido.

Quiero agradecer a Karina García, Tatiana Baliosian, Gabriel Torres, Facundo Torres y al resto de mi familia por su apoyo y confianza en mí durante toda la carrera.

A mis amigos, que me acompañaron en cada etapa de este camino, especialmente a Nahuel Bebelaqua y Romina Álvarez, quienes han sido parte constante en estos cinco años y medio de estudio.

Deseo también destacar el apoyo de todo el equipo de Inteka, quienes me dieron la oportunidad de integrarme y desarrollarme en el ámbito del Control y la Automatización Industrial, y que siempre facilitaron compatibilizar la vida laboral con la de estudiante.

Y finalmente, a la UdelaR, por brindarme la oportunidad de cursar esta increíble carrera que me permitió crecer profesionalmente y construir nuevos vínculos y amistades.

Facundo Gil: Quisiera expresar mi agradecimiento a nuestro tutor, Pablo Monzón, por el apoyo brindado durante el proyecto y por la interesante propuesta que nos confió. A la UdelaR, por hacer posible mi camino dentro de la Facultad, y a mis padres y amigos, por el apoyo constante a lo largo de todos los años de carrera.

También agradezco a mis compañeros de tesis por la disposición y buena actitud con que se llevó adelante este proyecto que, aunque desafiante, resultó muy gratificante y revelador.

Marcos Ibarburu: Primeramente agradecer a nuestro tutor de proyecto, Pablo Monzón, por la propuesta y el siguimiento continuo de este trabajo. También a mis compañeros, Facundo y Rodrigo, por la disposición y el gran trabajo que desempeñaron durante todo el año que abarcó la tésis.

Agradezco también a la UdelaR por permitirme estudiar esta carrera y a los numerosos docentes, tutores y compañeros de diferentes cursos, por los aprendizajes obtenidos durante todos estos años, que contribuyeron tanto a mi crecimiento personal como a mi formación profesional.

Por último, agradecer enormemente a familia y amigos, en especial a mi padre Wilson, quien fue en todo aspecto, un pilar fundamental durante toda mi carrera, y a Igor por su compañía durante muchas de las largas jornadas de estudio.

Esta página ha sido intencionalmente dejada en blanco.

Resumen

Este proyecto se centra en el desarrollo de un sistema de control para manipuladores robóticos, combinando técnicas de *Control Predictivo por Modelo* (MPC) con *Redes Neuronales Informadas por Física para Control* (PINCs), introducidas en 2024. El objetivo principal es aprovechar el conocimiento físico del sistema dinámico para mejorar la eficiencia y la robustez del control, reduciendo la dependencia de modelos analíticos exactos y mejorando la capacidad de generalización frente a perturbaciones o incertidumbres.

Se implementa un esquema de control MPC completo en *ROS 2*, integrando la comunicación en tiempo real con la simulación del robot y el cálculo de trayectorias articulares. Paralelamente, se entrena una PINC capaz de aproximar la dinámica del manipulador a partir de las ecuaciones físicas del sistema y de la acción de control impuesta, incorporando restricciones de consistencia dinámica en el entrenamiento. Esta red se integra luego dentro del esquema MPC, reemplazando al modelo analítico tradicional y permitiendo predecir la evolución del sistema bajo diferentes secuencias de control.

Como metodología de trabajo, se utilizan simulaciones basadas en el modelo del brazo robótico *OpenManipulator-X* dentro del entorno *Gazebo*, con el fin de validar el comportamiento del controlador y la precisión de la red en escenarios controlados. Se presentan pruebas comparativas entre controladores con modelo analítico y con modelo neuronal, evaluando desempeño.

Los resultados muestran que la incorporación de una red informada por la física dentro del MPC permite mantener el desempeño del control. Además, la integración en *ROS 2* proporciona una arquitectura modular y escalable, apta tanto para simulación como para futuras implementaciones físicas. En conjunto, el proyecto demuestra el potencial de las PINCs como modelos predictivos dentro de esquemas de control MPC, y sienta las bases para su aplicación en sistemas robóticos reales.

Esta página ha sido intencionalmente dejada en blanco.

Tabla de contenidos

Capítulo 1

Introducción

La presente investigación se enmarca en el desarrollo de la tesis de grado titulada *"Control MPC basado en PINCs para manipulador articulado"*, para la carrera de Ingeniería Eléctrica de Rodrigo Baliosian, Facundo Gil y Marcos Ibarburu.

Este trabajo aborda aspectos fundamentales del control predictivo aplicado a sistemas robóticos. En particular, se desarrolla un esquema de *Control Predictivo Basado en Modelo* (MPC) y se analizan las *Redes Neuronales Informada por Física* (PINN) y su extensión para incoporporar acciones de control (PINC) [2] y se utiliza dicho esquema como modelo dinámico del sistema.

A lo largo del proyecto se estudia la formulación matemática y computacional de las PINNs, su entrenamiento para la aproximación de la dinámica de un sistema, y su posterior integración dentro del esquema de control MPC.

Asimismo, se implementa la arquitectura completa del sistema en *ROS 2*, permitiendo la comunicación entre los nodos de simulación, control y predicción.

1.1. Introducción al proyecto

Para el desarrollo de este proyecto se trabajó sobre el entorno *ROS 2*, el cual permitió integrar los distintos módulos de control, simulación y comunicación entre nodos. Todo el trabajo desarrollado se implementó utilizando el lenguaje *Python*, haciendo uso extensivo de librerías de aprendizaje profundo como *PyTorch* y *TensorFlow* para el entrenamiento y validación de las redes neuronales informadas por física (PINNs).

El proyecto se estructuró en torno a dos ejes principales: el desarrollo del esquema de *Control Predictivo Basado en Modelo* (MPC) y la implementación de una *Red Neuronal Informada por la Física para Control* (PINC) orientada al control del sistema. El primero se encarga de optimizar las acciones de control bajo restricciones, mientras que el segundo actúa como modelo predictivo capaz de estimar la evolución del sistema sin depender de un modelo analítico explícito.

El sistema completo fue validado en un entorno de simulación sobre el manipulador robótico *OpenManipulator-X*, integrando los módulos de control y simulación mediante *ROS 2* y *Gazebo*. Todo el código fuente del proyecto se encuentra documentado en repositorios dedicados, que incluyen los módulos de entrenamiento de la PINC, el controlador MPC y los nodos de ROS 2 desarrollados.

Este documento constituye el componente final del trabajo, donde se presentan los fundamentos teóricos, las decisiones de diseño, la descripción de la arquitectura implementada, los resultados obtenidos y las conclusiones alcanzadas durante el desarrollo del proyecto.

1.1.1. Motivación

El *Control Predictivo Basado en Modelo* (MPC) es una estrategia de control muy potente. Su principal fortaleza radica en la capacidad de anticipar el comportamiento futuro de un sistema, optimizando las acciones de control a lo largo de un horizonte de predicción sujeto a restricciones físicas y operativas.

Sin embargo, la aplicación práctica del MPC depende de la disponibilidad de un modelo dinámico preciso y computacionalmente eficiente. En sistemas complejos (como los manipuladores robóticos), la resolución *on line* de las ecuaciones diferenciales que describen la dinámica puede implicar un costo computacional elevado, especialmente cuando se requiere operar en tiempo real o en entornos con recursos limitados.

En este contexto, las *Redes Neuronales Informadas por la Física* (PINNs) surgen como una alternativa prometedora. La correcta implementación de una PINN permite reducir significativamente el tiempo de cómputo requerido para predecir la evolución del sistema, haciendo viable la aplicación de esquemas MPC en sistemas cuya simulación o integración numérica resulta lenta o costosa. En consecuencia, la combinación de MPC y PINNs representa un paso hacia controladores predictivos más eficientes, escalables y generalizables, capaces de operar en tiempo real incluso en escenarios de alta complejidad dinámica.

1.1.2. Antecedentes

Los antecedentes con los que se contó para el desarrollo de este proyecto fueron sugeridos por el tutor. Estos trabajos y fuentes sirvieron como base teórica y metodológica para abordar las distintas etapas del proyecto, desde la comprensión del enfoque de redes neuronales informadas por la física para control (PINCs) hasta la implementación del esquema de control predictivo basado en modelo (MPC) dentro del entorno *ROS 2*.

El trabajo de Raissi et al. [24] constituye la base fundamental del enfoque de *Physics-Informed Neural Networks*, presentando la formulación general y la metodología de entrenamiento para resolver ecuaciones diferenciales ordinarias y parciales mediante redes neuronales profundas. Complementariamente, Karniadakis et al. [7] ofrece una revisión exhaustiva del campo del aprendizaje automático informado por física, destacando su potencial para la modelación de sistemas complejos con escasez de datos experimentales.

En relación con la aplicación de PINNs al control de sistemas dinámicos, el trabajo de Antonelo. [2] plantea una metodología para integrar redes informadas por física dentro de esquemas de control, destacando las ventajas de este enfoque frente a métodos puramente numéricos en términos de costo computacional y precisión.

Por otra parte, para la implementación del entorno de simulación y control se tomó como referencia la arquitectura del *Robot Operating System 2 (ROS 2)* [15], un entorno ampliamente utilizado en robótica moderna.

Finalmente, se consideró el manipulador robótico *OpenMANIPULATOR-X* [29] como caso de estudio, dada su compatibilidad con *ROS 2* y su documentación abierta, lo que permitió modelar y simular su dinámica mediante librerías como *Pinocchio* 5.3 y *gazebo_ros2_control* [3].

Estos antecedentes proporcionaron el marco teórico y tecnológico necesario para la correcta comprensión de los temas abordados y sirvieron de guía para el diseño y desarrollo del sistema propuesto

1.1.3. Objetivos

El objetivo general de este proyecto es abordar el uso de redes neuronales informadas por la física para control (PINCs), para su aplicación en un esquema de control. Esto implica el estudio de los fundamentos teóricos del control predictivo, la comprensión del funcionamiento de las PINCs, y su integración dentro de un esquema de *Control Predictivo Basado en Modelo*

(MPC). Asimismo, se busca adquirir habilidades en el uso de entornos de simulación y control robótico, siendo *ROS 2* el entorno de desarrollo elegido.

El producto del proyecto consiste en un entorno desarrollado en *ROS 2*, compuesto por múltiples nodos que se comunican entre sí. Estos nodos serán los encargados de:

- Procesar la retroalimentación del sistema (posiciones, velocidades y torques articulares).
- Aplicar el control predictivo (MPC) sobre la trayectoria del brazo robótico.
- Simular la dinámica del sistema y visualizar su comportamiento.

Para el modelado de la física del sistema se utilizará una *Red Neuronal Informada por la Física para Control* (PINC) previamente entrenada, capaz de predecir la evolución dinámica del manipulador dada una acción de control y reemplazar al modelo analítico dentro del bucle de control.

Con el fin de alcanzar el objetivo general, se establecieron los siguientes objetivos específicos:

- Estudiar los fundamentos teóricos y prácticos de las *Redes Neuronales Informadas por la Física* (PINNs).
- Analizar los mecanismos para incorporar la señal de control en una PINN orientada al control (*Physics-Informed Neural Controller*, PINC).
- Familiarizarse con el entorno *ROS 2* y sus herramientas asociadas (*Gazebo*, rviz, etc.).
- Definir el o los manipuladores robóticos con los que se trabajará, inicialmente a nivel de simulación, incluyendo la obtención de sus modelos URDF (*Unified Robot Description Format*) y su integración en ROS 2.
- Estudiar los mecanismos de realimentación necesarios para implementar controladores en lazo cerrado.
- Desarrollar al menos una estrategia de control predictivo (MPC) aplicada a un manipulador robótico, validando su funcionamiento con el modelo dinámico proporcionado por la PINN.
- Comparar el desempeño del esquema MPC utilizando modelos analíticos tradicionales frente al uso de modelos aprendidos mediante PINNs.
- Documentar el proceso de desarrollo, integración y validación del sistema de control, describiendo las herramientas empleadas, los resultados obtenidos y las conclusiones alcanzadas.
- Documentar el proceso de desarrollo y entrenamiento de una PINN

En conjunto, estos objetivos buscan integrar teoría, simulación y práctica en el diseño de un controlador predictivo avanzado asistido por aprendizaje profundo, aplicable a sistemas robóticos no lineales.

1.1.4. Alcance

El alcance de este proyecto comprende el estudio, análisis y comprensión de la bibliografía y los fundamentos teóricos necesarios para aplicar, a nivel de simulación, los conocimientos adquiridos sobre control predictivo y redes neuronales informadas por física (PINNs). El objetivo es desarrollar un esquema funcional que integre ambos enfoques dentro de un entorno de simulación, con la perspectiva de su futura implementación en un prototipo físico.

En particular, el trabajo abarca:

- El entendimiento del funcionamiento de las PINCs y su entrenamiento para la aproximación de la dinámica de sistemas físicos.
- La incorporación de la PINC dentro de una estrategia de control predictivo (MPC) aplicada a un manipulador robótico.
- La implementación y validación del sistema en un entorno de simulación basado en *ROS 2* y *Gazebo*.

No se contempla, en esta primera instancia, la experimentación en un sistema físico real ni el desarrollo completo de paquetes propios de *ROS 2*. El foco del trabajo está puesto en la validación conceptual y funcional del esquema propuesto, evaluando su comportamiento en simulaciones controladas y sentando las bases para futuras extensiones hacia implementaciones reales.

1.1.5. Supuestos

Para el desarrollo de este proyecto se establecen ciertos supuestos que permiten acotar el alcance y facilitar la implementación de las distintas etapas del trabajo.

El principal supuesto es que el modelado dinámico del brazo robótico sea lo suficientemente completo y preciso como para permitir el entrenamiento efectivo de la red neuronal informada por física. Esto implica que el modelo considerado debe representar de manera adecuada las relaciones entre posiciones, velocidades, aceleraciones y torques articulares.

Asimismo, se asume la compatibilidad total entre los módulos implementados en *ROS 2* y las herramientas de simulación utilizadas, particularmente en la comunicación entre los nodos encargados del control, la simulación y la predicción. Bajo este supuesto, se espera que la integración entre el entorno de simulación (*Gazebo*), las bibliotecas de modelado dinámico (*Pinocchio*) y los controladores desarrollados sea funcional y consistente.

Finalmente, se considera que los recursos computacionales disponibles son suficientes para entrenar y validar la red neuronal en tiempos razonables, sin requerir optimizaciones de hardware especializadas.

1.1.6. Validez de los supuestos

La validez de los supuestos planteados se sustenta en la solidez y coherencia del ecosistema de herramientas empleado. Tanto *ROS 2*, como *Gazebo* y *Pinocchio*, comparten un mismo marco de referencia basado en descripciones *URDF* del robot, lo que garantiza la consistencia entre el modelo utilizado para la simulación, el cálculo de la dinámica y la implementación del control.

Este entorno integrado proporciona un contexto robusto y ampliamente validado en la comunidad robótica, donde las herramientas se complementan de forma natural para el desarrollo, la simulación y la validación de manipuladores. Por lo tanto los supuestos realizados sobre la compatibilidad, precisión del modelado y capacidad de integración resultan razonables dentro del marco adoptado.

1.2. Introducción al documento

El propósito de este documento es ofrecer al lector una visión general del desarrollo del proyecto de grado titulado **"Control MPC basado en PINCs para manipulador articulado"**. En él se abordan los fundamentos teóricos, el modelado matemático, la implementación práctica y la validación de un esquema de control predictivo aplicado a un manipulador robótico, integrando técnicas de aprendizaje profundo informadas por la física dentro del entorno *ROS 2*.

La estructura del documento se organiza de la siguiente manera:

- **Capítulo 1 — Introducción:** Presenta el contexto del proyecto, la motivación que lo impulsa, los antecedentes teóricos, los objetivos generales y específicos, así como los alcances y supuestos de trabajo.
- **Capítulo 2 — Redes Neuronales Informadas por Física:** Describe en detalle los fundamentos teóricos de las *Physics-Informed Neural Networks* (PINNs), su formulación matemática y otros detalles sobre su implementación. Se incluye además una implementación práctica aplicada al oscilador de Van der Pol. Esta tarea permitió al equipo familiarizarse con los aspectos teóricos y prácticos de las PINNs y su extensión al control (PINCs) en el contexto de un sistema relativamente pequeño y muy conocido.
- **Capítulo 3 — Control Predictivo por Modelo:** Expone los principios del *Model Predictive Control* (MPC), su formulación general, función de costo, restricciones y métodos de optimización. Luego se presenta la integración del MPC con redes informadas por física para control (PINCs) y su validación mediante el caso de estudio del oscilador de Van der Pol.
- **Capítulo 4 — Entorno de simulación ROS 2:** Detalla el diseño de la arquitectura del sistema de control dentro del ambiente *ROS 2*. Se describen los nodos desarrollados (de trayectoria, control, y gestión), los servicios implementados, y la integración con el simulador *Gazebo*. Se analizan también los resultados de simulación y las limitaciones encontradas.
- **Capítulo 5 — Brazo robótico:** Presenta los conceptos teóricos necesarios para comprender el funcionamiento de manipuladores robóticos, incluyendo cinemática, dinámica y modelado. Se describe el robot utilizado (*OpenManipulator-X*), su modelado en *Pinocchio*, y la implementación de la PINN específica para su dinámica. Además, se detalla la integración completa del sistema, comparando la predicción mediante *Pinocchio* y mediante la PINC.
- **Capítulo 6 — Conclusiones:** Resume los resultados obtenidos y las conclusiones generales y técnicas del proyecto. Finalmente, se presentan posibles líneas de trabajo futuro orientadas a la extensión del esquema propuesto a implementaciones físicas reales y a la optimización del desempeño del modelo neuronal.

El documento intenta mantener una estructura progresiva que permita al lector comprender la evolución del proyecto desde los fundamentos teóricos hasta la validación final del sistema propuesto.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 2

Redes Neuronales Informadas por Física

En este capítulo se desarrolla el marco teórico y práctico de *Physical-informed neural networks* (PINNs). En primer lugar, se presentan los fundamentos conceptuales, describiendo su formulación general, la función de costo, elección de arquitectura, hiperparámetros y otras consideraciones, así como también sus aplicaciones, extensiones y limitaciones. Posteriormente, se presenta la aplicación de estos conceptos al oscilador de Van der Pol.

2.1. Introducción

Una tendencia común y actual en diversos dominios científicos es que la capacidad para recolectar y generar datos observacionales supera ampliamente la capacidad para asimilarlos de manera coherente, y mucho menos comprenderlos en profundidad. A pesar de su enorme potencial empírico, la mayoría de los enfoques actuales de aprendizaje automático (ML) aún no logran extraer información interpretable ni conocimiento significativo a partir de esta avalancha de datos.

Además, los modelos puramente basados en datos pueden ajustarse muy bien a las observaciones disponibles, pero sus predicciones pueden ser físicamente inconsistentes o inverosímiles. Esto se debe a fenómenos como la extrapolación fuera del dominio de entrenamiento o a sesgos presentes en los datos observacionales, lo que puede llevar a un bajo rendimiento en tareas de generalización. Por tanto, surge la necesidad de integrar las leyes físicas fundamentales y el conocimiento del dominio, mediante el "aprendizaje informado por física", es decir, enseñando a los modelos de ML las reglas físicas que gobiernan los fenómenos.

2.1.1. Motivación

La motivación principal detrás del desarrollo de las redes neuronales informadas por la física (PINNs) radica en que los conocimientos o restricciones físicas pueden proporcionar modelos más interpretables, robustos frente a datos imperfectos (como valores faltantes, ruido o datos atípicos), y capaces de generar predicciones precisas y físicamente consistentes incluso en tareas de extrapolación o generalización.

Este enfoque permite incorporar conocimientos previos,provenientes de nuestra comprensión empírica, observacional, física o matemática del mundo, como restricciones teóricas sólidas (*priors informativos*) que actúan como inductores de sesgos útiles, además de las restricciones impuestas por los datos. Un ejemplo destacado son las PINNs, las cuales integran de forma simultánea datos y operadores matemáticos abstractos, incluyendo ecuaciones en derivadas parciales (PDEs), incluso en presencia de fenómenos físicos desconocidos.

A pesar de que existen numerosos repositorios de datos públicos, el volumen de datos experimentales realmente útiles para sistemas físicos complejos es limitado. El enfoque adecuado para

modelar predictivamente estos sistemas depende fuertemente de la cantidad de datos disponibles y de la complejidad del sistema en cuestión. En un extremo del espectro, se encuentra el paradigma clásico donde se asume que los únicos datos disponibles son las condiciones iniciales y de borde, mientras que las PDEs que gobiernan el sistema y sus parámetros se conocen con precisión. En el otro extremo, se dispone de abundantes datos (por ejemplo, series temporales), pero no se conoce la ley física subyacente que rige el sistema.

Para la mayoría de las aplicaciones reales, el caso más representativo es un escenario intermedio, en el cual se conoce parcialmente la física (por ejemplo, las leyes de conservación), pero se desconoce la relación constitutiva o funcional exacta. Sin embargo, se dispone de mediciones dispersas de ciertas variables de estado, las cuales pueden utilizarse para inferir parámetros o incluso para descubrir términos funcionales faltantes en las ecuaciones, al tiempo que se reconstruye la solución del sistema. Este caso mixto representa una generalización de los extremos anteriores y suele ser el más complejo.

De esta manera, las PINNs incorporan el conocimiento de las leyes físicas directamente en la función de pérdida, actuando así como un regularizador que limita el espacio de las soluciones admisibles e incrementa la capacidad de generalización del modelo. Esto permite aprovechar conocimiento previo sobre la dinámica de un sistema físico en el proceso de entrenamiento. [7]

2.2. Fundamentos de PINNs

2.2.1. Formulación general

Dada una ecuación diferencial general de la forma:

$$\mathcal{N}[y](\mathbf{x}, t) = f(\mathbf{x}, t), \quad \mathbf{x} \in \Omega, \quad t \in [0, T] \tag{2.1}$$

donde $\mathcal{N}$ es un operador diferencial (que puede ser lineal o no lineal), $y(\mathbf{x}, t)$ es la solución buscada y $f(\mathbf{x}, t)$ representa una fuente externa. Las condiciones de frontera y de contorno se especifican como:

$$\mathcal{B}[y](\mathbf{x}, t) = 0, \quad \mathbf{x} \in \partial\Omega, \quad t = 0 \tag{2.2}$$

donde $\mathcal{B}$ es un operador de frontera.

En el marco de las PINNs, una red neuronal $y_\theta(\mathbf{x}, t)$ parametrizada por θ se utiliza como aproximación de la solución. Para asegurar que esta red respeta las ecuaciones físicas, se define una función de pérdida compuesta por tres términos principales:

- **Pérdida en los datos observacionales (si existen)**:

$$\mathcal{L}_{\text{data}} = \frac{1}{N_d} \sum_{i=1}^{N_d} \left| y_\theta(\mathbf{x}_i, t_i) - y_i^{\text{obs}} \right|^2 \tag{2.3}$$

- **Pérdida en la ecuación diferencial**:

$$\mathcal{L}_{\text{physics}} = \frac{1}{N_r} \sum_{i=1}^{N_r} \left| \mathcal{N}[y_\theta](\mathbf{x}_i, t_i) - f(\mathbf{x}_i, t_i) \right|^2 \tag{2.4}$$

  donde $f(\mathbf{x}_i, t_i)$ representa el valor teórico esperado del operador diferencial en el punto $(\mathbf{x}_i, t_i)$ del dominio espacio-temporal, y a dichos puntos se los denomina *puntos de colocación*. Estos puntos se distribuyen en el dominio de interés y constituyen las ubicaciones en las que se evalúa el residuo de la ecuación diferencial. En el contexto de las PINNs, los

puntos de colocación permiten imponer el cumplimiento de la dinámica física subyacente como parte de la función de pérdida.

Para calcular los términos de la función de pérdida asociados a la ecuación diferencial, es necesario obtener derivadas temporales de las predicciones de la red neuronal. Estas derivadas corresponden a aquellas involucradas en el operador diferencial $\mathcal{N}[y_\theta](\mathbf{x}_i, t_i)$, que representa la dinámica del sistema. Esto requiere que la salida de la red sea diferenciable respecto a sus entradas y parámetros, propiedad que se garantiza por la estructura diferenciable de la red y las funciones de activación utilizadas.

En este trabajo, dichas derivadas se obtienen automáticamente mediante el sistema de diferenciación automática de `TensorFlow`, utilizando la función `autograd.grad`. Esta herramienta no define las derivadas en sí mismas, sino que proporciona un mecanismo eficiente y preciso para calcularlas, construyendo el grafo computacional de la red y aplicando la regla de la cadena sobre las operaciones involucradas.

En el contexto de PINNs, esta capacidad es clave, ya que permite evaluar los residuos de la ecuación diferencial en puntos del dominio continuo sin necesidad de una discretización explícita del mismo.

- **Pérdida en las condiciones de frontera**:

$$\mathcal{L}_{\text{boundary}} = \frac{1}{N_b} \sum_{i=1}^{N_b} |\mathcal{B}[y_\theta](\mathbf{x}_i, t_i)|^2 \tag{2.5}$$

Finalmente, la función de pérdida total se define como:

$$\mathcal{L}(\theta) = \lambda_d \mathcal{L}_{\text{data}} + \lambda_r \mathcal{L}_{\text{physics}} + \lambda_b \mathcal{L}_{\text{boundary}} \tag{2.6}$$

donde los coeficientes $\lambda_d, \lambda_r, \lambda_b$ ponderan la importancia relativa de cada término.
El algoritmo 1 presenta otra forma de visualizar el rol de estos nuevos términos de pérdidas.

Algorithm 1 Algoritmo general de entrenamiento de una PINN

Entradas: condiciones iniciales (x_0, y_0), solución verdadera (si se usa), tiempo T, pesos de pérdida w_d, w_{col}, w_{bc}, número de épocas N_{ep}, número de puntos de colocación N_{col}, número de puntos de datos N_{data} (si existen), learning rate, entre otros.
Inicializar: pesos de la red neuronal, ejemplo inicialización Xavier
Definir optimizador (por ejemplo, Adam) y scheduler de tasa de aprendizaje (opcional).
for *epoch* $= 1$ **hasta** N_{ep} **do**
    Calcular pérdida de datos: $L_{data} \leftarrow \text{MSE}(\text{PINN}(t, x_0, y_0), \text{datos})$
    Hallar derivadas necesarias y calcular pérdida física (ODE): $L_{ODE} \leftarrow \text{MSE}\left(\frac{dx}{dt} + f(x, y)\right)$
    Calcular pérdida de condiciones iniciales: $L_{BC} \leftarrow \text{MSE}(\text{PINN}(t = 0), [x_0, y_0])$
    Combinar pérdidas: $L_{total} \leftarrow w_d \cdot L_{data} + w_{col} \cdot L_{ODE} + w_{bc} \cdot L_{BC}$
    Retropropagar: $\nabla_\theta L_{total}$
    Actualizar pesos de la red
    (Opcional) Actualizar tasa de aprendizaje con scheduler
end for

2.3. Escalado de variables

Uno de los aspectos fundamentales al entrenar redes neuronales, y en particular PINNs, es la normalización de los datos de entrada. Esta etapa previa al entrenamiento cumple un rol clave en la estabilidad numérica, la eficiencia del entrenamiento y la capacidad de generalización del modelo.

Durante el entrenamiento de una red neuronal, los gradientes calculados mediante backpropagation son sensibles a las escalas relativas de las variables de entrada y salida. Si los datos presentan magnitudes muy dispares (por ejemplo, torques en el orden de 10^{-2} y posiciones articulares en el orden de unidades), los gradientes asociados a cada componente tendrán órdenes de magnitud distintos. Esto provoca que la optimización se vuelva inestable o que algunas variables dominen el proceso de aprendizaje, respecto de otras. Además, cuando se aplican penalizaciones físicas (como el residual de la ecuación diferencial), es importante que los distintos términos involucrados se encuentren en escalas comparables, ya que de lo contrario será difícil equilibrar adecuadamente los pesos relativos de cada término en la función de pérdida. [35] Los beneficios de normalizar los datos son:

- Aceleración del entrenamiento: al reducir la disparidad en la magnitud de los gradientes, el optimizador puede avanzar más eficientemente en el espacio de parámetros.
- Mejor condicionamiento numérico: evita problemas de exploding o vanishing gradients, particularmente relevantes en redes profundas.
- Estabilidad en la combinación de pérdidas: permite que los términos de datos, condiciones de contorno y ecuación diferencial sean balanceados sin requerir valores extremos para los coeficientes de ponderación.
- Transferencia y comparación: facilita comparar modelos entrenados en diferentes configuraciones, o transferir conocimiento entre problemas con escalas distintas.

2.4. Arquitectura de red y función de activación

2.4.1. Estructura general de la red

La arquitectura de la red neuronal cumple un rol crucial en la capacidad del modelo para aproximar soluciones suaves y físicamente coherentes a ecuaciones diferenciales. En esta subsección se detallan las decisiones relativas a la estructura de la red y la selección de funciones de activación, justificando su impacto en el desempeño general del modelo.

Si bien se exploraron otros tipos (véase sección 2.9.6), en su forma más común, una PINN se implementa como una red neuronal completamente conectada (fully-connected o feedforward neural network), donde cada capa aplica una transformación afín seguida de una función no lineal de activación (Figura 2.1). Esta arquitectura es capaz de aproximar funciones suaves sobre dominios continuos, lo cual la hace adecuada para resolver problemas de dinámica de sistemas. La red toma como entradas las variables independientes del sistema (por ejemplo, el tiempo y, eventualmente, condiciones iniciales o parámetros de control) y produce como salida las variables de estado, (tales como posición y velocidad). La arquitectura utilizada típicamente se caracteriza por:

- Una capa de n entradas.
- Un número fijo de capas ocultas (L).
- Una cantidad constante de neuronas por capa (k).

- Un esquema simétrico entre capas.
- Una capa de salida de m neuronas con activación lineal.

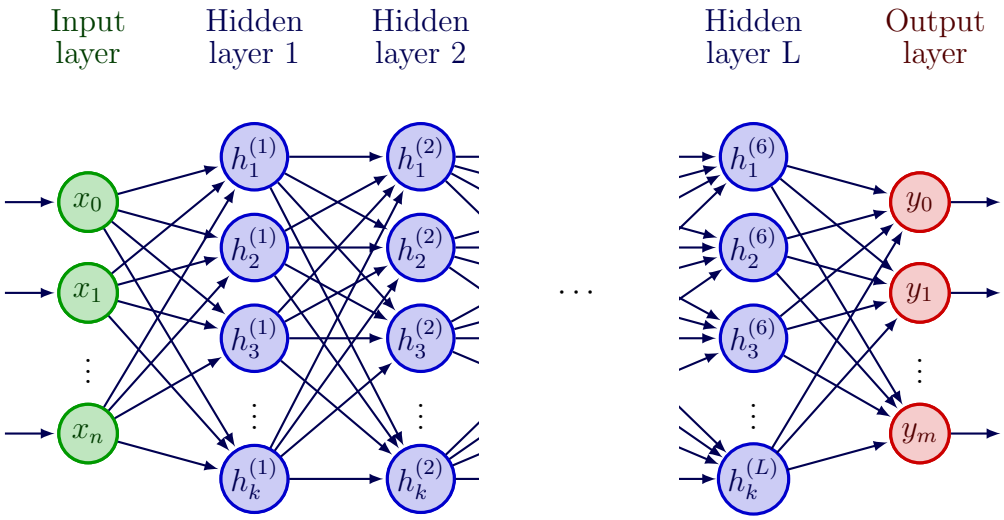


Figura 2.1: Diagrama general de red tipo fully-connected

2.4.2. Elección de funciones de activación

La elección de la función de activación es otro aspecto esencial, al introducir no linealidades que permiten a la red aproximar relaciones complejas. La elección adecuada de la función de activación puede influir significativamente en la capacidad de la red para capturar transiciones suaves, evitar artefactos numéricos y acelerar la convergencia del entrenamiento.

En experimentos preliminares se evaluó el uso de funciones como *ReLU*, cuyo gráfico se observa en la Figura 2.2, ampliamente utilizada por su simplicidad y eficiencia. Sin embargo, se observó que su naturaleza por tramos lineales inducía segmentaciones abruptas en las predicciones, especialmente visibles como líneas rectas en regiones donde se esperaban transiciones suaves (ver sección 2.9.6). Este comportamiento es especialmente perjudicial cuando se intenta modelar fenómenos físicos continuos y diferenciables. Por tal motivo, se optó por funciones de activación suaves y diferenciables como *Tanh* (ver Figura 2.3).

2.5. Optimización

El proceso de entrenamiento de una red neuronal dentro del enfoque PINN (Physics-Informed Neural Network) consiste en minimizar una función de pérdida que combina información de los datos y de las ecuaciones diferenciales que rigen el sistema. Para llevar a cabo esta minimización se recurre a algoritmos de optimización, que ajustan los parámetros de la red neuronal en forma iterativa. En esta sección se describen los dos métodos de optimización más utilizados en el contexto de PINNs: Adam y L-BFGS.

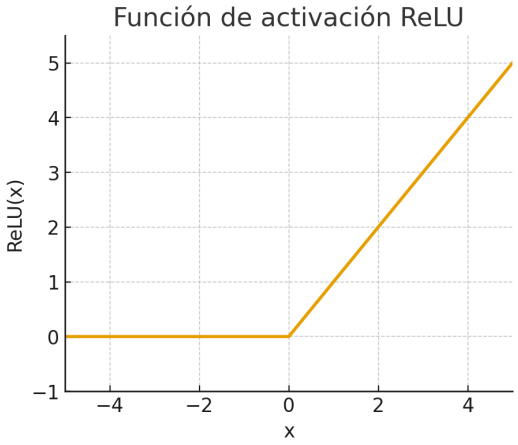


Figura 2.2: Gráfico de función de activación ReLU.

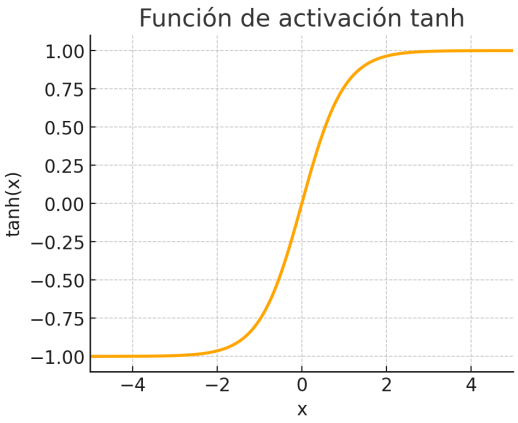


Figura 2.3: Gráfico de función de activación Tanh.

2.5.1. Adam

El optimizador Adam (Adaptive Moment Estimation) es un método basado en gradiente descendente estocástico, que adapta la tasa de aprendizaje para cada parámetro utilizando estimaciones del primer y segundo momento de los gradientes. Esto le permite combinar las ventajas de los algoritmos AdaGrad y RMSProp, mejorando la convergencia en problemas con funciones de pérdida no estacionarias o ruidosas [9]. En el contexto de PINNs, Adam se utiliza comúnmente en las etapas iniciales del entrenamiento debido a su robustez frente a malas inicializaciones y su capacidad para escapar de mínimos locales poco profundos. Su naturaleza estocástica lo vuelve adecuado para trabajar con lotes pequeños de condiciones iniciales y puntos de colocación.

Su actualización está dada por las siguientes ecuaciones:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla\mathcal{L}(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)\left[\nabla\mathcal{L}(\theta_t)\right]^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon}
\end{aligned}
$$

Donde m_t y v_t son los momentos de primer y segundo orden, θ_t representa los parámetros de la red en la iteración t, η es la tasa de aprendizaje, β_1 y β_2 son coeficientes de decaimiento típicamente cercanos a 1 y $\nabla\mathcal{L}(\theta_t)$ es el gradiente de la pérdida respecto a los parámetros.

2.5.2. L-BFGS

El método L-BFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) es un optimizador determinista de segundo orden que no requiere computar ni almacenar la matriz Hessiana completa. En cambio, utiliza una aproximación de la inversa de la Hessiana basada en los últimos pasos de gradiente, lo que lo hace más eficiente en términos de memoria que los métodos clásicos de Newton. En PINNs, L-BFGS se emplea típicamente como etapa de refinamiento luego del entrenamiento con Adam. Dado que es un método cuasi-Newton, puede encontrar con precisión mínimos locales cercanos, especialmente si se parte de una buena inicialización. Sin embargo, su costo computacional por iteración es más elevado, y requiere evaluar el gradiente de toda la función de pérdida completa en cada paso, lo que puede ser prohibitivo para redes grandes o datasets extensos.

La idea general de L-BFGS es resolver la siguiente actualización de parámetros:

$$\theta_{k+1} = \theta_k - \alpha_k H_k \nabla\mathcal{L}(\theta_k) \tag{2.7}$$

Donde H_k es la matriz inversa aproximada del Hessiano θ_k los vectores de parámetros en el paso k, α_k el tamaño de paso y $\nabla\mathcal{L}(\theta_k)$ el gradiente de la función de pérdida [12].

En la práctica, es común usar Adam para una primera fase de exploración del espacio de parámetros, seguido de L-BFGS como etapa de ajuste fino (fine-tuning), aprovechando lo mejor de ambos optimizadores:

- Adam permite una convergencia rápida inicial con menor riesgo de quedar atrapado en mínimos poco óptimos.
- L-BFGS puede afinar la solución hallada, mejorando la precisión en etapas finales.

Esta estrategia combinada ha demostrado ser efectiva en numerosos trabajos relacionados con PINNs y se adopta como esquema estándar en muchas aplicaciones donde los recursos computacionales lo permiten [2] [7].

2.6. Muestreo de collocation points

Para la selección de los puntos de colocación temporales t_{col}, se empleó una estrategia basada en muestreo estratificado utilizando el método Latin Hypercube Sampling (LHS) [18]. Esta técnica asegura una cobertura más uniforme del intervalo de interés [0,T], en comparación

con un muestreo completamente aleatorio o equidistante, lo cual puede resultar beneficioso en problemas donde las soluciones presentan regiones de mayor variabilidad temporal.

A continuación se describen los pasos aplicados para generar los N_{col} puntos de colocación temporales:

1. Se divide el intervalo $[0, T]$ en N_{col} subintervalos de igual tamaño, por ejemplo en [0,1] resulta:
$$\left[0, \frac{1}{N}\right), \left[\frac{1}{N}, \frac{2}{N}\right), \ldots, \left[\frac{N-1}{N}, 1\right)$$
2. En cada subintervalo se selecciona un punto de manera aleatoria (uniforme), garantizando que haya exactamente una muestra por subintervalo.
3. Para mantener el orden temporal en la dinámica, los puntos obtenidos se ordenan de forma creciente.

Esta elección permite capturar la dinámica del sistema de manera más eficiente y con menos redundancia que el muestreo equiespaciado, lo cual resulta especialmente útil cuando se busca entrenar modelos precisos con una cantidad limitada de puntos de colocación.

2.7. Aplicaciones y Extensiones

Desde su introducción, las Physics-Informed Neural Networks (PINNs) han encontrado aplicación en una amplia gama de disciplinas. Su capacidad para incorporar explícitamente leyes físicas como restricciones durante el entrenamiento las ha convertido en una herramienta prometedora tanto para simulación como para inferencia inversa en sistemas físicos [7] [2].

- Mecánica de fluidos: solución de ecuaciones de Navier-Stokes para flujos incompresibles, aerodinámica y flujos turbulentos.
- Transporte y difusión: resolución de ecuaciones de difusión-convección en problemas de transferencia de calor o de concentración de sustancias.
- Problemas inversos: estimación de parámetros físicos o condiciones de frontera desconocidas a partir de observaciones parciales del sistema.
- Sistemas dinámicos no lineales: modelado de osciladores, sistemas caóticos, y robótica.
- Modelado de materiales: simulación de comportamiento elástico, viscoelástico y plástico de materiales bajo diferentes condiciones.
- Biofísica y medicina: propagación de señales eléctricas en el corazón (modelo de FitzHugh Nagumo), dinámica de glucosa-insulina, entre otros.
- Control basado en modelos físicos.

En el contexto del **control basado en modelos físicos**, una extensión importante de las PINNs es el uso de **PINNs orientadas al control (PINC)** introducidas por Antonelo y Camponagara [2], donde se introduce una variable de control u_c que se aprende junto con la solución, permitiendo la optimización de estrategias de control basadas en modelos físicos. Cuando se usan PINNs orientadas al control (PINC), la variable de control u_c forma parte directamente del entrenamiento de la red. Esto evita tener que resolver la PDE repetidamente para cada iteración de la optimización, eliminando además la necesidad de calcular derivadas adjuntas. Por lo tanto, **este enfoque disminuye notablemente el costo computacional comparado con métodos numéricos tradicionales**, siendo especialmente útil para implementar control predictivo en aplicaciones en tiempo real.

2.7.1. Modo autorregresivo

Además del uso estándar de una PINC, donde se predice la evolución del estado a lo largo de un intervalo continuo de tiempo $t \in [0, T]$ dados una condición inicial y un control fijos, es posible utilizarla en modo *autorregresivo*. En esta modalidad, la red se ejecuta repetidamente sobre subintervalos consecutivos, alimentando como nueva condición inicial la última salida predicha. De esta forma, se simula una trayectoria completa mediante la concatenación de múltiples predicciones parciales. Este enfoque se puede aplicar de dos maneras distintas. Por un lado, como simulación libre del sistema, encadenando M bloques de duración T, con el control y estado inicial fijos en cada bloque, y variando únicamente el tiempo t dentro de ese intervalo. Así, la predicción del siguiente paso se obtiene como:

$$x[k] = f_\theta(T, x[k-1], u[k]) \tag{2.8}$$

, donde $x[k-1]$ es el último estado predicho y $u[k]$ el control aplicado en ese intervalo (ver Figura 2.4).

Por otro lado, se puede usar una variante de mayor resolución temporal, en la cual el intervalo de control (por ejemplo, T_c) se subdivide en múltiples pasos internos más pequeños (por ejemplo, $T = T_c/m$), y la red se evalúa autoregresivamente varias veces antes de aplicar el siguiente control. Esto permite entrenar la PINC con una resolución temporal más fina, y luego utilizarla múltiples veces dentro de un mismo paso de control, generando así trayectorias más detalladas sin necesidad de integración numérica explícita. En ambos casos, este uso autoregresivo amplía la utilidad de la PINC, permitiendo simular dinámicas de largo plazo desde redes entrenadas sólo en intervalos cortos.

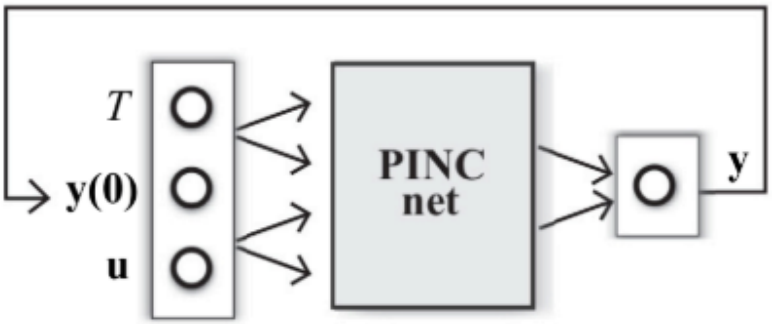


Figura 2.4: Esquema conceptual de una red PINC en modo autorregresivo. Extraído de [2]

2.8. Limitaciones

A pesar de su potencial teórico y empírico, las PINNs enfrentan una serie de desafíos que deben ser considerados al aplicar esta técnica, especialmente en entornos reales o de gran escala. Algunas de las limitaciones más relevantes son:

2.8.1. Dificultad de entrenamiento

- La pérdida total en una PINN suele ser una combinación ponderada de múltiples términos (datos, ecuaciones, condiciones iniciales o de borde). Ajustar estos pesos adecuadamente no es trivial, y un mal balance puede hacer que el entrenamiento se desvíe o no converja.
- El optimizador L-BFGS, que suele ser recomendado para PINNs, requiere gran cantidad de memoria y puede ser ineficiente en redes grandes o problemas con muchos puntos de colocación.

2.8.2. Escalabilidad

- Las PINNs funcionan bien en dominios pequeños, pero su rendimiento disminuye considerablemente cuando se intenta aplicarlas a problemas de alta dimensión, geometrías complejas o largos horizontes temporales.

- Al trabajar con puntos de colocación en todo el dominio, el entrenamiento no se beneficia naturalmente de la paralelización por lotes (mini-batching) como en otros modelos de deep learning.

2.8.3. Evaluación del error y validación

- En muchas aplicaciones, no se dispone de la solución exacta para comparar. Esto complica la evaluación objetiva del desempeño de una PINN entrenada, especialmente si se entrena con pocos datos.

- El valor de la función de pérdida total no siempre refleja con fidelidad la calidad de la solución aprendida, ya que puede haber compensaciones entre los términos de la pérdida.

2.9. Implementación de una PINC para el oscilador de Van der Pol

2.9.1. ¿Por qué el oscilador de Van der Pol?

Antes de abordar el modelado de la dinámica de un brazo robótico con múltiples grados de libertad, se consideró conveniente aplicar la metodología de PINCs a un sistema más simple pero representativo desde el punto de vista dinámico. En este sentido, se eligió la ecuación de Van der Pol como primer caso de estudio [8].

Esta ecuación fue elegida por varias razones:

- Solución conocida y bien estudiada: la ecuación ha sido ampliamente analizada en la literatura, y se dispone de soluciones numéricas de referencia para evaluar cuantitativamente el desempeño del modelo entrenado.

- No linealidad controlada: permite analizar el comportamiento de la PINN frente a una dinámica no lineal cuyo grado de rigidez puede ajustarse mediante el parámetro μ, lo cual sirve como banco de pruebas para observar la estabilidad y precisión del entrenamiento.

- Estructura simple pero desafiante: a pesar de tener solo dos variables de estado, el sistema exhibe comportamientos complejos, como ciclos límite, que lo convierten en un buen punto medio entre sistemas triviales y sistemas de alta dimensión como el brazo robótico.

- Formulación como ODE de primer orden: se adapta naturalmente al marco de PINNs que resuelven sistemas de ecuaciones diferenciales ordinarias, facilitando la implementación y prueba del flujo de trabajo completo (generación de datos, definición de pérdidas, entrenamiento y validación).

- Antecedentes académicos: la ecuación de Van der Pol es un caso de prueba habitual en trabajos sobre PINNs y métodos de aprendizaje de dinámica. Su inclusión permite comparar resultados con estudios previos y validar la implementación.

2.9.2. Modelo y ecuaciones

La ecuación de Van der Pol es un sistema de segundo orden no lineal que describe un oscilador con amortiguamiento no lineal dependiente del estado. Considerando además la señal de control añadida su forma es:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = u(t) \tag{2.9}$$

la cual puede reescribirse como un sistema de primer orden:

$$\begin{cases} \dot{x}_1 = x_2, \\ \dot{x}_2 = \mu(1 - x_1^2)x_2 - x_1 + u(t), \end{cases} \tag{2.10}$$

donde $\mu > 0$ regula el grado de no linealidad y rigidez del sistema. El caso de $\mu = 0$ recupera el clásico oscilador armónico que fue estudiado por el ingeniero Van der Pol en el contexto de sus trabajos con los primeros circuitos electrónicos a principios del siglo XX.

El proceso para llegar a una PINN que fuera capaz de aprender la ecuación de Van der Pol para distintos controles y condiciones iniciales fue gradual, comenzando por una que simplemente aprendiera la ecuación para una única condicion inicial, con término $\mu = 1$ (no linealidad) y sin término de control, siguiendo por una red que aprendiera la ecuación para diversas condiciones iniciales, introduciendo también la no linealidad ($\mu \neq 0$), y por último una que se entrena con diversas condiciones iniciales y diversos valores de control. Por esto, la sección de resultados (2.9.6) se divide en estos tres casos.

2.9.3. Función de pérdida

La función de pérdida total utilizada durante el entrenamiento está compuesta por dos términos:

- **Pérdida física (residuo de la dinámica)**:

$$\mathcal{L}_{\text{physics}}(\theta) = \frac{1}{N_r}\sum_{i=1}^{N_r}\left[\left|\frac{d\hat{x}}{dt}(t_i) - \hat{y}(t_i)\right|^2 + \left|\frac{d\hat{y}}{dt}(t_i) - \mu(1 - \hat{x}^2(t_i))\hat{y}(t_i) + \hat{x}(t_i) - u(t_i)\right|^2\right] \tag{2.11}$$

- **Pérdida en condiciones iniciales**:

$$\mathcal{L}_{\text{IC}}(\theta) = |\hat{x}(0) - x(0)|^2 + |\hat{y}(0) - y(0)|^2 . \tag{2.12}$$

  En este caso no se emplea la componente de pérdida basada en datos ($loss_{data}$), ya que el sistema estudiado (la ecuación de Van der Pol) se modela directamente a partir de su formulación matemática conocida. Al no contar con datos experimentales reales que representen trayectorias observadas del sistema, no tiene sentido calcular el error entre predicciones de la PINN y un conjunto de datos externos.

  Los valores tomados como solución objetivo con los cuales contrastar a los valores resultados de la PINN en cada cada t_i son obtenidos resolviendo la ecuación de Van der Pol utilizando el método explícito Runge-Kutta de orden 5.

  La pérdida total empleada en el entrenamiento es una combinación ponderada de estos términos, utilizando los pesos relativos $w_{\text{loss\_col}}$ y $w_{\text{loss\_bc}}$:

$$\mathcal{L}(\theta) = w_{\text{loss\_col}}\mathcal{L}_{\text{physics}}(\theta) + w_{\text{loss\_bc}}\mathcal{L}_{\text{IC}}(\theta) \tag{2.13}$$

2.9.4. Especificaciones de entrenamiento

Arquitectura

Luego de experimentar también con redes neuronales del tipo GRU (Gated Recurrent Unit) y RNN (Recurrent Neural Network) para una única condición inicial, por los resultados obtenidos que se observan en la siguiente sección, la arquitectura adoptada para las siguientes fases de entrenamiento es una red completamente conectada (*fully-connected neural network, FCNN*) compuesta por seis capas ocultas con 100 neuronas cada una, una capa de entrada de 4 neuronas correspondiente a las 4 dimensiones de los datos y una capa de salida de 2 neuronas:

$$[t,\, x_0,\, y_0,\, u] \;\longrightarrow\; [\hat{x}(t),\, \hat{y}(t)].$$

Inicialización de pesos

La inicialización utilizada para los pesos por los que se pondera la salida de cada neurona es la inicialización *Xavier Normal* la cual asigna los valores al azar con una distribución normal.

Optimizador

El entrenamiento de la red se realiza utilizando el optimizador **Adam** con un learning rate controlado por un `scheduler`, con un valor inicial de (5×10^{-3}) que se múltiplica por un factor de 0.3 por cada 200 épocas (parámetro de paciencia) en que el la función de costo no disminuye, pudiendo llegar a un mínimo de 1×10^{-5}. Esto es adecuado para problemas no convexos y entornos de entrenamiento ruidosos. El optimizador L-BFGS fue descartado pues no generó bueno resultados tanto al utilizarse de forma única como al emplearse como método de refinación luego de optimizado con Adam.

Hiperparámetros

La elección de los hiperparámetros es un aspecto fundamental en el rendimiento de una red neuronal, especialmente en el contexto de PINNs, donde interactúan tanto el ajuste de datos como la resolución de ecuaciones diferenciales. Por ello, casi todos los hiperparámetros relevantes del modelo fueron estudiados empíricamente durante el desarrollo, explorando distintos valores y combinaciones posibles. Los resultados se encuentran en la seccion 2.9.6. Los hiperparámetros explorados en profundidad son:

- Número de capas ocultas
- Número de neuronas por capa
- Función de activación
- Cantidad de épocas de entrenamiento
- Pesos relativos de términos de Loss
- Cantidad de condiciones iniciales
- Cantidad de valores de control
- Cantidad de puntos de colocación

2.9.5. Diagrama de la implementación

A continuación, en la figura 2.5 se muestra el diagrama representativo del proceso de entrenamiento realizado.

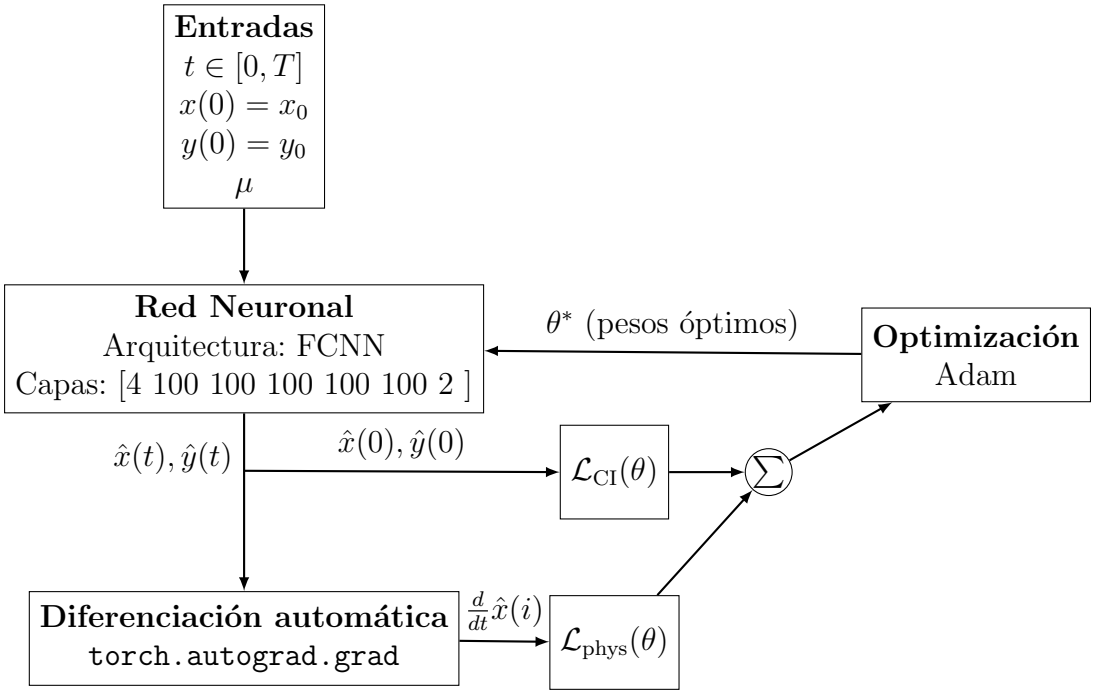


Figura 2.5: Diagrama de implementación de la PINN entrenada con ecuación del oscilador de Van der Pol.

2.9.6. Experimentos y Resultados

Con el objetivo de validar de forma progresiva el enfoque propuesto, se plantearon objetivos intermedios que permitieran avanzar de manera gradual en la complejidad del problema. En una primera etapa, se entrenó una PINN para aprender la dinámica de la ecuación de Van der Pol bajo una única condición inicial fija, lo cual permitió analizar con mayor claridad la capacidad de aproximación del modelo en un entorno controlado. Superado este primer desafío, se amplió el alcance al considerar múltiples condiciones iniciales, lo que introdujo variabilidad en los estados iniciales del sistema y permitió evaluar la generalización del modelo. Finalmente, se abordó el caso más general, incorporando además una entradas de control con valor constante, con el fin de modelar un escenario dinámico más representativo del objetivo del proyecto. Esta secuencia escalonada facilitó tanto el ajuste fino de los hiperparámetros como la identificación de las limitaciones específicas en cada nivel de complejidad.

Única condición inicial fija

Antes de comenzar, fue necesario definir una arquitectura base y un conjunto de hiperparámetros iniciales para entrenar la red neuronal. Dado que no existen lineamientos específicos ampliamente establecidos para el uso de PINNs aplicadas al modelado dinámico de brazos robóticos, se optó por tomar como referencia estudios previos y trabajos afines que abordan problemas similares en el ámbito de sistemas dinámicos y control.

En particular, se consideraron configuraciones comunes empleadas en la literatura para sistemas no lineales en tiempo continuo, con el objetivo de contar con una base razonable

desde la cual ajustar posteriormente los parámetros mediante experimentación empírica. En este caso, la elección de hiperparámetros iniciales se basó fuertemente en el trabajo sobre PINNs del Ing. Christian Diaz [5].

Esta estrategia permitió establecer un punto de partida viable ante la ausencia de un marco teórico definitivo que guíe la selección óptima de hiperparámetros en este tipo de aplicaciones.

Se comenzó entonces con una red de arquitectura fully-connected (FCNN) con los hiperparámetros de la tabla 2.1.

Tabla 2.1: Hiperparámetros iniciales para única condición inicial

Parámetro	Detalle
Cantidad de capas	8
Cantidad de neuronas por capa	50
T	20s
Learning rate	1×10^{-4}
Activación	ReLU
Pesos	$w_{bc} = 0{,}5, w_{col} = 0{,}5$
Optimizador	Adam
Cantidad de épocas	5000
Cantidad de puntos de colocación	5000
Parámetro no linealidad(μ)	0

Como era de esperarse, con resultados lejos de ser satisfactorios, se comenzaron a probar entrenamientos con distintas configuraciones de hiperparámetros. En primer lugar se fue aumentando la cantidad de puntos de colocación y épocas pero aún así se obtenían resultados como los de la Figura 2.6. En ella se observa que la predicción hecha por la PINN comienza muy bien pero rapidamente se va a 0, entendiendo así que aún faltaba penalización en cuanto al error por puntos de colocación, ya que la penalización por condición inicial se cumple perfectamente.

Luego de hechas estas correcciones, si bien, como se observa en la Figura 2.7, la solución se asemeja más a una sinusoide, se aprecian lineas rectas y cambios bruscos en contraparte a las transiciones suaves que presentan las funciones sinusoidales.

Debido a la complejidad introducida por la gran cantidad de hiperparámetros que regulan el aprendizaje de cualquier red neuronal, fue difícil hallar el causante de este problema, siendo este uno de los primeros obstáculos debido al tiempo que llevó encontrar la solución. Tras un análisis más detallado, se identificó que el uso de la función de activación $ReLU$ estaba introduciendo este comportamiento no deseado, ya que su naturaleza por tramos lineal impide representar eficientemente funciones derivables o con transiciones suaves, especialmente en contextos donde se espera continuidad y diferenciabilidad del campo dinámico. Por esta razón, se reemplazó la activación por la función $Tanh$, que es suavemente diferenciable en todo su dominio y más adecuada para modelar la dinámica de sistemas físicos continuos. Esta modificación resultó en una mejora significativa en la calidad de las predicciones, evidenciada tanto en la reducción de la pérdida como en la forma de las trayectorias generadas por la red. Con esta activación, para $T = 10s$, 2500 épocas, $N_{col} = 5000$ y $\mu = 1$ (no linealidad) la solución brindada por la PINN es casi idéntica a la solución exacta, como se observa en la Figura 2.8.

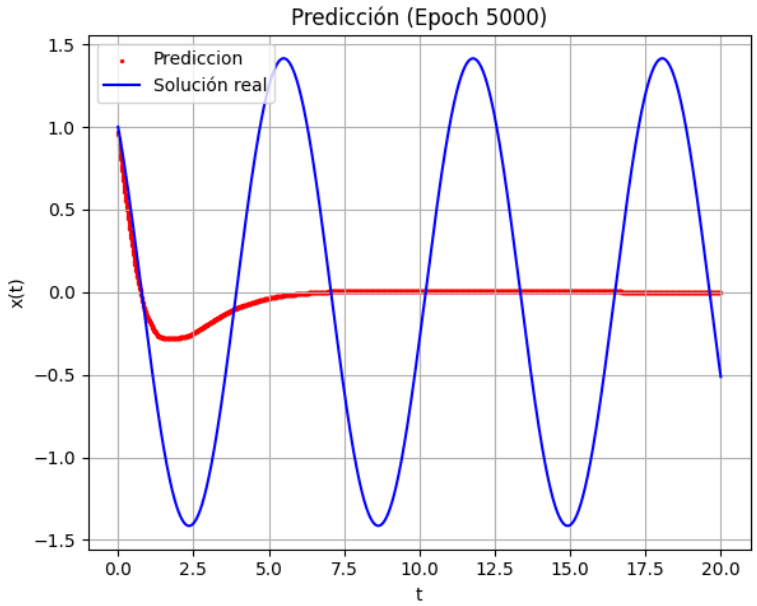


Figura 2.6: Gráfico de coordenada x(t) de ecuación de Van der Pol con condición inicial $x(0) = 1{,}0$ con valores de hiperparámetros de tabla 2.1. La curva roja representa la predicción de la PINN mientras la curva azul representa la solución hayada con el método Runge-Kutta de orden 4 (RK4).

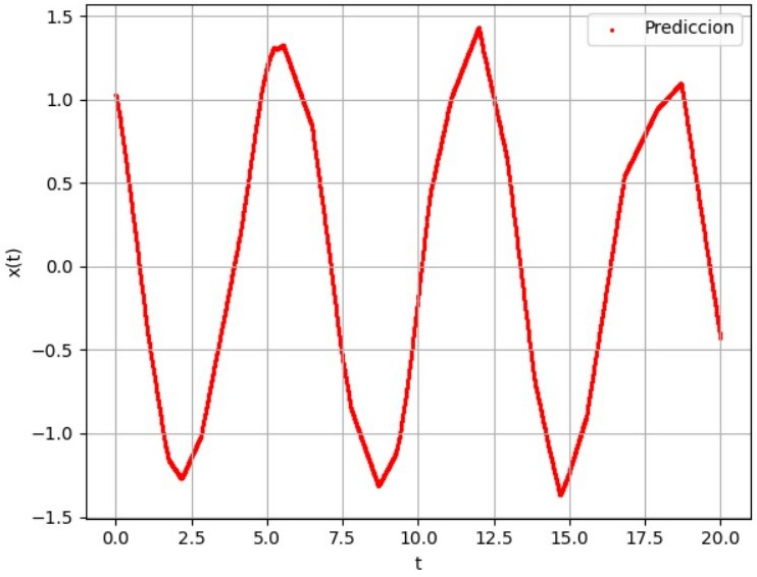


Figura 2.7: Gráfico de coordenada x(t) de ecuación de Van der Pol con condición inicial $x(0) = 1{,}0$ con valores de hiperparámetros de tabla 2.1.

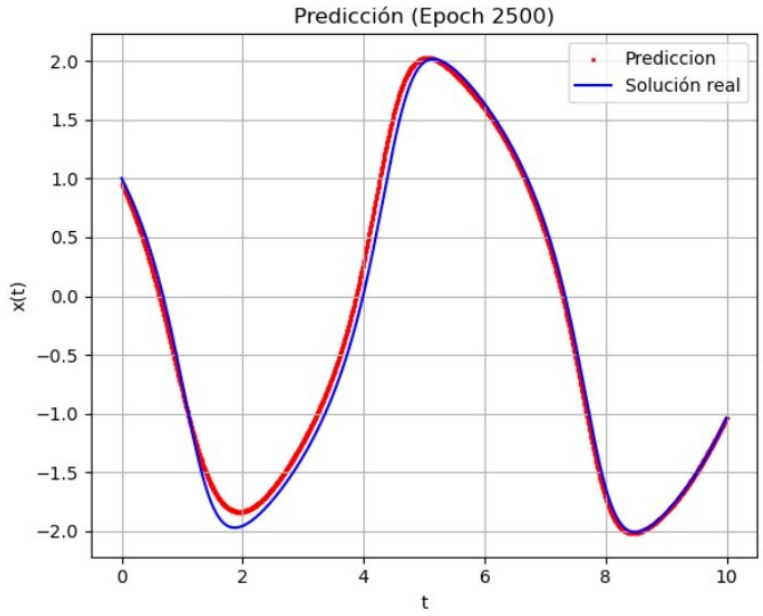


Figura 2.8: Gráfico de x(t) para ecuación no lineal de Van der Pol con condición inicial $x(0) = 1{,}0, T = 10s$, 2500 épocas de entrenamiento, 5000 puntos de colocación, parámetro de no linealidad $\mu = 1$ y función de activación $Tanh$. Optimizador, pesos, learning rate y estructura se mantienen valores de tabla 2.1.

Múltiples condiciones iniciales

Con el objetivo de avanzar hacia escenarios de mayor generalidad y robustez, se consideró ahora el aprendizaje de la dinámica de la ecuación de Van der Pol para múltiples condiciones iniciales, formulándose el paradigma como muestra la ecuación 2.14.

$$\ddot{x} - \mu\,(1 - x^2)\,\dot{x} + x = 0, \qquad \mu > 0, \tag{2.14}$$

Esta extensión implica no solo un desafío adicional para la red, sino también la necesidad de definir un rango adecuado de valores para las condiciones iniciales $(q_0, \dot{q_0})$. Además, a diferencia de las primeras pruebas realizadas sobre intervalos de tiempo prolongados (por ejemplo, hasta T=10 segundos), en este caso se eligió un horizonte temporal considerablemente menor, del orden de décimas de segundo. Esta decisión se alinea con la escala temporal de actualización típica en sistemas de control como los robots manipuladores, donde los modelos dinámicos deben predecir comportamientos en intervalos muy breves. Forzar a la red a capturar la dinámica sobre ventanas temporales innecesariamente largas no solo es costoso computacionalmente, sino que puede también perjudicar el ajuste fino de los parámetros en el rango temporal de interés. Como consecuencia, es posible reducir la complejidad de la red sin perder eficacia.

Definiendo los posibles valores iniciales tanto para x_0 como para y_0 como un valor al azar de una distribución uniforme en el rango $[-2, 2]$, manteniendo algunos hiperpárametros de la etapa anterior y modificando otros, se entrenaron distintos modelos.

Por otra parte, para medir cuantitativamente el desempeño de los modelos entrenados, se diseñó una función de evaluación que compara las predicciones de la red neuronal con la solución real de la ecuación de Van der Pol. Esta función permite probar el modelo sobre múltiples condiciones iniciales aleatorias, utilizando una grilla temporal compartida, y calcula diferentes métricas de error. Primero, se generan de manera aleatoria múltiples condiciones iniciales (x_0, y_0) dentro del rango definido. Luego, se resuelve la ecuación diferencial para cada una de ellas utilizando un integrador numérico (considerado como

"verdad física") y se comparan los resultados con las predicciones generadas por la red PINN entrenada, sobre la misma grilla de tiempos. Se consideran dos tipos de métricas:

- Error final: mide el error cuadrático medio (MSE) entre la predicción y la verdad física solo en el tiempo final $t = T$.
- Error de trayectoria: mide el MSE promedio a lo largo de toda la trayectoria $t \in [0, T]$.

Ambos errores se calculan por separado para cada variable del sistema (x, y), y luego se combinan para obtener una pérdida total.

En base a esta métrica y al testeo cualitativo mediante la observación de gráficas comparativas se llegó a los hiperparámetros adecuados. En la tabla 2.3 se presentan algunos de los modelos entrenados y sus resultados.

El resto de hiperparámetros se fijaron con los valores de la tabla 2.2.

Tabla 2.2: Hiperparámetros para PINN entrenada con múltiples condiciones iniciales

Parámetro	Valor
Tiempo de simulación (T)	0.5s
Learning rate	1×10^{-4}
Pesos	$w_{bc} = 0{,}4, w_{col} = 0{,}6$
Optimizador	Adam
Cantidad de puntos de colocación	500
Parámetro no linealidad(μ)	1

Tabla 2.3: Hiperparámetros utilizados y desempeño de los modelos

Capas	Neuronas por capa	Épocas	Condiciones iniciales	Valor RMSE
4	10	500	10	$8{,}7 \times 10^{-2}$
4	10	1000	10	$2{,}9 \times 10^{-2}$
6	20	1000	10	$2{,}0 \times 10^{-2}$
4	10	1000	30	$1{,}2 \times 10^{-2}$
4	10	1000	50	$9{,}3 \times 10^{-3}$
4	10	1000	70	$3{,}3 \times 10^{-3}$
4	10	1000	100	$1{,}5 \times 10^{-2}$
8	20	1000	100	$4{,}5 \times 10^{-4}$
10	40	1000	100	$1{,}4 \times 10^{-4}$

Obteniendo para el último modelo de la tabla, para una condición inicial al azar (no usada en el entrenamiento) gráficas como la de la imágen 2.9 donde se puede observar que la red aprendió la ecuación en ese entorno.

Además de utilizar redes totalmente conectadas (FCNN), se experimentó con arquitecturas recurrentes como RNN (Recurrent Neural Network) y GRU (Gated Recurrent Unit). Estas estructuras están diseñadas específicamente para modelar datos secuenciales, como señales temporales, ya que incorporan bucles internos que les permiten retener información de estados anteriores. En una RNN estándar, cada salida depende no solo del input actual, sino también de una memoria del paso anterior. La GRU, por su parte, es una evolución de la RNN que introduce mecanismos de compuertas para controlar de forma más

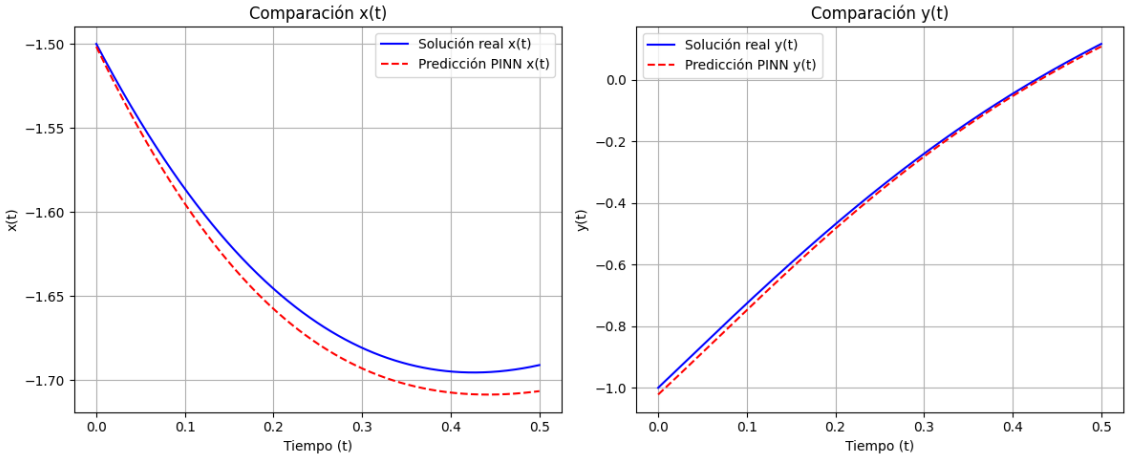


Figura 2.9: Gráfico comparativo de x(t) entre solución (curva azul continua) y predicción de la PINN (curva roja discontinua) para condición inicial de validación $(-1{,}5, 1{,}0)$ con parámetros T, learning rate, pesos, optimizador, cantidad de puntos de colocación y parámetro de no linealidad de tabla 2.2 estructura de 10 capas de 40 neuronas, 1000 épocas y 100 condiciones iniciales.

eficiente qué información se mantiene y cuál se descarta, lo que facilita el entrenamiento y mejora el rendimiento en secuencias largas.

Dado que la solución de la ecuación de Van der Pol puede interpretarse como una señal dinámica en el tiempo, con posibles oscilaciones complejas, se exploró la hipótesis de que este tipo de redes podría capturar mejor la estructura temporal de la dinámica, en comparación con una red estática como la FCNN. En consecuencia, se entrenaron y evaluaron versiones PINN basadas en RNN y GRU, analizando su desempeño tanto en el ajuste de trayectorias como en la predicción del estado final del sistema.

Luego de entrenar distintos modelos utilizando estas estructuras y de explorar diversos valores de hiperparámetros para cada una de ellas, se concluyó que las redes recurrentes no ofrecían ventajas sustanciales en este caso. Si bien estas estructuras están diseñadas para trabajar con secuencias temporales, los resultados obtenidos no superaron a los modelos basados en redes totalmente conectadas (FCNN) en términos de error de predicción. Más aún, el entrenamiento de las redes recurrentes resultó considerablemente más costoso en tiempo, tanto por la mayor complejidad interna de cada célula como por la dificultad para optimizar sus parámetros. Por estos motivos, se optó por continuar el desarrollo únicamente con arquitecturas FCNN, que demostraron ser más eficientes y con mejor desempeño en este problema en particular. Reflejo de esto son los valores de la tabla 2.4 donde se entrenó a las tres estructuras utilizando los mismos valores de hiperparámetros del mejor modelo obtenido con FCNN en la tabla 2.3. Se puede observar que si bien el modelo RNN llega a errores del orden de FCNN, el tiempo de entrenamiento es mayor, es por esto que se concluye que la estructura FCNN es mas conveniente.

Tabla 2.4: Comparación de desempeño entre distintas arquitecturas de red

Arquitectura	RMSE	Tiempo entren.(min)
FCNN	$1,4 \times 10^{-4}$	12
RNN	$5,7 \times 10^{-4}$	20
GRU	$2,3 \times 10^{-3}$	78

Múltiples condiciones iniciales y constantes de control (PINC)

Con el objetivo de acercarse progresivamente al tipo de entrada que utilizará el modelo final aplicado al brazo robótico, se planteó un tercer escenario en el que se entrena la red para predecir la evolución de la ecuación de Van der Pol no solo a partir de distintas condiciones iniciales, sino también considerando un valor de control constante asociado a cada una de ellas. Este valor de control se mantiene fijo durante toda la simulación y representa una entrada adicional al sistema, lo que implica una extensión del vector de entrada en la red. En términos prácticos, esto significa que el modelo ahora recibe tres variables: el tiempo, la condición inicial de posición, la condición inicial de velocidad, y se le agrega una cuarta dimensión correspondiente al valor de control aplicado. Esta nueva configuración permite simular escenarios más cercanos al comportamiento que se espera en tareas de control continuo, y sienta las bases para futuras aplicaciones sobre sistemas dinámicos más complejos.

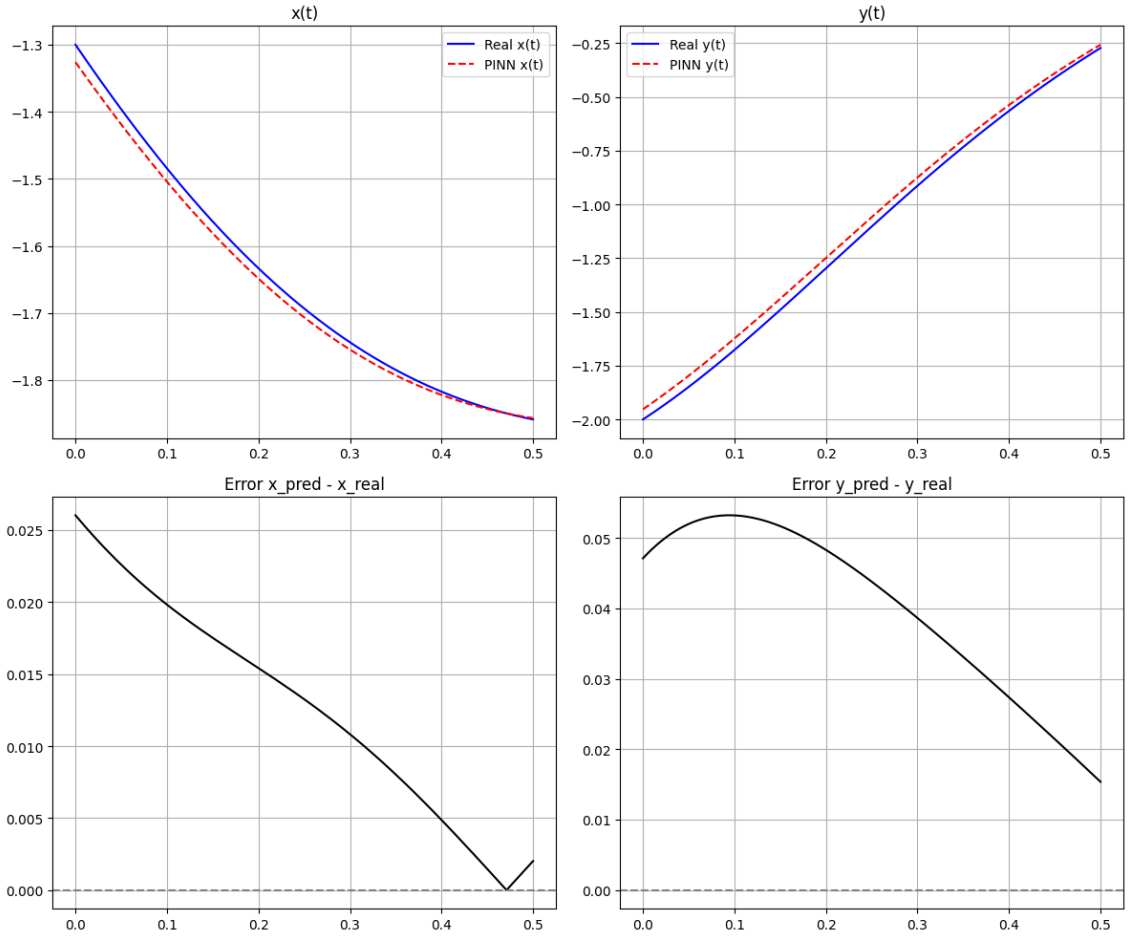


Figura 2.10: Arriba:Gráfico comparativo entre solución (curva azul continua) y predicción de la PINC (curva roja discontinua) para condicion inicial $(-1{,}3, -2{,}0)$ y valor de control al azar uniforme en $[-0{,}1, 0{,}1]$. Abajo: A la izquierda, gráfico de error entre curva x(t) predicha por la PINC y solución brindada por método RK45, a la derecha, gráfico de error entre curva y(t) predicha por la PINC y solución brindada por método RK45.

Si bien era posible incorporar el control de distintas formas, por ejemplo, asignando múltiples valores de control constante a cada condición inicial, se optó por una estrategia más simple y coherente con la aplicación prevista en el brazo robótico definiendo un único valor de control constante por cada condición inicial. Dichos valores se generaron de manera aleatoria dentro del rango [-0.1, 0.1], representando pequeños torques o fuerzas de entrada

que permiten explorar cómo varía la dinámica del sistema ante distintas excitaciones iniciales sin introducir una complejidad innecesaria en la etapa de entrenamiento. Como era de esperar, además de aumentar el tiempo de entrenamiento por aumentar la dimensión de entrada, obtener un desempeño del orden del mejor modelo obtenido sin valores de control, requiere más épocas, en este caso 1500. El tiempo de entrenamiento fue de 20 minutos frente a los 12 del entrenamiento sin valores de control. La imagen muestra el resultado para un valor al azar dentro del rango definido tanto para la condición inicial como para el valor de control.

Capítulo 3

Control predictivo por modelo

En este capítulo se desarrolla el marco teórico y práctico del *Model Predictive Control* (MPC), técnica empleada como núcleo del sistema de control propuesto. En primer lugar, se presentan los fundamentos conceptuales del MPC, describiendo su formulación general, la función de costo, las restricciones y los métodos de optimización asociados. Posteriormente, se introduce su extensión hacia sistemas no lineales y su integración con modelos aprendidos mediante redes neuronales informadas por física (*Physics-Informed Neural Networks*, PINNs).

3.1. Introducción

El *Control Predictivo por Modelo* (MPC, *Model Predictive Control*) es una de las estrategias de control avanzado más utilizadas en la actualidad para sistemas de varias variables y no lineales. Su principio fundamental se basa en la predicción del comportamiento futuro del sistema mediante un modelo dinámico, resolviendo en cada instante un problema de optimización sobre un horizonte temporal finito [4, 16]. A diferencia de los controladores clásicos, MPC permite incorporar restricciones sobre los estados y las acciones de control, lo que lo convierte en una herramienta adecuada para sistemas físicos con limitaciones de actuadores o saturaciones de variables.

En cada paso de control, el MPC calcula una secuencia de acciones de control futuras que minimiza una función de costo cuadrática, la cual penaliza tanto la desviación del estado respecto a la referencia como la variación de los controles. Solo la primera acción de control de la secuencia óptima es aplicada al sistema, mientras que el proceso se repite en el siguiente instante con el estado actualizado. Este enfoque, conocido como *horizonte móvil* o *receding horizon*, otorga al controlador una capacidad de adaptación continua frente a perturbaciones y no linealidades [4, 33].

El MPC combina las ventajas del control óptimo con la flexibilidad de las técnicas numéricas modernas, y su formulación es lo suficientemente general como para abarcar desde sistemas lineales discretos hasta dinámicas altamente no lineales.

3.1.1. Motivación

El *Model Predictive Control* (MPC) cuenta con la ventaja de ser utilizado como estrategia de control para sistemas dinámicos complejos. Su principal fortaleza radica en la capacidad de anticipar el comportamiento futuro del sistema mediante un modelo interno que predice la evolución de los estados a lo largo de un horizonte temporal. Esta

característica permite calcular acciones de control óptimas que no sólo corrigen el error actual, sino que consideran la respuesta futura del sistema y las restricciones físicas de los actuadores [4, 16].

A diferencia de otros controladores clásicos como los PID o los esquemas lineales de retroalimentación, el MPC es especialmente ventajoso cuando se dispone de un modelo capaz de describir de manera precisa o aproximada la dinámica del sistema. En este sentido, su estructura predictiva lo convierte en una herramienta ideal para el control de sistemas no lineales, donde la linealización local o la aproximación empírica pueden resultar insuficientes. Además, su formulación por optimización permite incorporar de forma explícita restricciones sobre los estados, las velocidades o los torques, manteniendo un desempeño estable dentro de los límites operativos del sistema [16].

El potencial del MPC se ve cuando se combina con modelos de aprendizaje basados en física, donde se es capaz de estimar la dinámica del sistema en tiempo real. En este contexto, las *Physics-Informed Neural Networks* (PINNs) surgen como una alternativa robusta para representar la dinámica de sistemas complejos mediante redes neuronales que respetan las leyes físicas que los gobiernan [24]. Antonelo (2024) propone una extensión de este enfoque denominada *Physics-Informed Neural Nets for Control* (PINC), donde la red neuronal no sólo predice la evolución del sistema, sino que se integra directamente en el lazo de control predictivo, sirviendo como modelo interno dentro del MPC [2].

La combinación de MPC y PINNs esquematizada en la Figura 3.1 ofrece un control variable, donde el MPC aprovecha la predicción y la capacidad para manejar restricciones, mientras que la PINN aporta un modelo dinámico diferenciable y de bajo costo computacional. En conjunto, constituyen un sistema capaz de abordar problemas de control no lineal con precisión, de forma estable y con la capacidad de generalización.

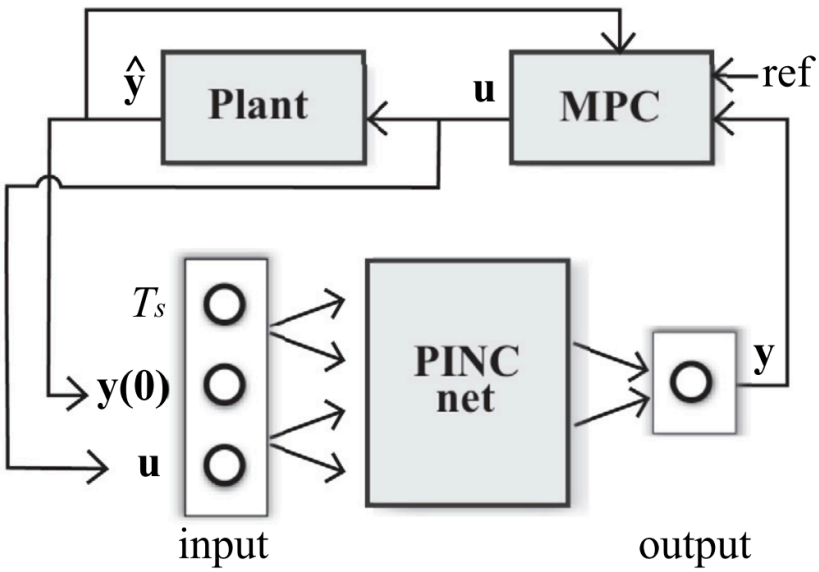


Figura 3.1: Esquema conceptual de la integración entre una red PINN y el controlador MPC. La red aprende la dinámica del sistema a partir de las ecuaciones físicas, mientras el MPC optimiza la acción de control considerando predicciones en horizonte móvil. Extraído de [2]

3.2. Fundamentos de MPC

3.2.1. Formulación general del MPC no lineal

El *Model Predictive Control* (MPC) no lineal se fundamenta en la predicción del comportamiento futuro de un sistema dinámico discreto, representado de manera general como:

$$\mathbf{x}(k+1) = f\big(\mathbf{x}(k), \mathbf{u}(k)\big), \tag{3.1}$$

donde:

- $\mathbf{x}(k) \in \mathbb{R}^n$ representa el vector de estados del sistema en el instante de muestreo k;
- $\mathbf{u}(k) \in \mathbb{R}^m$ es el vector de acciones de control aplicadas;
- $f(\cdot)$ describe la dinámica del sistema, que puede ser lineal o no lineal, continua o con retardo de tiempo.

Este planteo general sigue la formulación descrita por Normey-Rico y Camacho [20], donde se destaca la capacidad del MPC para anticipar la respuesta del sistema mediante un modelo explícito, compensando efectos de dinámica retardada y no linealidad. La estructura permite, en cada instante, calcular una secuencia futura de controles óptimos $\mathbf{U} = \{\mathbf{u}(k), \mathbf{u}(k+1), \ldots, \mathbf{u}(k+N_2-1)\}$ sobre un horizonte de predicción finito N_2.

El objetivo del controlador es minimizar una función de costo cuadrática definida como:

$$J = \sum_{i=0}^{N_2-1} \ell\big(\mathbf{x}(k+i), \mathbf{u}(k+i)\big) + V_f\big(\mathbf{x}(k+N_2)\big), \tag{3.2}$$

donde el término $\ell(\cdot)$ penaliza las desviaciones instantáneas respecto a la referencia y el esfuerzo de control, y $V_f(\cdot)$ es un término que contribuye a la estabilidad del sistema al final del horizonte.

La función de costo instantánea adopta típicamente la forma cuadrática:

$$\ell(\mathbf{x}, \mathbf{u}) = (\mathbf{x} - \mathbf{x}_{\text{ref}})^T Q (\mathbf{x} - \mathbf{x}_{\text{ref}}) + \mathbf{u}^T R \mathbf{u}, \tag{3.3}$$

donde $Q \succeq 0$ y $R \succ 0$ son matrices de peso que ponderan el seguimiento de la referencia y el esfuerzo de control. El horizonte de predicción N_2 determina el número de pasos futuros considerados en la optimización.

En cada ciclo de control, el problema de optimización se resuelve en línea bajo un esquema de *horizonte móvil*: se aplica únicamente la primera acción óptima $\mathbf{u}(k)$, se mide el nuevo estado del sistema, y el horizonte se desplaza un paso hacia adelante, como se ilustra en la Figura 3.2. Este enfoque otorga al MPC su carácter adaptativo y robusto frente a perturbaciones externas, incertidumbre del modelo y no linealidades del sistema.

3.2.2. Función de costo

La función de costo es el componente principal del *Nonlinear Model Predictive Control* (MPC), definiendo el criterio de desempeño que el optimizador busca minimizar en cada ciclo de control. Su formulación combina la penalización del error de seguimiento entre la salida del sistema y la referencia deseada, junto con una penalización adicional sobre las variaciones del control aplicado, buscando suavizar la respuesta del sistema.

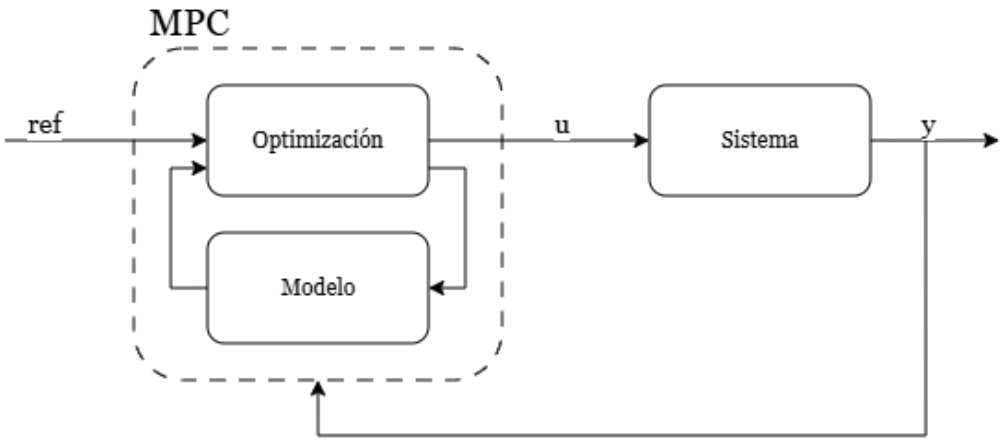


Figura 3.2: Esquema general del controlador predictivo no lineal (MPC). En cada instante k, el MPC optimiza una secuencia de acciones futuras utilizando un modelo del sistema, aplicando solo el primer control y repitiendo el proceso en el siguiente paso.

Siguiendo la formulación propuesta por Camacho y Bordons [4] y empleada en Antonelo [2], el costo total del MPC discreto se expresa como:

$$J = \sum_{j=N_1}^{N_2} \big[\mathbf{x}(k+j) - \mathbf{x}_{\text{ref}}(k+j)\big]^T Q \big[\mathbf{x}(k+j) - \mathbf{x}_{\text{ref}}(k+j)\big] + \sum_{i=0}^{N_u-1} \Delta\mathbf{u}(k+i)^T R\, \Delta\mathbf{u}(k+i), \tag{3.4}$$

sujeto a la dinámica del sistema modelada por:

$$\mathbf{x}(k+j+1) = f_\theta\big(\mathbf{x}(k+j), \mathbf{u}(k+j)\big), \quad j = 0, \ldots, N_2 - 1, \tag{3.5}$$

donde $f_\theta(\cdot)$ representa el modelo dinámico aprendido mediante la *Physics-Informed Neural Network* (PINN), que sustituye al modelo físico tradicional en la predicción del comportamiento del sistema.

En las ecuaciones anteriores:

- $\mathbf{x}(k) \in \mathbb{R}^n$ es el vector de estados en el instante k;
- $\mathbf{u}(k) \in \mathbb{R}^m$ es el vector de entradas de control;
- $\Delta\mathbf{u}(k) = \mathbf{u}(k) - \mathbf{u}(k-1)$ representa el incremento del control;
- N_1 (comunmente en 0) y N_2 son los límites inferior y superior del horizonte de predicción;
- N_u es el horizonte de control (cantidad de acciones a optimizar, siendo $N_u \leq N_2$);
- $Q \succeq 0$ y $R \succ 0$ son matrices de peso que penalizan el error de estado y la variación de control, respectivamente.

El primer término de la ecuación (3.4) asegura el seguimiento de la referencia a lo largo del horizonte de predicción, mientras que el segundo término limita los cambios sucesivos en la señal de control, evitando esfuerzos excesivos o comportamientos abruptos en los actuadores.

Este tipo de formulación es especialmente relevante para sistemas mecánicos o robóticos, donde las variaciones rápidas en el torque pueden generar oscilaciones o inestabilidades.

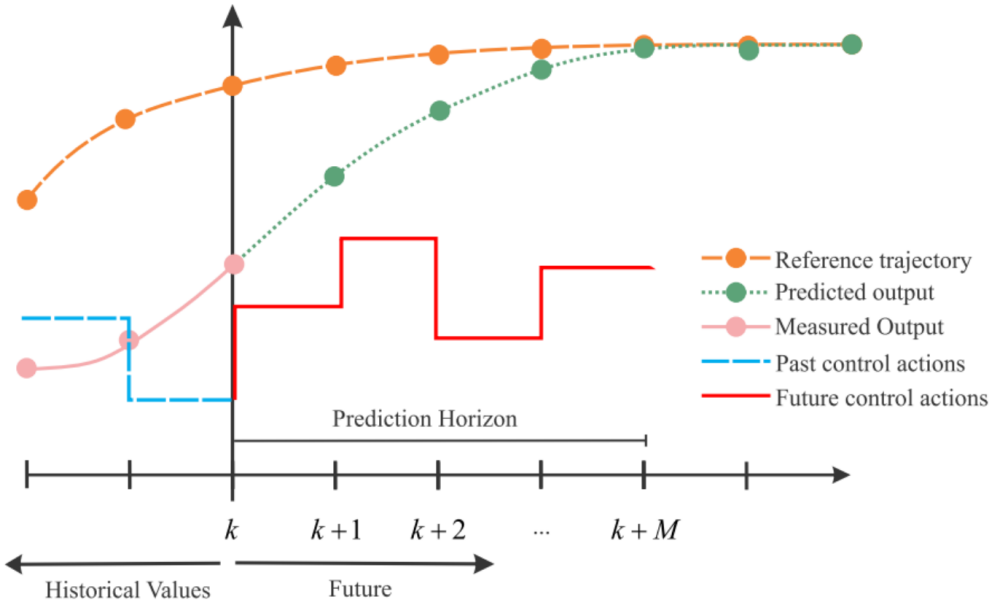


Figura 3.3: Representación esquemática del horizonte de predicción en el MPC. La referencia deseada (naranja) es comparada con la salida predicha del modelo (verde), mientras que las acciones de control futuras (rojo) son optimizadas para reducir el error en cada instante. Solo la primera acción es aplicada al sistema real, siguiendo el principio de horizonte móvil. Extraído de [2].

Como se observa, la Figura 3.3 muestra el principio fundamental del MPC, donde el controlador predice la evolución futura del sistema a lo largo de un horizonte de tiempo y calcula las acciones de control que minimizan el error entre la referencia deseada y la respuesta predicha. En cada instante k, el optimizador determina una secuencia de acciones $\{\mathbf{u}(k), \mathbf{u}(k+1), \ldots, \mathbf{u}(k+N_u-1)\}$ que genera una trayectoria predicha $\hat{\mathbf{y}}(k+i)$ lo más cercana posible a la referencia $\mathbf{y}_{\text{ref}}(k+i)$. Solo la primera acción $\mathbf{u}(k)$ se aplica al sistema real, y en el siguiente paso se repite el proceso utilizando las nuevas mediciones del sistema, implementando así el principio de *horizonte móvil*.

Este mecanismo permite que el MPC se adapte continuamente ante perturbaciones, variaciones del modelo o incertidumbres en la dinámica del sistema, manteniendo un control anticipativo basado en predicciones actualizadas. El comportamiento resultante mostrado en la Figura 3.3 evidencia cómo las acciones futuras (en rojo) son optimizadas para reducir la diferencia entre la trayectoria predicha (en verde) y la referencia (en naranja), garantizando una convergencia progresiva del sistema hacia el objetivo.

3.2.3. Restricciones

El *Model Predictive Control* (MPC) permite colocar restricciones sobre los estados y las entradas de control. Una de las principales ventajas de esta estrategia de control. Estas restricciones reflejan las limitaciones físicas del sistema y de actuadores, garantizando que las acciones de control generadas sean realizables y seguras dentro del dominio.

Siguiendo la formulación presentada por Camacho y Bordons [4] y adoptada por Antonelo [2], las restricciones del MPC discreto se expresan como:

$$\mathbf{u}(k+j) = \mathbf{u}(k+N_u-1), \qquad j = N_u, \ldots, N_2, \tag{3.6}$$
$$h\big(\mathbf{x}(k+j), \mathbf{u}(k+j)\big) \leq 0, \qquad j = N_1, \ldots, N_2, \tag{3.7}$$
$$g\big(\mathbf{x}(k+j), \mathbf{u}(k+j)\big) = 0, \qquad j = N_1, \ldots, N_2. \tag{3.8}$$

La ecuación (3.6) impone que, una vez alcanzado el horizonte de control N_u, el valor de la señal de control se mantiene constante hasta el final del horizonte de predicción N_2. Esto permite reducir la dimensionalidad del problema de optimización sin afectar la estabilidad del control.

Por su parte, las restricciones de desigualdad (3.7) establecen límites sobre las variables de estado y de control, tales como posiciones articulares, velocidades o torques máximos admisibles, mientras que las restricciones de igualdad (3.8) pueden representar condiciones cinemáticas o dinámicas específicas del sistema.

3.2.4. Optimización

El problema de optimización del *Model Predictive Control* (MPC) consiste en determinar, en cada instante de muestreo, la secuencia de acciones de control que minimiza la función de costo definida en la ecuación (3.4), sujeta a la dinámica del sistema y a las restricciones impuestas sobre los estados y las entradas. En su forma general, este problema se expresa como:

$$\begin{aligned} \min_{\mathbf{U}} \quad & J(\mathbf{x}(k), \mathbf{U}) \\ \text{sujeto a} \quad & \mathbf{x}(k+j+1) = f(\mathbf{x}(k+j), \mathbf{u}(k+j)), \quad j = 0, \ldots, N_2-1, \\ & \mathbf{u}_{\text{mín}} \leq \mathbf{u}(k+j) \leq \mathbf{u}_{\text{máx}}, \qquad\qquad j = 0, \ldots, N_2-1, \end{aligned} \tag{3.9}$$

donde $\mathbf{U} = \{\mathbf{u}(k), \mathbf{u}(k+1), \ldots, \mathbf{u}(k+N_u-1)\}$ es el vector de decisiones que contiene las entradas de control a optimizar.

El problema anterior constituye, en general, una *optimización cuadrática restringida* (QP) en el caso lineal, o un *problema de optimización no lineal* (NLP) cuando la dinámica $f(\cdot)$ es no lineal. En sistemas no lineales o con modelos aprendidos como los que emplean redes neuronales informadas por física (PINNs), el problema resulta no convexo y debe resolverse mediante métodos numéricos iterativos [2, 4].

Diversos enfoques pueden aplicarse para resolver el problema de optimización en línea:

- **Métodos deterministas:** tales como la Programación Cuadrática Secuencial (SQP) y los métodos de Punto Interior, ampliamente utilizados en MPC por su robustez y precisión, aunque con alta demanda computacional [4].
- **Métodos de gradiente descendente:** más simples y eficientes computacionalmente, se basan en la evaluación iterativa del gradiente del costo respecto a las variables de decisión.
- **Optimizadores adaptativos:** en entornos donde el modelo se representa mediante una red neuronal, es habitual emplear métodos derivados del aprendizaje profundo, como *Adam* o *RMSprop*, que ajustan dinámicamente la tasa de aprendizaje para acelerar la convergencia.

En cada iteración, el optimizador actualiza las variables de control según la dirección del gradiente descendente del costo total J:

$$\mathbf{U}^{(t+1)} = \mathbf{U}^{(t)} - \eta \nabla_{\mathbf{U}} J(\mathbf{U}^{(t)}), \tag{3.10}$$

donde η representa el learning rate y t el índice de iteración. El proceso continúa hasta que se cumple un criterio de parada, como la convergencia del costo o un número máximo de iteraciones, garantizando así la obtención de una solución factible dentro del intervalo de muestreo.

El algoritmo 2 ilustra de forma conceptual el flujo de optimización dentro del lazo MPC. En cada paso, el optimizador genera una secuencia de control candidata, el modelo predictivo estima la evolución del sistema, y se calcula el costo asociado. El proceso se repite de manera iterativa hasta encontrar la secuencia que minimiza el costo total bajo las restricciones establecidas.

Algorithm 2 Algoritmo general de optimización en el MPC no lineal

Entradas: estado actual $\mathbf{x}(k)$, trayectoria de referencia $\mathbf{x}^{ref}$, parámetros Q, R
Inicializar: secuencia de control $\mathbf{U}^{(0)} = \{\mathbf{u}(k), \ldots, \mathbf{u}(k+N_u-1)\}$
for $iter = 0$ **hasta** $iter_{max}$ **do**
    Predecir evolución del sistema: $\hat{\mathbf{x}}(k+i+1) = f(\hat{\mathbf{x}}(k+i), \mathbf{u}^{(t)}(k+i))$
    Calcular costo: $J^{(t)} = \sum_{i=N_1}^{N_2} \|\hat{\mathbf{x}}(k+i) - \mathbf{x}^{ref}(k+i)\|_Q^2 + \sum_{i=0}^{N_u-1} \|\Delta\mathbf{u}(k+i)\|_R^2$
    Evaluar gradiente: $\nabla_{\mathbf{U}} J^{(t)}$
    Actualizar control: $\mathbf{U}^{(t+1)} = \mathbf{U}^{(t)} - \eta\, \nabla_{\mathbf{U}} J^{(t)}$
    if criterio de convergencia cumplido o $t \geq t_{max}$ **then**
        break
    end if
end for
Aplicar primer control $\mathbf{u}(k) = \mathbf{U}^*(1)$
Avanzar un paso temporal $k \leftarrow k+1$.

Este enfoque permite que el controlador predictivo mantenga una estructura adaptable a diferentes niveles de complejidad del modelo, desde formulaciones lineales hasta representaciones no lineales aprendidas, preservando la consistencia del marco de optimización y la capacidad de operación en tiempo real.

3.2.5. Estabilidad y robustez

La estabilidad en el *Model Predictive Control* (MPC) se alcanza gracias a su formulación recursiva: en cada instante de muestreo se resuelve un problema de optimización finito, aplicando únicamente la primera acción de control y recalculando el siguiente ciclo con el estado medido. Esta estructura de realimentación continua garantiza que el sistema tienda a la referencia incluso ante pequeñas perturbaciones o errores de modelado [4, 25].

La estabilidad nominal del sistema cerrado se asegura mediante el diseño de una función de costo estrictamente positiva y la inclusión de un término terminal $V_f(\mathbf{x}(k+N))$ que actúa como función de Lyapunov, penalizando estados alejados del equilibrio. Asimismo, un horizonte de predicción suficientemente grande permite capturar la dinámica dominante del sistema y evitar comportamientos no deseados [17, 26].

En cuanto a la robustez, el MPC posee una capacidad inherente de compensación frente a incertidumbres, dado que el control se reoptimiza a cada paso a partir de las condiciones

reales del sistema. De esta forma, corrige desviaciones sin necesidad de una formulación explícitamente robusta, manteniendo factibilidad y desempeño aceptable incluso ante perturbaciones moderadas.

3.3. MPC implementado con PINCs

En formulaciones clásicas del *Model Predictive Control* (MPC), la predicción de la evolución futura del sistema depende de un modelo dinámico explícito, ya sea lineal o no lineal, obtenido a partir de ecuaciones físicas o de identificación experimental. Sin embargo, en sistemas complejos, la resolución de un modelo exacto puede resultar costoso computacionalmente. En este contexto, se incorporan las *Physics-Informed Neural Networks* (PINNs) como alternativa para modelar la dinámica del sistema de manera eficiente dentro del lazo predictivo. A diferencia de los modelos de tiempo discreto convencionales, que dependen de una discretización de las ecuaciones diferenciales (por ejemplo, mediante los métodos de Euler o Runge–Kutta), la PINN aprende una representación continua de la dinámica, y es capaz de predecir el estado futuro directamente a partir del estado actual y la acción de control aplicada. Esto elimina la necesidad de integrar paso a paso la dinámica en el horizonte de predicción.

En consecuencia, el controlador MPC con predicción basada en PINNs mantiene la estructura general del esquema predictivo, pero reemplaza el modelo físico explícito por una red neuronal informada por la física, logrando un balance entre precisión dinámica y eficiencia computacional.

En este proyecto el objetivo no es solo emplear una PINN convencional únicamente, sino un modelo extendido del tipo *Physics-Informed Neural Network for Control* (PINC) [2].

A diferencia de las PINNs tradicionales, las PINCs incorporan explícitamente la entrada de control $\mathbf{u}(t)$ dentro del conjunto de variables de entrada de la red, permitiendo representar de forma diferenciable la respuesta dinámica del sistema ante distintas acciones de control. Esto da la posibilidad de su integración directa dentro del lazo del controlador predictivo.

Como se ha comentado anteriormente, el modelo resultante recibe como entrada el estado actual $\mathbf{x}(t)$, la señal de control $\mathbf{u}(t)$ y el tiempo t, y devuelve una predicción del estado futuro $\mathbf{x}(t + \Delta t)$. De esta forma, el PINC actúa como un modelo de predicción eficiente capaz de capturar las relaciones no lineales entre las variables de estado y las acciones de control, manteniendo coherencia física a alta velocidad y con un costo computacional bajo.

En relación a la implementación con el MPC, la PINN reemplaza el modelo de predicción $f(\cdot)$ utilizado en la ecuación (3.5) por la función fordward de la red neuronal entrenada $f_\theta(\cdot)$. Esto permite mantener inalterada la estructura general del controlador, mientras el modelo dinámico es estimado mediante una aproximación aprendida:

$$\mathbf{x}(k+1) = f_\theta(\mathbf{x}(k), \mathbf{u}(k)). \tag{3.11}$$

Durante la resolución del problema de optimización, la PINC es utilizada de forma autorregresiva (como se introdujo en 2.7.1) a lo largo del horizonte de predicción. Es decir, a partir del estado actual medido $\hat{\mathbf{x}}(k)$, se encadenan M pasos de predicción sin retroalimentación del proceso real (modo *open-loop*), reutilizando en cada paso el estado predicho como condición inicial del siguiente. Esto permite generar toda la predicción futura requerida por el optimizador utilizando únicamente una propagación de red por paso. Aunque

pueden acumularse errores dentro del horizonte, el reinicio del proceso en cada iteración del lazo de control a partir del estado medido $\hat{\mathbf{x}}(k)$ garantiza que dichos errores no se propaguen en el tiempo global.

Esta integración presenta varias ventajas respecto a los enfoques basados en modelos analíticos tradicionales. En primer lugar, la red PINC es completamente diferenciable, lo que permite calcular los gradientes necesarios para la optimización del MPC de forma directa y eficiente. Además, una vez entrenada, ofrece predicciones rápidas de la dinámica en tiempo real con un costo computacional reducido. La estructura del modelo también facilita su reentrenamiento con nuevos datos, otorgándole capacidad de adaptación frente a variaciones del sistema o condiciones no modeladas, mientras que el uso de información física durante el entrenamiento garantiza una mejor generalización fuera del dominio de los datos originales.

La Figura 3.1 muestra de forma conceptual cómo la PINC se integra dentro del lazo del controlador predictivo. La red reemplaza al modelo analítico del sistema y actúa como bloque de predicción en la evaluación del costo durante la optimización. De esta manera, el MPC puede operar sobre sistemas no lineales complejos sin requerir un modelo explícito, preservando su estructura general de predicción y corrección recursiva.

3.4. Implementaciones y casos de estudio

3.4.1. Oscilador de Van der Pol

Como se detalló en el Capítulo 2, Sección 2.9, el oscilador de Van der Pol constituye un sistema no lineal de segundo orden caracterizado por una dinámica auto-oscilatoria y una fuerte dependencia del parámetro de no linealidad μ. Su comportamiento, descrito por la ecuación diferencial presentada anteriormente, lo convierte en un caso de estudio clásico para evaluar estrategias de control predictivo no lineal.

En este capítulo se aplica el controlador predictivo desarrollado a dicho sistema, utilizando inicialmente el modelo analítico del Van der Pol y, posteriormente, el modelo aprendido mediante la red PINC entrenada (véase Capítulo 2).

De esta manera, se busca validar tanto la formulación teórica del MPC como la capacidad del modelo neuronal de reproducir la dinámica del sistema y sostener un desempeño equivalente al modelo físico de referencia.

Modelo dinámico

La dinámica continua del oscilador controlado se expresa como:

$$\ddot{x} - \mu\,(1 - x^2)\,\dot{x} + x = u(t), \qquad \mu > 0, \tag{3.12}$$

donde x es la salida, μ regula la no linealidad, y $u(t)$ es la entrada de control. En forma de espacio de estados:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ \mu\,(1 - x_1^2)\,x_2 - x_1 + u \end{bmatrix} = f(\mathbf{x}, u), \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{3.13}$$

3.4.2. MPC aplicado al oscilador de Van der Pol

El objetivo es seguir una referencia $\mathbf{x}_{\text{ref}}(k)$ minimizando el error y suavizando la acción de control. Se adopta el costo con penalización sobre Δu (en línea con 3.2.2):

$$\min_{\{u_k,\ldots,u_{k+N_u-1}\}} J = \sum_{j=N_1}^{N_2} \left\|\mathbf{x}_{k+j} - \mathbf{x}_{\text{ref},k+j}\right\|_Q^2 + \sum_{i=0}^{N_u-1} \left\|\Delta\mathbf{u}_{k+i}\right\|_R^2, \tag{3.14}$$

$$\begin{aligned} \text{sujeto a:} \quad & \mathbf{x}_{k+j+1} = f(\mathbf{x}_{k+j}, \mathbf{u}_{k+j}), \quad j = 0, \ldots, N_2 - 1, \\ & \Delta\mathbf{u}_{k+i} = \mathbf{u}_{k+i} - \mathbf{u}_{k+i-1}, \quad i = 0, \ldots, N_u - 1. \end{aligned} \tag{3.15}$$

Se imponen además cotas (si aplica en el experimento):

$$u_{\text{mín}} \leq u_{k+i} \leq u_{\text{máx}}, \qquad \mathbf{x}_{\text{mín}} \leq \mathbf{x}_{k+j} \leq \mathbf{x}_{\text{máx}}. \tag{3.16}$$

El problema se resuelve iterativamente (p.ej. gradiente descendente/optimizador adaptativo), aplicando sólo el primer control u_k y re-optimizando al paso siguiente (horizonte móvil).

3.4.3. Parámetros de la implementación

Los parámetros usados en las implementaciones de prueba se resumen en la Tabla 3.1.

Tabla 3.1: Parámetros del MPC para Van der Pol (valores de la prueba)

Parámetro	Símbolo	Valor
No linealidad	μ	1.0
Tiempo de simulación	T_{sim}	20.0s
Paso de muestreo	dt	0.1s
Horizonte de predicción	N_2	9
Horizonte de control	N_u	5
Pesos de estado	$Q = \text{diag}(\cdot)$	[5, 1]
Peso de Δu	$R = \text{diag}(\cdot)$	[1]
Límites de control	$u_{\text{mín}}, u_{\text{máx}}$	[-5, 5]

Por otra parte, el optimizador utilizado para resolver el problema de minimización en los dos siguientes casos fue `Adam`.

3.4.4. Caso 1: Predicción de la dinámica mediante RK4

En este primer caso de implementación, el modelo predictivo emplea directamente la ecuación diferencial del oscilador de Van der Pol, integrada numéricamente mediante el método de Runge–Kutta de orden 4 (`RK4`). De esta manera, la predicción de la dinámica dentro del horizonte de control se realiza a partir del modelo físico exacto, sin aproximaciones.

El sistema se evaluó frente a tres referencias de posición $x_{1,\text{ref}}(t)$:

- una **constante**, $x_{1,\text{ref}}(t) = 1{,}0$,
- una **exponencial**, $x_{1,\text{ref}}(t) = 1 - e^{-0{,}3t}$,
- y una **sinusoidal**, $x_{1,\text{ref}}(t) = \sin(0{,}3t)$,

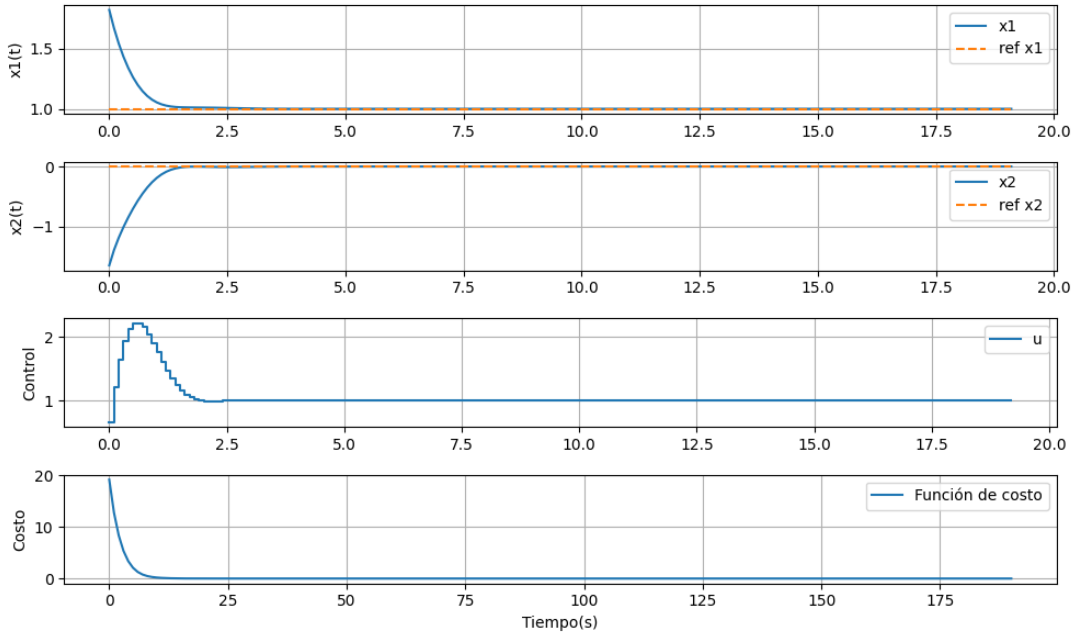


Figura 3.4: Simulación del MPC aplicado al oscilador de Van der Pol con **referencia constante**. Primer gráfico muestra la evolución temporal de la posición $x_1(t)$ del sistema (línea azul continua) junto con su referencia constante $x_{1,\text{ref}}$ (línea naranja discontinua). Segundo gráfico presenta la velocidad $x_2(t)$ (línea azul continua) y su referencia fija $x_{2,\text{ref}} = 0$ (línea naranja discontinua). Tercer gráfico representa la señal de control $u(t)$ calculada por el MPC en cada instante. Cuarto gráfico muestra la evolución de la función de costo J evaluada en cada iteración del lazo de control. La simulación corresponde a una integración de $T_{\text{sim}} = 20{,}0\,\text{s}$ con paso $dt = 0{,}1\,\text{s}$.

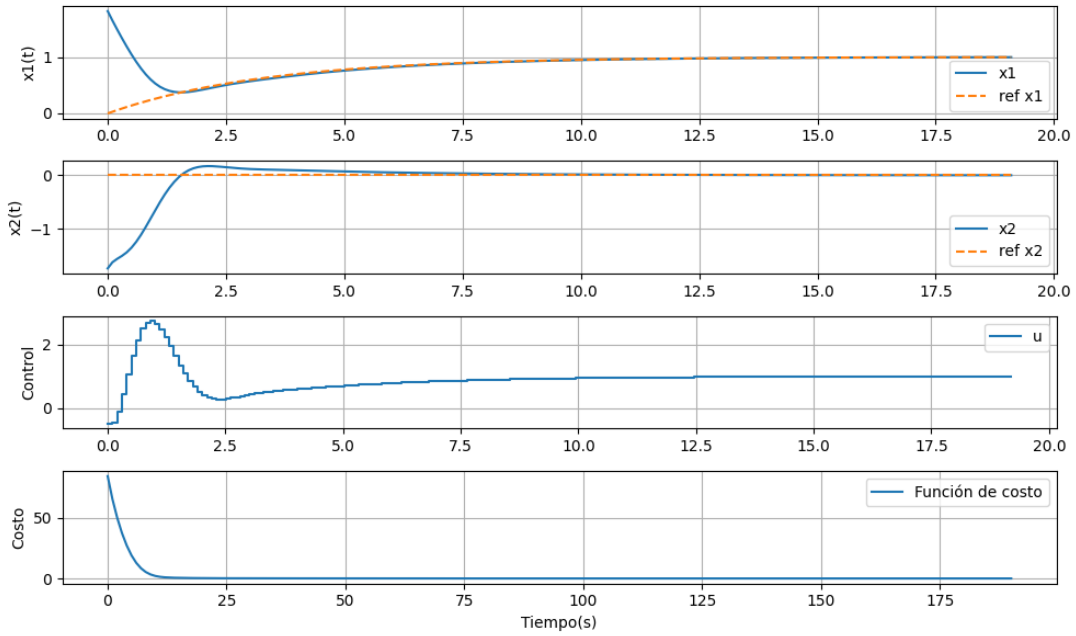


Figura 3.5: Simulación del MPC aplicado al oscilador de Van der Pol con **referencia exponencial**. Primer gráfico muestra la evolución temporal de la posición $x_1(t)$ del sistema (línea azul continua) junto con su referencia $x_{1,\text{ref}}(t) = 1 - e^{-0{,}3t}$ (línea naranja discontinua). Segundo gráfico presenta la velocidad $x_2(t)$ (línea azul continua) y su referencia fija $x_{2,\text{ref}} = 0$ (línea naranja discontinua). Tercer gráfico representa la señal de control $u(t)$ calculada por el MPC en cada instante. Cuarto gráfico muestra la evolución de la función de costo J evaluada en cada iteración del lazo de control. La simulación corresponde a una integración de $T_{\text{sim}} = 20{,}0\,\text{s}$ con paso $dt = 0{,}1\,\text{s}$.

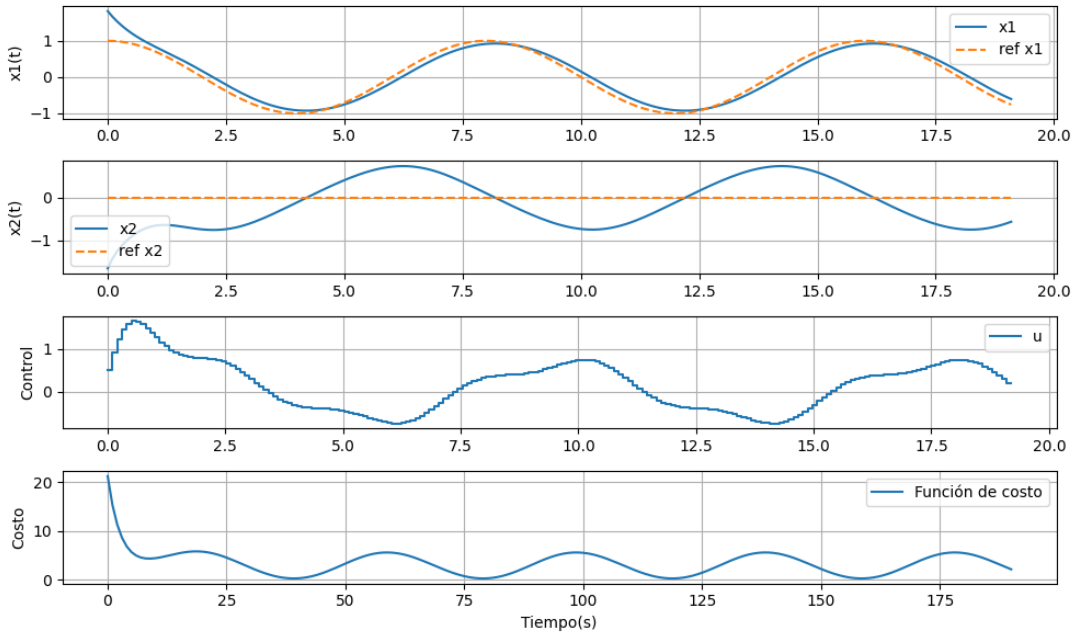


Figura 3.6: Simulación del MPC aplicado al oscilador de Van der Pol con **referencia exponencial**. Primer gráfico muestra la evolución temporal de la posición $x_1(t)$ del sistema (línea azul continua) junto con su referencia $x_{1,\text{ref}}(t) = 1 - e^{-0{,}3t}$ (línea naranja discontinua). Segundo gráfico presenta la velocidad $x_2(t)$ (línea azul continua) y su referencia fija $x_{2,\text{ref}} = 0$ (línea naranja discontinua). Tercer gráfico representa la señal de control $u(t)$ calculada por el MPC en cada instante. Cuarto gráfico muestra la evolución de la función de costo J evaluada en cada iteración del lazo de control. La simulación corresponde a una integración de $T_{\text{sim}} = 20{,}0\,\text{s}$ con paso $dt = 0{,}1\,\text{s}$.

manteniendo en todos los casos $x_{2,\text{ref}}(t) = 0$ (no se evaluó respecto a este estado).

Las Figuras 3.4–3.6 presentan las series temporales de los estados (x_1, x_2), la acción de control u y el valor del costo J para cada trayectoria de referencia. Se observa que el MPC logra un seguimiento estable y preciso, con esfuerzos de control suaves y costos decrecientes, lo que valida la implementación numérica del esquema predictivo.

Si bien los resultados son satisfactorios, se busca disminuir el tiempo de cómputo sin perder desempeño, aspecto que motiva la búsqueda de modelos predictivos más eficientes, como los basados en redes PINC.

3.4.5. Caso 2: Predicción de la dinámica mediante red PINC

En este segundo caso, el modelo predictivo del controlador se sustituyó por una red neuronal del tipo *Physics-Informed Neural Network for Control* (PINC), entrenada para aproximar la dinámica del oscilador de Van der Pol. Se evaluó el desempeño de múltiples redes PINN entrenadas variando los hiperparámetros, la cantidad de condiciones iniciales y el tipo de señal de control aplicada, donde se evaluaron dos enfoques principales:

- Un único valor de control por condición inicial.
- Varios valores de control constantes por cada condición inicial. Es decir, permitir entradas donde se repita una condición inicial pero con otro valor constante de control aplicado.

Tras analizar los resultados en términos de estabilidad, error de predicción y capacidad de generalización, se optó por los hiperparámetros de la tabla 3.2 y emplear una sola señal

de control constante por cada condición inicial como estrategia base por su rendimiento en los casos de las Figuras 3.7, 3.8 y 3.9. Las Entradas y salidas se definen en tabla 3.3.

Tabla 3.2: Hiperparámetros del modelo PINC utilizado para la predicción de la dinámica del oscilador de Van der Pol.

Hiperparámetro	**Valor**
T	1
μ	1
N_{col}	300
N° condiciones iniciales	75
N° capas ocultas	4
N° neuronas por capa	64
w_{bc}	0.5
w_{ode}	0.5

Tabla 3.3: Entradas y salida del modelo PINC utilizado para la predicción de la dinámica del oscilador de Van der Pol.

Tipo	**Descripción**
Entrada 1	Estado del sistema $\mathbf{x}(t) = [x_1(t),\, x_2(t)]^T$
Entrada 2	Señal de control aplicada $\mathbf{u}(t)$
Entrada 3	Instante de tiempo actual t
Salida	Predicción del estado futuro $\mathbf{x}(t + \Delta t)$

El controlador empleó los mismos parámetros de simulación definidos en el la tabla 3.1. Las simulaciones se realizaron bajo el mismo esquema del MPC, reemplazando la función de predicción del modelo:

$$\mathbf{x}(k+1) = f_\theta(\mathbf{x}(k), \mathbf{u}(k)), \tag{3.17}$$

donde f_θ representa la función de propagación aprendida por la red neuronal con parámetros θ.

Las Figuras correspondientes muestran los resultados obtenidos para cada tipo de referencia. El modelo PINC reproduce con prácticamente la misma precisión la dinámica no lineal del oscilador pero con una baja considerable en el tiempo de cómputo promedio por paso. Mientras que en el caso de `RK4`, cada uno de los tres casos (tres distintas referencias) tomaron un tiempo aproximado de 11 segundos, el modelo MPC con PINC logró reproducir los mismos resultados en 6 segundos, es decir, casi la mitad del tiempo.

En cualquiera de los tres casos se observa que la trayectoria de referencia es seguida de forma precisa. Sin embargo, en el caso de integración con el método de Runge-Kutta de orden 4 (RK4), el control aplicado presenta un perfil más suave y sostenido a lo largo del tiempo, con una acción significativa únicamente durante la etapa transitoria inicial. En cambio, cuando se utiliza el modelo PINC dentro del lazo de control, se observa una señal de control más persistente y fluctuante a lo largo del horizonte temporal, reflejando una necesidad continua de corrección para mantener la trayectoria deseada. Esto puede atribuirse a la naturaleza aproximada del modelo PINN, que introduce pequeñas imprecisiones acumuladas que el controlador debe compensar constantemente.

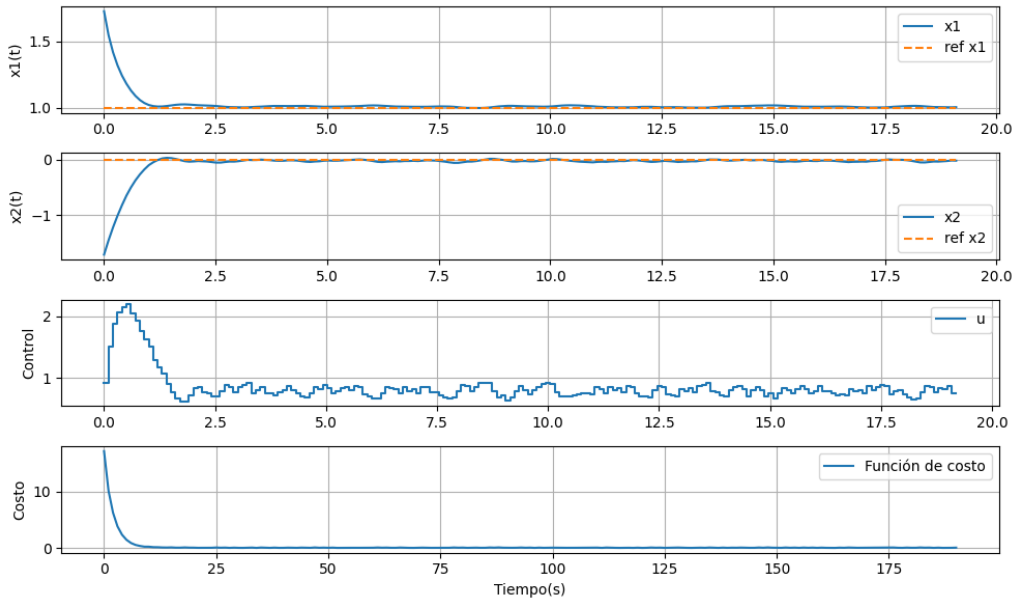


Figura 3.7: Simulación del MPC aplicado al oscilador de Van der Pol utilizando el **modelo PINC** para la predicción de la dinámica, con **referencia constante**. Primer gráfico muestra la evolución temporal de la posición $x_1(t)$ del sistema (línea azul continua) junto con su referencia constante $x_{1,\text{ref}}$ (línea naranja discontinua). Segundo gráfico presenta la velocidad $x_2(t)$ (línea azul continua) y su referencia fija $x_{2,\text{ref}} = 0$ (línea naranja discontinua). Tercer gráfico representa la señal de control $u(t)$ calculada por el MPC en cada instante, mientras que el cuarto gráfico muestra la evolución de la función de costo J evaluada en cada iteración del lazo de control. La simulación corresponde a una integración de $T_{\text{sim}} = 20{,}0\,\text{s}$ con paso $dt = 0{,}1\,\text{s}$, donde la dinámica del sistema fue predicha por el modelo neuronal $f_\theta(\mathbf{x}, \mathbf{u})$.

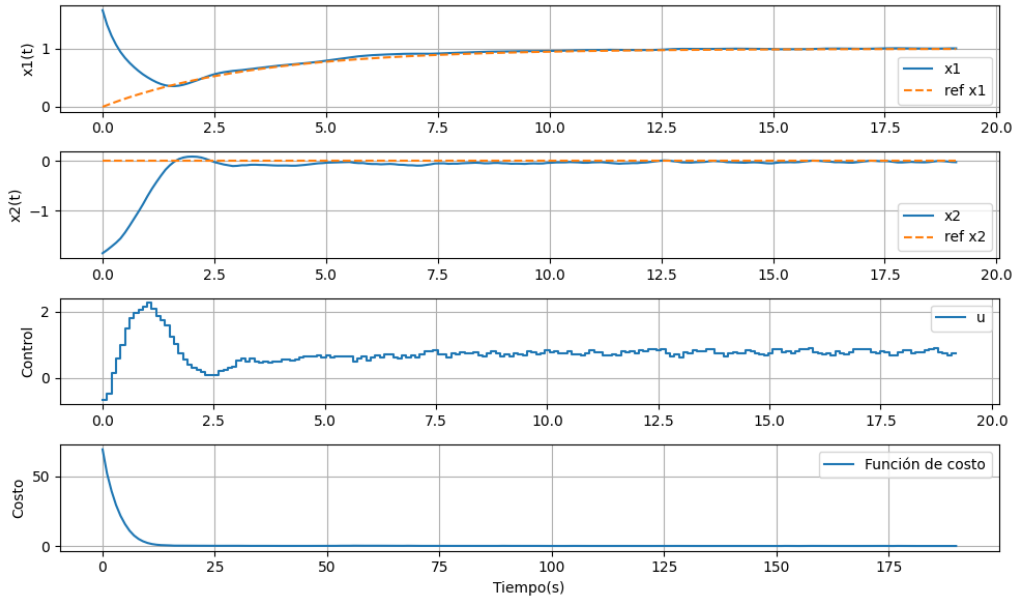


Figura 3.8: Simulación del MPC aplicado al oscilador de Van der Pol utilizando el **modelo PINC** para la predicción de la dinámica, con **referencia exponencial**. Primer gráfico muestra la evolución temporal de la posición $x_1(t)$ del sistema (línea azul continua) junto con su referencia $x_{1,\text{ref}}(t) = 1 - e^{-0{,}3t}$ (línea naranja discontinua). Segundo gráfico presenta la velocidad $x_2(t)$ (línea azul continua) y su referencia fija $x_{2,\text{ref}} = 0$ (línea naranja discontinua). Tercer gráfico representa la señal de control $u(t)$ calculada por el MPC en cada instante, y el cuarto gráfico muestra la evolución de la función de costo J a lo largo del horizonte de simulación. La simulación corresponde a una integración de $T_{\text{sim}} = 20{,}0\,\text{s}$ con paso $dt = 0{,}1\,\text{s}$, con predicción de la dinámica realizada por la red neuronal $f_\theta(\mathbf{x}, \mathbf{u})$.

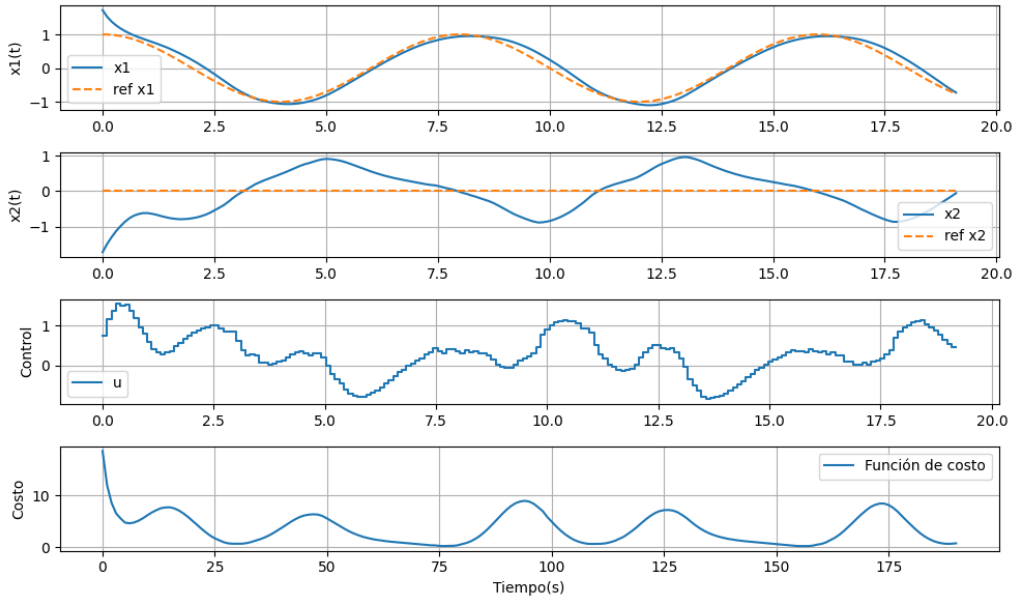


Figura 3.9: Simulación del MPC aplicado al oscilador de Van der Pol utilizando el **modelo PINC** para la predicción de la dinámica, con **referencia sinusoidal**. Primer gráfico muestra la evolución temporal de la posición $x_1(t)$ del sistema (línea azul continua) junto con su referencia $x_{1,\text{ref}}(t) = \sin(0{,}3t)$ (línea naranja discontinua). Segundo gráfico presenta la velocidad $x_2(t)$ (línea azul continua) y su referencia fija $x_{2,\text{ref}} = 0$ (línea naranja discontinua). Tercer gráfico representa la señal de control $u(t)$ calculada por el MPC en cada instante. Cuarto gráfico muestra la evolución temporal de la función de costo J durante el seguimiento periódico. La simulación corresponde a una integración de $T_{\text{sim}} = 20{,}0\,\text{s}$ con paso $dt = 0{,}1\,\text{s}$, donde la dinámica se predice mediante el modelo PINC $f_\theta(\mathbf{x}, \mathbf{u})$.

Finalmente, se observó que la ponderación $Q = \text{diag}(5, 1)$ otorgó mayor énfasis a la precisión en posición (x_1), mientras que el peso $R = 1$ limitó la magnitud de la acción de control.

En síntesis, el uso del modelo PINC permitió obtener un desempeño dinámico practicamente igual al resuelto por `RK4`, pero con una reducción significativa del costo computacional.

3.5. Limitaciones

A pesar del desempeño satisfactorio del controlador predictivo en las simulaciones realizadas, se identifican ciertas limitaciones del enfoque MPC-PINC en su implementación práctica.

En primer lugar, el costo computacional asociado al proceso de optimización sigue siendo un aspecto a considerar. En cada paso de control se resuelve un problema de minimización no lineal sujeto a restricciones, lo cual históricamente ha representado un cuello de botella para aplicaciones en tiempo real. Si bien en este caso el tiempo total de resolución para una simulación de $T_{sim} = 20{,}0\,\text{s}$ se encuentra en un intervalo de $T_{real} = [6, 10]\,\text{s}$, acercando al esquema a una posible implementación práctica en sistemas embebidos o con capacidades de cómputo moderadas, el desempeño en tiempo real dependerá del intervalo de muestreo requerido, la complejidad del modelo PINC, y la longitud del horizonte de predicción utilizado.

Por otro lado, se mantiene la **sensibilidad de los parámetros de ajuste del MPC**. Las matrices Q y R afectan directamente la estabilidad, precisión y tiempo de conver-

gencia del sistema, y su selección requiere un compromiso cuidadoso entre desempeño y esfuerzo de control. El ajuste empírico empleado en este trabajo representa una solución viable en entornos de simulación, pero puede requerir estrategias más sistemáticas (como sintonización automática o aprendizaje reforzado) para contextos reales.

Estas limitaciones deben tenerse en cuenta para la resolución futura de un sistema a controlar de mayor complejidad.

Capítulo 4

Entorno de simulación ROS2

En este capítulo se presenta el entorno de simulación empleado para el desarrollo y validación del sistema de control predictivo. El capítulo se divide en dos partes principales: en la Sección 4.1 se introducen los conceptos teóricos de ROS 2 que constituyen la base de la solución propuesta, tales como su arquitectura, los mecanismos de comunicación y los componentes fundamentales. Posteriormente, en la Sección 4.2, se describe la implementación concreta del sistema de control predictivo en ROS 2, detallando la interacción entre los distintos nodos, servicios y tópicos diseñados para el proyecto. En conjunto, este capítulo proporciona el marco conceptual y práctico necesario para comprender la integración del controlador dentro de ROS 2.

4.1. Conceptos Teóricos

En esta sección se presentan los conceptos teóricos que se consideran esenciales para comprender la solución implementada. Se introducen los principios fundamentales de ROS 2, que constituye la base del diseño de la arquitectura del sistema. Asimismo, se abordan aspectos relacionados con el problema de control, tales como la organización en nodos, la comunicación mediante tópicos y servicios, y los mecanismos de sincronización entre los diferentes componentes que integran el sistema.

4.1.1. ROS 2

La presente subsección tiene como objetivo introducir los conceptos y características principales de ROS 2, el cual constituye un requisito fundamental para la solución desarrollada en este trabajo.

El proyecto se desarrolla en el entorno del sistema operativo robótico **ROS 2 (Robot Operating System 2)**, versión **Humble Hawksbill**, una distribución con soporte extendido (LTS) mantenida hasta el año 2027 [23]. La elección de esta versión responde a la necesidad de contar con un entorno estable, ampliamente probado y con soporte garantizado a largo plazo, reduciendo así los riesgos de incompatibilidad entre paquetes y dependencias.

ROS 2 es un framework de software de código abierto y multiplataforma, ampliamente utilizado en el ámbito de la robótica. Si bien su denominación, *Robot Operating System*, podría sugerir que se trata de un sistema operativo, en realidad corresponde a un conjunto de bibliotecas, controladores y herramientas que proporcionan la infraestructura necesaria para el desarrollo de sistemas robóticos complejos.

Además, ROS 2 cuenta con el respaldo de una activa comunidad de desarrolladores e investigadores, lo que asegura un ecosistema en constante evolución, soporte extendido y la disponibilidad de una amplia variedad de paquetes y utilidades.

Al finalizar este capítulo, se espera que el lector adquiera una comprensión adecuada de las bases de ROS 2, lo que permitirá un mejor entendimiento de la solución implementada y su integración con el entorno de simulación, descrita en la Sección 5.2.

Arquitectura

ROS 2 se distingue por su flexibilidad y modularidad, cualidades sustentadas en su arquitectura en capas como se observa en la Figura 4.1 . Esta organización permite separar la lógica de la aplicación de los mecanismos de comunicación y del sistema operativo, lo que facilita el desarrollo distribuido, la integración en múltiples lenguajes y la portabilidad en diferentes plataformas [22].

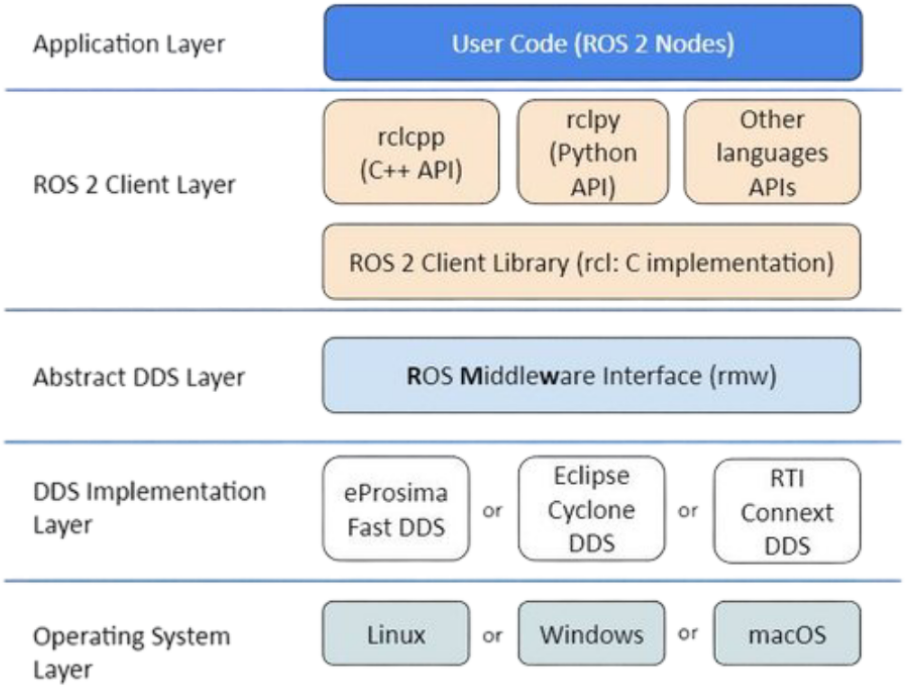


Figura 4.1: Arquitectura de ROS 2. Extraido de [31].

En el nivel superior se encuentra la **capa de aplicación**, donde se implementan los nodos que conforman el sistema. En este proyecto se incluyen, entre otros, el `trajectory_node` para la generación de trayectorias, el `mpc_controller_node` para la optimización predictiva y el `manager_node` que coordina y gestiona el ciclo de control.

Por debajo se ubica la **capa cliente**, compuesta por bibliotecas que permiten el desarrollo en distintos lenguajes. En este trabajo se emplea principalmente `rclpy` (Python), lo que asegura compatibilidad con librerías de optimización y aprendizaje automático. Esta capa encapsula la lógica de la librería central (`rcl`), garantizando un comportamiento coherente de ROS 2 independientemente del lenguaje de programación empleado.

La comunicación distribuida se gestiona a través de **DDS/RTPS** (*Data Distribution Service / Real-Time Publish-Subscribe*), una tecnología que proporciona intercambio de datos en tiempo real de manera descentralizada. ROS 2 implementa una interfaz denominada *ROS Middleware Interface* (RMW) que abstrae los detalles de cada implementación concreta de DDS, como eProsima Fast DDS o Cyclone DDS [21]. Esto representa una mejora significativa respecto a ROS 1, donde la comunicación dependía de un nodo maestro centralizado.

Finalmente, la capa inferior corresponde al **sistema operativo y hardware**, que proporciona las capacidades en tiempo real y la interacción con el entorno físico o simulado.

Componentes de ROS 2

Una de las principales características de ROS 2, que lo diferencia de otros entornos de desarrollo en robótica, es su organización modular basada en un grafo dirigido. En dicho grafo, los vértices representan nodos de cómputo y las aristas representan el flujo de información entre ellos [15,22]. Esta estructura fomenta el desarrollo de sistemas altamente escalables y reutilizables, donde las funcionalidades se encuentran desacopladas y pueden evolucionar de manera independiente. Se logra observar un ejemplo de la arquitectura en la Figura 4.2.

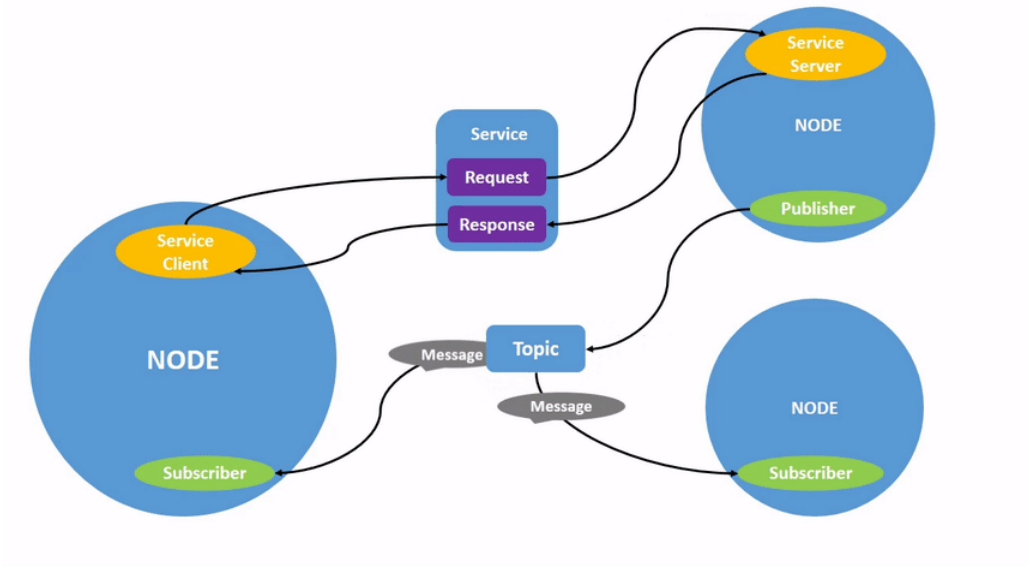


Figura 4.2: Componentes principales de ROS 2: nodos, tópicos, servicios y acciones. Extraido de [10].

Nodos

Los nodos en ROS 2 constituyen las unidades básicas de procesamiento dentro del sistema distribuido. Cada nodo representa un proceso independiente que ejecuta una o más tareas específicas, y que se comunica con otros nodos a través del middleware subyacente basado en DDS (*Data Distribution Service*) mediante interfaces bien definidas. Estas interfaces se implementan bajo tres modalidades principales: *tópicos* 4.1.1 (para comunicación asíncrona), *servicios* 4.1.1 (para comunicación síncrona) y *acciones* 4.1.1 (para tareas prolongadas con retroalimentación continua).

Desde el punto de vista del sistema operativo, cada nodo se ejecuta como un proceso separado que mantiene su propio espacio de memoria, pero comparte un dominio de comunicación común definido por el identificador de dominio DDS. ROS 2 garantiza la interoperabilidad entre nodos escritos en distintos lenguajes de programación principalmente C++ y Python. Esta arquitectura modular favorece la escalabilidad, el desarrollo distribuido y la reutilización de componentes, permitiendo estructurar sistemas robóticos complejos en unidades independientes que interactúan de manera concurrente y determinista. [15]

Tópicos

Los tópicos siguen el patrón de comunicación *publicador/suscriptor*, donde un nodo puede publicar mensajes en un canal identificado por un nombre, y múltiples nodos pueden suscribirse para recibirlos (Figura 4.3). Esta arquitectura desacoplada permite que los publicadores y suscriptores operen de manera asíncrona, sin necesidad de conocerse entre sí, lo que favorece la escalabilidad y la tolerancia a fallos del sistema. Además, ROS 2 soporta distintos perfiles de calidad de servicio (*Quality of Service, QoS*) que permiten ajustar el comportamiento de la comunicación según los requerimientos del sistema. Gracias a esta flexibilidad, los tópicos pueden adoptar configuraciones 1 a 1, 1 a N, N a 1 o N a N, facilitando la construcción de arquitecturas distribuidas y modulares [15, 22].

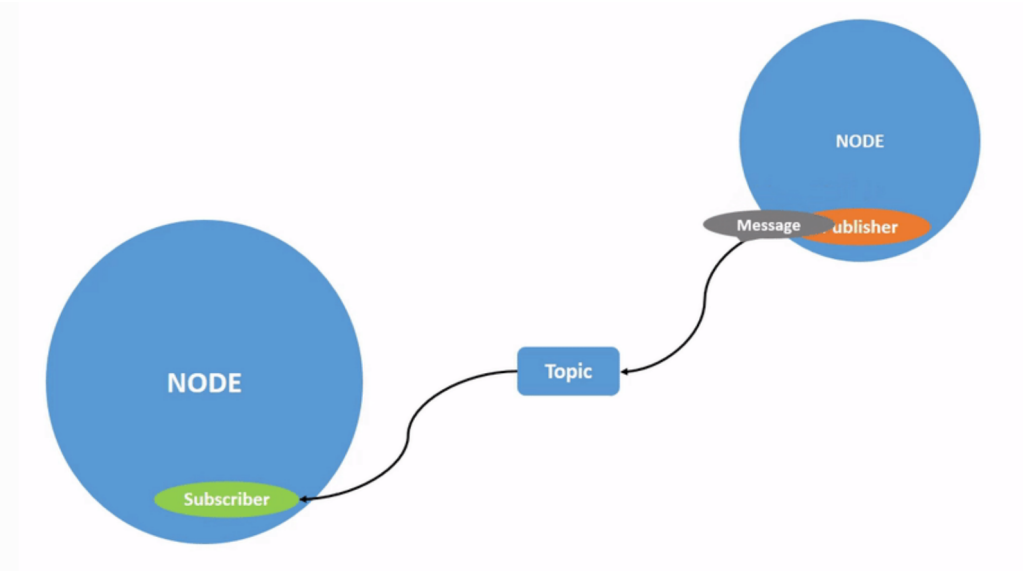


Figura 4.3: Ejemplo esquemático de tópicos en ROS 2. Extraído de [22]

Servicios

Los servicios utilizan una arquitectura *cliente-servidor*, en la cual un nodo cliente realiza una solicitud y un nodo servidor procesa la información y devuelve una respuesta (Figura 4.4). A diferencia de los tópicos, la comunicación mediante servicios es síncrona y transaccional: el cliente se bloquea hasta recibir la respuesta del servidor, lo que hace a este mecanismo particularmente útil para operaciones puntuales, determinísticas o de corta duración [15, 22]. Cada servicio se define a partir de un archivo `.srv` que especifica las estructuras de datos de la solicitud (*request*) y de la respuesta (*response*), garantizando la compatibilidad de los mensajes intercambiados entre nodos escritos en distintos lenguajes de programación. En ROS 2, las llamadas a servicios se gestionan sobre el mismo

middleware DDS, manteniendo independencia espacial y temporal entre procesos, lo que permite que clientes y servidores se ejecuten en diferentes máquinas dentro de un mismo dominio de comunicación.

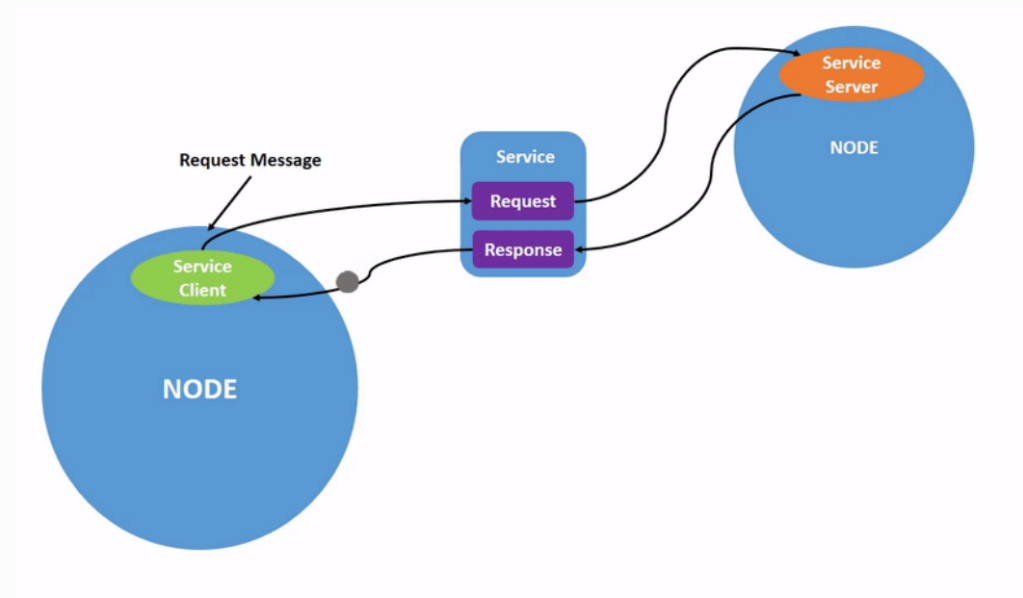


Figura 4.4: Ejemplo esquemático de servicios en ROS 2. Extraído de [22]

Acciones

Las acciones también emplean una arquitectura *cliente-servidor*, pero se diferencian de los servicios tradicionales en que incorporan un canal de *retroalimentación continua* y permiten cancelar o sobrescribir un objetivo en ejecución. Este mecanismo resulta ideal para tareas prolongadas o con duración variable, como el seguimiento de trayectorias, la manipulación de objetos o la navegación autónoma hacia un punto específico [15, 22].

Cada acción se define a través de un archivo `.action`, que contiene tres estructuras: *Goal* (objetivo solicitado por el cliente), *Feedback* (información periódica enviada por el servidor durante la ejecución) y *Result* (respuesta final una vez completada la acción). El protocolo subyacente combina dos servicios, para la recepción y confirmación del objetivo, y un tópico adicional para la transmisión de la retroalimentación, lo que permite mantener la comunicación activa durante todo el proceso. Gracias a este diseño, las acciones ofrecen un equilibrio entre control síncrono y asincrónico, garantizando flexibilidad y robustez en aplicaciones que requieren supervisión y adaptación en tiempo real.

Launchfiles y estructura de paquetes

Los sistemas desarrollados en ROS 2 suelen estar compuestos por múltiples nodos que deben ejecutarse de manera conjunta, con parámetros específicos y en un orden definido. Para facilitar esta tarea se utilizan los *launchfiles*, que permiten describir en un único archivo la configuración completa del sistema. Un *launchfile* puede estar escrito en Python, XML o YAML, y especifica qué nodos deben ejecutarse, qué parámetros se les deben asignar, qué tópicos o servicios deben remapearse, y en qué *namespace* debe operar cada proceso. Esto permite inicializar con un solo comando un sistema completo, garantizando reproducibilidad, trazabilidad y escalabilidad, especialmente en proyectos que involucran

simulación, control y visualización de forma simultánea. Además, los *launchfiles* facilitan la gestión de dependencias entre nodos y la integración con herramientas externas, como Gazebo o MoveIt, mediante el lanzamiento coordinado de sus componentes.

Por otro lado, los proyectos en ROS 2 se organizan en *workspaces*, que agrupan paquetes relacionados. Cada paquete constituye una unidad independiente y contiene los códigos fuente, recursos y configuraciones necesarias para una funcionalidad específica. Los paquetes escritos en C++ deben incluir los archivos `CMakeLists.txt` y `package.xml`, mientras que los desarrollados en Python incorporan los archivos `setup.py` y `setup.cfg`, además del directorio principal con las implementaciones de los nodos. Esta estructura modular permite desarrollar, compilar y desplegar componentes de manera aislada o colectiva dentro del mismo entorno.

En el siguiente esquema se muestra un ejemplo de organización básica de un workspace:

```
workspace/
└── src/
    ├── package1/
    ├── package2/
    ├── .../
    └── packageN/
```

Temporización y sincronización

En sistemas robóticos resulta esencial mantener una adecuada temporización y sincronización entre los diferentes procesos. ROS 2 proporciona mecanismos para gestionar el tiempo de ejecución de los nodos mediante temporizadores (`rclcpp::Timer` o `rclpy.Timer`), que permiten programar la ejecución periódica de funciones o *callbacks* a intervalos definidos. Esto resulta fundamental, por ejemplo, en el caso del bucle de control predictivo, que debe ejecutarse de manera determinista cada dt segundos, asegurando una frecuencia constante de actualización de estados y cálculo de torques.

El middleware subyacente, basado en DDS, también garantiza la inclusión de sellos temporales (*timestamps*) en cada mensaje publicado, lo que permite sincronizar la información sensorial, las trayectorias de referencia y las acciones de control. Para sistemas distribuidos, ROS 2 admite la sincronización de relojes entre nodos a través de la infraestructura de comunicación, asegurando coherencia temporal incluso cuando los procesos se ejecutan en diferentes dispositivos o simuladores.

Además, ROS 2 soporta tanto el tiempo del sistema (*system time*) como el tiempo simulado (*sim time*). Esta característica es particularmente útil en entornos de simulación como Gazebo, donde los relojes internos pueden desacoplarse del tiempo real, permitiendo reproducir y analizar el comportamiento del sistema bajo condiciones controladas. La correcta gestión del tiempo y la sincronización entre nodos garantiza que los datos sensoriales, las trayectorias de referencia y las acciones de control se encuentren alineados en el dominio temporal, evitando inconsistencias en la ejecución y asegurando la estabilidad del bucle predictivo.

Descripción del robot mediante URDF

El *Unified Robot Description Format* (URDF) es un formato basado en XML utilizado en ROS 2 para describir la estructura física y geométrica de un robot [14, 30]. Un archivo

URDF contiene la información necesaria para representar los enlaces (*links*) y articulaciones (*joints*) que conforman el modelo cinemático y dinámico de un robot. Cada enlace define su geometría, masa, propiedades inerciales y materiales, mientras que cada articulación especifica su tipo (revoluta, prismática o fija), límites de movimiento y relación entre los cuerpos conectados.

Para mejorar la legibilidad y reutilización del código, en este trabajo se emplearon archivos `xacro` (XML Macros), que permiten parametrizar componentes repetitivos y generar automáticamente el archivo URDF final mediante la herramienta `xacro.py`.

Esta metodología simplifica la modificación de parámetros como longitudes, masas o restricciones, favoreciendo una descripción modular y escalable del robot.

El archivo URDF es interpretado por distintos componentes del ecosistema ROS 2, como RViz para la visualización 3D, Gazebo para la simulación física y MoveIt para la planificación de movimiento.

En el caso del presente entorno, el modelo se utiliza como base para definir las propiedades físicas y de colisión en Gazebo, asegurando la coherencia entre el comportamiento dinámico del modelo simulado y las ecuaciones del controlador predictivo.

La siguiente Figura 4.5 muestra un fragmento de código como ejemplo simplificado de la estructura interna de un archivo URDF. Cada enlace (`<link>`) contiene las secciones `<visual>`, `<collision>` e `<inertial>`, que describen respectivamente la geometría visible del robot, la utilizada para la detección de colisiones y las propiedades dinámicas asociadas al cálculo de su movimiento. A su vez, las articulaciones (`<joint>`) definen las relaciones cinemáticas entre los enlaces, especificando los elementos padre e hijo, el eje de rotación o traslación y su tipo (revoluta, prismática o continua). Esta descripción modular permite representar con precisión la estructura física del robot y es interpretada por Gazebo, RViz y MoveIt para la simulación y el control.

```
<link name="link_name">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </visual>

  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>

  <inertial>
    <mass value="1.0"/>
    <inertia ixx="1.0" iyy="1.0" izz="1.0"/>
  </inertial>
</link>

<joint name="joint_name" type="continuous">
  <parent link="link1"/>
  <child link="link2"/>
  <axis xyz="1 0 0"/>
</joint>
```

Figura 4.5: Ejemplo simplificado de un archivo URDF que define un enlace y su articulación asociada. El enlace incluye las secciones *visual*, *collision* e *inertial*, mientras que la articulación establece la conexión cinemática con el enlace siguiente.

Entorno de simulación Gazebo

Gazebo es un entorno de simulación física de código abierto ampliamente utilizado en el ámbito de la robótica. Su principal objetivo es proporcionar un espacio virtual donde es posible modelar, visualizar y evaluar el comportamiento dinámico de robots y entornos de manera realista, antes de su implementación en un sistema físico. El simulador combina motores físicos (*physics engines*), renderizado 3D y una infraestructura de comunicación modular, lo que permite reproducir interacciones complejas entre cuerpos rígidos, articulaciones, sensores y actuadores.

Desde el punto de vista funcional, Gazebo permite simular dinámicas multirrobot, sensores inerciales, cámaras RGB-D, láseres, contactos, fricción y controladores de movimiento, utilizando modelos descritos en formatos estándar como `URDF` (*Unified Robot Description Format*) mencionado anteriormente o `SDF` (*Simulation Description Format*). Estos modelos definen la geometría, masa, inercia, articulaciones y propiedades físicas de cada componente, posibilitando una representación fiel del sistema real.

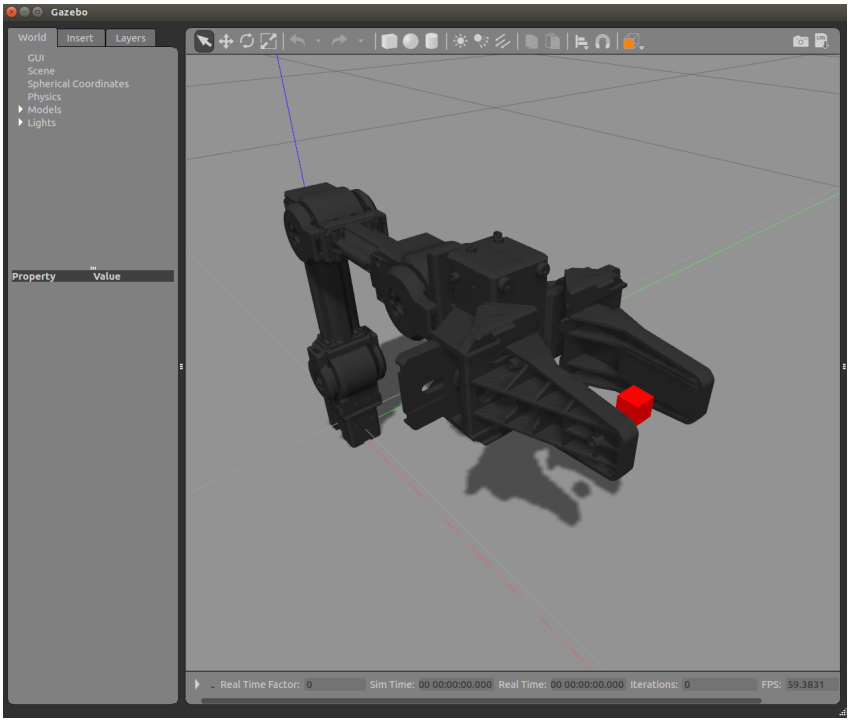


Figura 4.6: Entorno de simulación en Gazebo utilizado para las pruebas del OpenMANIPULATOR-X. Se observa el modelo del robot montado sobre una base fija dentro del mundo virtual, junto con los ejes de referencia y los marcadores visuales empleados para la validación de trayectorias. El punto rojo representa la posición cartesiana objetivo del efector final en el instante inicial de la simulación, utilizada como referencia para la resolución del problema de cinemática inversa y el inicio del lazo de control.

En la Figura 4.6 se muestra el entorno de simulación empleado para este proyecto, donde el modelo del manipulador OpenMANIPULATOR-X se encuentra montado sobre una base fija dentro del mundo virtual. Este entorno permite visualizar en tiempo real las trayectorias de referencia, los torques aplicados y las respuestas dinámicas resultantes, sirviendo como plataforma de validación para el controlador predictivo.

En el contexto de ROS 2, Gazebo se integra a través del paquete `gazebo_ros2_control`, que establece la interfaz entre los controladores ROS y los modelos simulados. Esta integración permite publicar y suscribirse a tópicos, ejecutar servicios y aplicar torques o fuerzas directamente sobre las articulaciones del robot, manteniendo coherencia completa

entre el entorno virtual y la arquitectura de control implementada. De esta forma, Gazebo actúa como una plataforma intermedia entre la teoría del control y la validación experimental, posibilitando el desarrollo, ajuste y prueba de estrategias avanzadas, como el control predictivo por modelo, en un entorno seguro y reproducible.

4.2. Control Predictivo en ROS 2

4.2.1. Arquitectura general del sistema

Diagrama de nodos

El sistema implementado se organiza en torno a un conjunto de nodos que interactúan entre sí mediante tópicos y servicios, conformando un grafo de comunicación típico de ROS 2. La Figura 4.7 presenta el esquema general, donde se representan las conexiones de información entre los componentes principales. En el diagrama se incluyen tanto los nodos propios del proyecto (`trajectory_node`, `mpc_node` y `manager_node`) como los nodos externos de soporte (`move_group` y el simulador Gazebo). Las flechas continuas indican comunicación mediante tópicos, mientras que las flechas punteadas representan interacción mediante servicios.

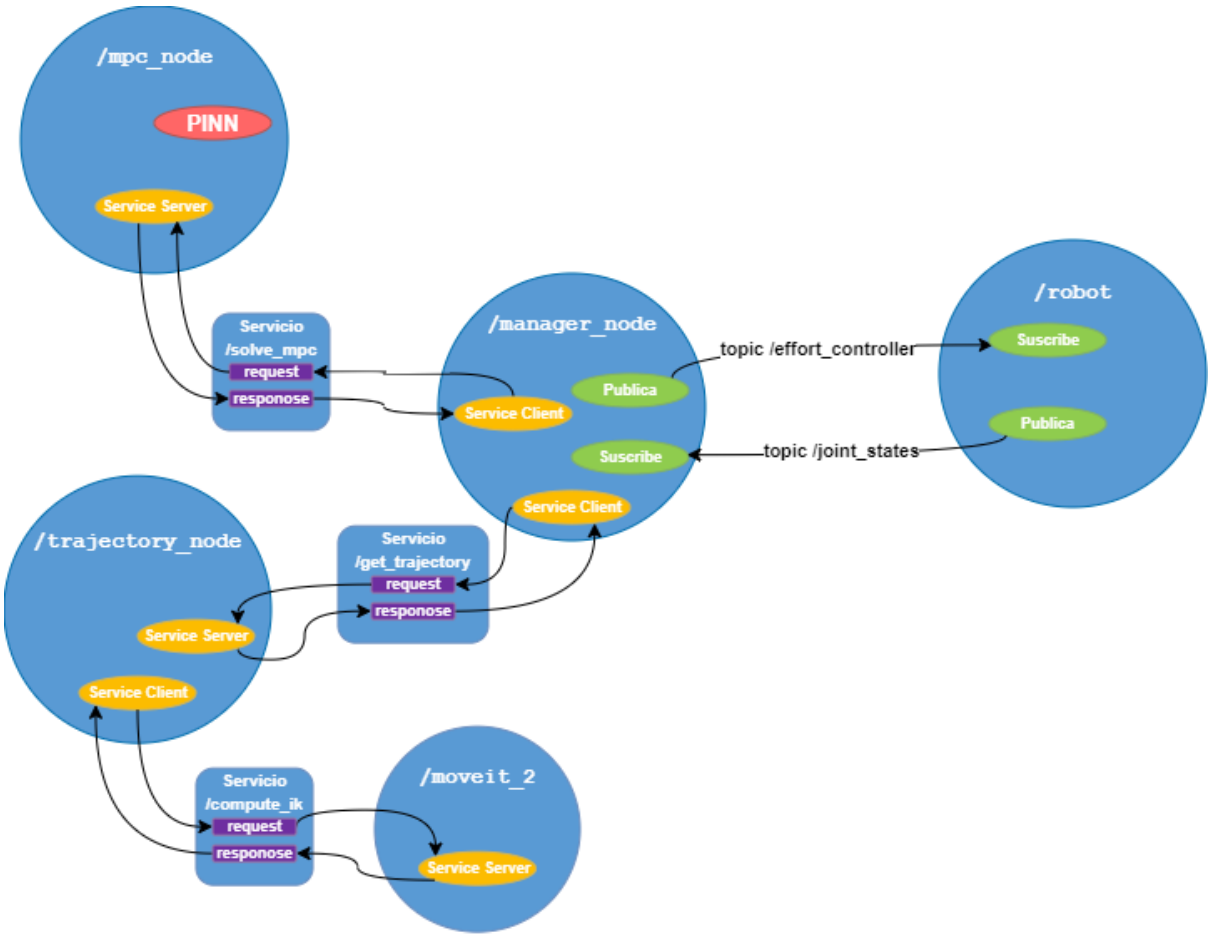


Figura 4.7: Arquitectura general del sistema de control predictivo implementado en ROS 2.

Roles de cada componente

Como se observa en la Figura 4.7, cada nodo dentro de la arquitectura desempeña un rol claramente definido:

- **trajectory_node**: genera trayectorias cartesianas semicirculares, realiza la conversión a coordenadas articulares mediante el servicio de MoveIt2, `/compute_ik`, y expone el servicio `get_trajectory_segment`, que permite obtener segmentos de trayectoria en coordenadas articulares y velocidades asociadas.
- **manager_node**: actúa como coordinador y gestor del ciclo de control predictivo. Se suscribe al tópico `/joint_states` para obtener el estado actual del robot, solicita segmentos de referencia a `trajectory_node` y solicita la resolución del control óptimo a través del servicio `solve_mpc` provisto por `mpc_node`. Finalmente, publica los torques de control en el tópico `/joint_effort_controller`. Además, dispone de servicios para iniciar y detener la simulación (`start_mpc`, `stop_mpc`).
- **mpc_node**: implementa la resolución del problema de control predictivo. Este nodo recibe el estado actual y la trayectoria de referencia, y mediante un esquema de optimización en TensorFlow obtiene la secuencia de torques óptimos. El nodo devuelve el primer control u_0 y el valor del costo asociado.
- **move_group**: proporciona el servicio `/compute_ik`, utilizado exclusivamente por `trajectory_node` para la conversión de trayectorias cartesianas a articulares.
- **Gazebo**: ejecuta la simulación dinámica del robot, publica en el tópico `/joint_states` la información de posiciones y velocidades de las articulaciones, y recibe los comandos de torque a través del tópico `/joint_effort_controller`.

4.2.2. Nodo de generación de trayectorias (trajectory_node)

El nodo `trajectory_node` tiene como función principal la generación de una trayectoria cartesiana predeterminada que servirá como referencia para el controlador predictivo. Dicha trayectoria se construye en el marco de referencia global del robot y mantiene fija la orientación del efector final, de manera que únicamente varía la posición en el espacio. Un aspecto fundamental es la **discretización uniforme en intervalos equiespaciados en el tiempo de largo dt**, lo que garantiza que todos los puntos de la trayectoria se encuentren alineados temporalmente con el periodo de muestreo definido para el lazo de control. Si bien este valor de muestreo se configuró en $dt = 0{,}1$ s, el nodo fue diseñado para admitir valores configurables, lo cual le otorga flexibilidad frente a diferentes aplicaciones.

En esta implementación, la generación de la trayectoria completa se realiza de forma previa a la ejecución del control. Este diseño responde a una decisión estratégica: el objetivo principal era centrar el esfuerzo en el desarrollo del esquema de control predictivo, utilizando un generador de trayectorias básico que permitiera validar la arquitectura completa del sistema. No obstante, se deja abierta la posibilidad de sustituir este generador por variantes más complejas, tales como trayectorias dinámicas dependientes del entorno, como el seguimiento de objetos móviles.

El funcionamiento general del nodo se resume en el Algoritmo 3, el cual detalla las principales etapas involucradas en la generación, conversión y publicación de la trayectoria articular utilizada por el controlador predictivo.

Algorithm 3 Funcionamiento general del nodo `trajectory_node`

Inicio del nodo
Cargar parámetros de configuración: dt, número de puntos N, y límites de workspace.
Generar la trayectoria cartesiana predeterminada $\{p_i\}_{i=1}^{N}$ en el marco global.
for cada punto cartesiano p_i **do**
    Enviar solicitud al servicio `/compute_ik` de MoveIt.
    if la resolución de IK es exitosa **then**
        Guardar el conjunto de articulaciones q_i correspondiente.
    else
        Asignar una configuración articular por defecto y registrar advertencia.
    end if
end for
Calcular las velocidades articulares $\dot{q}_i$ mediante diferencias finitas:

$$\dot{q}_i = \frac{q_i - q_{i-1}}{dt}$$

Construir las trayectorias discretizadas $\{q_i, \dot{q}_i\}$ consistentes con el URDF del robot.
Publicar la trayectoria cartesiana en `/reference_trajectory_marker`.
Ofrecer el servicio `get_trajectory_segment` para enviar segmentos de $\{q, \dot{q}\}$ al nodo `manager_node`.
Fin del nodo

Cinemática inversa (/compute_ik)

Como es correctamente explicado en el algoritmo 3, la trayectoria cartesiana generada debe transformarse a coordenadas articulares para poder ser seguida por el robot. Para ello, el nodo implementa un cliente del servicio `/compute_ik`, provisto por MoveIt2, que resuelve la cinemática inversa para el grupo de articulaciones del brazo. Cada pose cartesiana se envía con un tiempo máximo de resolución de dos segundos. En caso de éxito, se obtiene un conjunto de posiciones articulares; si la resolución falla, el nodo asigna una solución por defecto y notifica la condición mediante un mensaje de advertencia. Este comportamiento refleja el carácter experimental de la implementación y puede perfeccionarse en versiones futuras, por ejemplo, incorporando estrategias de redundancia o validación de colisiones.

Cálculo de trayectoria articular

Una vez obtenidas las posiciones articulares q, el nodo construye la trayectoria articular completa. Las velocidades $\dot{q}$ se calculan mediante diferencias finitas hacia atrás, con el mismo periodo de muestreo dt. De esta forma, el primer valor de velocidad se inicializa en cero, mientras que los restantes se derivan de la diferencia entre pasos consecutivos. El orden de las articulaciones se mantiene consistente con el definido en el URDF del robot, asegurando la compatibilidad con el nodo `manager_node`, que utiliza el mismo convenio al suscribirse al tópico `/joint_states`. Este procedimiento garantiza que las trayectorias discretizadas sean coherentes en tiempo y espacio, evitando inconsistencias al momento de ser consumidas por el controlador predictivo.

La trayectoria cartesiana generada por el nodo se ilustra en la Figura 4.8, donde se observa la secuencia de puntos que definen la posición deseada del efector final en el espacio de trabajo.

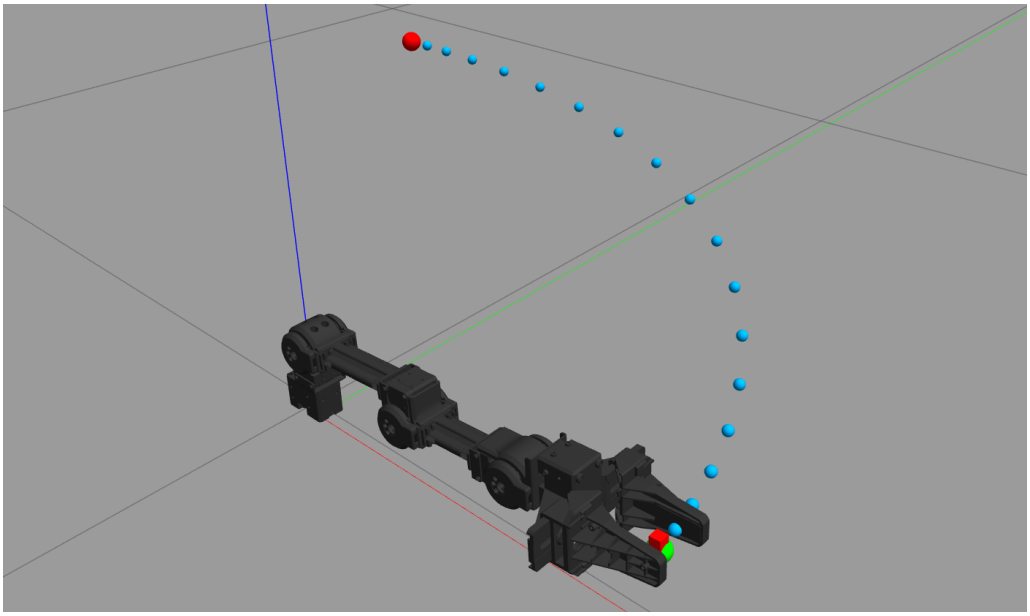


Figura 4.8: Trayectoria cartesiana predeterminada generada por el nodo `trajectory_node`. Se muestra el modelo del robot OpenMANIPULATOR-X en su posición inicial (punto verde), la posición final (punto rojo) y los puntos intermedios discretizados de la trayectoria (esferas celestes). La trayectoria describe un arco semicircular en el plano vertical, utilizado como referencia para la planificación y cálculo de la cinemática inversa.

4.2.3. Nodo de control predictivo (mpc_node)

El nodo `mpc_node` implementa el núcleo del control predictivo basado en modelo. Una de sus principales ventajas es su flexibilidad, ya que permite operar tanto con un modelo analítico de la dinámica del robot (implementado mediante **Pinocchio** el cual es explicado detenidamente en la subsección 5.3) como con una red neuronal informada físicamente (PINN). Esto lo convierte en un componente adaptable, capaz de alternar entre un modelo de referencia clásico y un modelo de aprendizaje automático según los requerimientos de la aplicación.

Cada vez que el nodo recibe una solicitud a través del servicio `solve_mpc`, se ejecuta un ciclo de optimización que busca determinar el torque óptimo a aplicar en el instante actual. El proceso completo se resume en el Algoritmo 4, que describe las principales etapas del cálculo, desde la recepción del estado actual hasta la devolución del par óptimo $(u_0, J^\star)$ al nodo `manager_node`.

El nodo devuelve siempre el mejor control encontrado hasta el momento dentro del intervalo de tiempo permitido. Los criterios de parada combinan tolerancia sobre la variación del costo, límite máximo de iteraciones y cumplimiento del presupuesto temporal impuesto por el periodo de muestreo dt, lo cual garantiza la ejecución del MPC en tiempo real.

Este diseño modular facilita el mantenimiento, dado que el nodo simplemente se encarga de la predicción del próximo paso y no del bucle de control descrito en la sección 4.2.4.

Algorithm 4 Funcionamiento general del nodo `mpc_node`

Al recibir solicitud de servicio `solve_mpc` con $(x_k, X^{\text{ref}}_{k:k+N}, dt)$
Cargar parámetros: horizontes N_2, N_u, matrices de peso Q, R, límites $(q, \dot{q}, \tau)$, tolerancias ε_J, número máximo de iteraciones $I_{\text{máx}}$ y presupuesto temporal $T_{\text{máx}} \leftarrow dt$
Inicializar secuencia de control $\mathbf{U} \leftarrow \{u_k, \ldots, u_{k+N_u-1}\}$
Inicializar mejor costo $J^\star \leftarrow +\infty$ y mejor control $\mathbf{U}^\star \leftarrow \mathbf{U}$
Seleccionar modelo de predicción $f(\cdot)$:
    ▷ **Opción A (analítico):** Dinámica forward mediante Pinocchio
    ▷ **Opción B (aprendido):** Red PINN/PINC $f_\theta(x, u)$
Iniciar cronómetro `t_start`
for $i = 1$ **hasta** $I_{\text{máx}}$ **do**
    Predecir estados sobre el horizonte N_2: $x_{k+j+1} \leftarrow f(x_{k+j}, u_{k+j})$ (mantener u_{k+N_u-1} constante para $j \geq N_u - 1$)
    Calcular incrementos de control $\Delta u_{k+j} \leftarrow u_{k+j} - u_{k+j-1}$ (con $u_{k-1} \equiv u_k$ para $j = 0$)
    Evaluar la función de costo:

$$J(\mathbf{U}) = \sum_{j=1}^{N_2} \|x_{k+j} - x^{\text{ref}}_{k+j}\|^2_Q + \sum_{j=0}^{N_u-1} \|\Delta u_{k+j}\|^2_R$$

    Aplicar restricciones: $u \in [\tau_{\text{mín}}, \tau_{\text{máx}}]$
    if $J(\mathbf{U}) < J^\star$ **then**
        $J^\star \leftarrow J(\mathbf{U})$; $\mathbf{U}^\star \leftarrow \mathbf{U}$
    end if
    Calcular gradiente $\nabla_{\mathbf{U}} J$ mediante autodiferenciación
    Actualizar controles: $\mathbf{U} \leftarrow \text{RMSprop}(\mathbf{U}, \nabla_{\mathbf{U}} J, \eta)$
    Criterios de parada:
    si $|J_{\text{prev}} - J(\mathbf{U})| < \varepsilon_J$ **entonces** detener
    si $i = I_{\text{máx}}$ **entonces** detener
    si (now − `t_start`) $> T_{\text{máx}}$ **entonces** detener
    $J_{\text{prev}} \leftarrow J(\mathbf{U})$
end for
Seleccionar control a aplicar: $u_0 \leftarrow \mathbf{U}^\star[0]$
Devolver respuesta del servicio: $(u_0, J^\star)$

Optimización de torques

La secuencia de torques óptimos se obtiene resolviendo un problema de optimización en un horizonte finito, cuya función de costo se define como:

$$J = \sum_{k=0}^{N} (x_k - x_k^{ref})^T Q (x_k - x_k^{ref}) + \sum_{k=0}^{N-1} \Delta u_k^T R \Delta u_k \tag{4.1}$$

donde x_k es el estado del sistema en el instante k, x_k^{ref} es el estado de referencia, Q es la matriz de penalización de estados, R es la matriz de penalización de variaciones de control y $\Delta u_k = u_k - u_{k-1}$.

La minimización de (4.1) se realiza de manera iterativa utilizando el optimizador `RMSprop` de Keras. Cada iteración propone una actualización de la secuencia de torques y recal-

cula el costo asociado. El proceso se detiene cuando se cumple alguna de las siguientes condiciones:

- La diferencia relativa de costo entre iteraciones consecutivas cae por debajo de un umbral predefinido.
- Se alcanza el límite de tiempo de 0,1 s, correspondiente al periodo de muestreo del sistema.
- Se completa el número máximo de iteraciones configurado para el optimizador.

Un aspecto importante es que no se selecciona necesariamente el último control calculado, sino el que haya alcanzado el menor valor de costo durante las iteraciones. De esta manera, se garantiza la elección de una solución estable y eficiente dentro de las restricciones de tiempo real.

Predicción de siguientes estados

Para la validación inicial del esquema de control y el correcto funcionamiento del nodo, la predicción de los siguientes estados se realizó empleando el modelo analítico de la dinámica del robot a través de la librería `Pinocchio`. Esta aproximación permite calcular de forma explícita las posiciones y velocidades articulares a partir de las ecuaciones dinámicas y el modelo URDF del manipulador, ofreciendo una referencia directa para la evaluación del desempeño del controlador predictivo y de la coherencia del lazo de simulación. El uso de `Pinocchio` resultó especialmente útil en las primeras etapas del desarrollo, ya que proporcionó una base fiable y computacionalmente eficiente para comparar los resultados obtenidos con el modelo aprendido.

Posteriormente, con el objetivo de integrar un modelo de aprendizaje, se implementó una red neuronal informada físicamente (PINN). Dicha red fue entrenada originalmente en `PyTorch` y posteriormente migrada a `TensorFlow/Keras` mediante la transferencia de sus pesos y arquitectura. Esta decisión respondió a que el desarrollo de la PINN y del controlador MPC se basó en fuentes bibliográficas y marcos de referencia distintos, donde las implementaciones originales empleaban librerías diferentes para el aprendizaje y la optimización, respectivamente. De este modo, se mantuvo la estructura y entrenamiento de la red en el entorno más adecuado (`PyTorch`) y, una vez finalizado el proceso, se trasladó a `TensorFlow` para integrarla sin inconvenientes con el optimizador del MPC.

La PINN recibe como entrada un vector que codifica el estado actual $(q, \dot{q})$ de las articulaciones, el instante de tiempo t, y el vector de torques aplicados τ, resultando en una dimensión de entrada acorde a $1 + (3 \cdot DOF)$. Su salida corresponde a la predicción de la posición y velocidad futuras $(q_{k+1}, \dot{q}_{k+1})$, asociadas a un intervalo de integración fijo dt, coherente con la discretización global del sistema.

De esta forma, el modelo PINN permite generar trayectorias predichas sobre múltiples horizontes temporales mediante iteraciones sucesivas de su función de propagación, reemplazando la simulación numérica explícita de mayor costo computacional, por una inferencia diferenciable de alta eficiencia, compatible con el lazo de optimización del controlador predictivo.

Comunicación del control óptimo obtenido

El nodo `mpc_node` comunica el resultado de la optimización a través del servicio `solve_mpc`. La respuesta incluye el primer torque óptimo u_0, correspondiente al instante presente, y el valor de la función de costo J. Esta información es recibida por el `manager_node`, que

se encarga de aplicar u_0 al robot publicando en el tópico `/joint_effort_controller`. De este modo, se asegura una separación clara de responsabilidades: el `mpc_node` se concentra en la resolución del problema de control predictivo, mientras que la aplicación física del torque queda a cargo del `manager_node`.

4.2.4. Nodo de control predictivo (manager_node)

El nodo `manager_node` actúa como el núcleo de coordinación del sistema de control predictivo basado en el modelo (MPC). Su función principal es integrar los distintos módulos que conforman el entorno de simulación y control, garantizando la comunicación entre los nodos de generación de trayectorias, adquisición de estados y resolución del optimizador.

En cada ciclo de ejecución, el nodo recibe el estado actual del robot proveniente del simulador o de los sensores, solicita al servicio de trayectorias el conjunto de referencias correspondientes al horizonte de predicción y utiliza estos datos para formular y resolver el problema de optimización del controlador MPC. Una vez obtenido el control óptimo, el `manager_node` aplica el primer valor de la secuencia al sistema (real o simulado), registrando la evolución de las variables y gestionando la sincronización temporal del proceso.

Este nodo constituye, por tanto, el punto de enlace entre la dinámica simulada y la lógica de optimización predictiva, asegurando la coherencia en la ejecución y el intercambio de información dentro del entorno ROS 2.

Estructura del bucle de control

El nodo `manager_node` gestiona el ciclo de control predictivo, ya que es el encargado de coordinar la interacción entre la generación de referencias, la resolución del problema de optimización y la aplicación del control al robot. El bucle de control se ejecuta de manera periódica con un tiempo de muestreo de dt, garantizando la coherencia temporal con el resto del sistema.

El bucle de control implementado en el nodo `manager_node` coordina la ejecución completa del sistema MPC dentro del entorno ROS 2, asegurando la comunicación secuencial entre la simulación, la generación de trayectorias y la resolución del problema de optimización.

El flujo completo de coordinación del lazo predictivo implementado por el `manager_node` se resume en el Algoritmo 5, que detalla las etapas de inicialización, adquisición de estado, solicitud de referencias, resolución del MPC, aplicación del control y sincronización temporal con el periodo de muestreo.

Algorithm 5 Funcionamiento general del nodo `manager_node`

Inicialización
Cargar parámetros globales: dt, T, N_{total}, N_2, N_u, Q, R, n_{joints}, n_{states}, n_{controls}, límites de torque $(u_{\text{mín}}, u_{\text{máx}})$.
Inicializar estado actual $q \leftarrow \mathbf{0}$, $\dot{q} \leftarrow \mathbf{0}$ y control $u \leftarrow \mathbf{0}$.
Crear **suscriptor** a `/joint_states`
Crear **cliente** de `/get_trajectory_segment` (servicio `GetTrajectorySegment`).
Crear **cliente** de `/solve_mpc` (servicio `SolveMPC`);
Crear **publicador** `/joint_effort_controller/commands` (`Float64MultiArray`).
Exponer **servicios** `/start_mpc` y `/stop_mpc` (tipo `Trigger`) para controlar el bucle.
Esperar a que `/solve_mpc` y `/get_trajectory_segment` estén disponibles.
Construir vector de tiempos $T_{\text{ref}} \leftarrow \text{linspace}(0, T, N_{\text{total}})$.
Al recibir `/start_mpc` → **activar** bucle
for cada índice i y tiempo t en $T_{\text{ref}}[0 : N_{\text{total}} - 1]$ **do**
    $t_{\text{ini}} = 0$
    (1) Lectura del estado desde `/joint_states`: construir $x_0 = [q^T, \dot{q}^T]^T$.
    (2) Referencia futura: invocar `/get_trajectory_segment` con (t, N_2)
    if no respuesta en **o** la respuesta es inválida **then**
        break.
    end if
    Reconvertir matrices planas a $q_{\text{ref}} \in \mathbb{R}^{(N_2+1)\times n_{\text{joints}}}$ y $\dot{q}_{\text{ref}} \in \mathbb{R}^{(N_2+1)\times n_{\text{joints}}}$.
    Construir $X_{\text{ref}} = [q_{\text{ref}},\ \dot{q}_{\text{ref}}] \in \mathbb{R}^{(N_2+1)\times n_{\text{states}}}$
    (3) Resolver MPC: invocar `/solve_mpc` con (x_0, X_{ref}).
    if no hay respuesta **o** la respuesta no contiene u_0 **then**
        Registrar advertencia/error y fijar $u_k \leftarrow \mathbf{0}$, $J \leftarrow$ NaN.
    else
        Extraer u_k y J.
    end if
    Guardar $t_{\text{solver}}[i] \leftarrow (t_{\text{actual}} - t_{\text{ini}})$ para la porción de optimización.
    (4) Aplicación de control: publicar u_k en `/joint_effort_controller`.
    (5) Temporización del ciclo: $t_{\text{dur}} \leftarrow t_{\text{actual}} - t_{\text{ini}}$
    if $dt - t_{\text{dur}} > 0$ **then**
        Esperar $dt - t_{\text{dur}}$ para respetar el periodo de muestreo.
    else
        Contabilizar sobre–tiempo (*overrun*) del ciclo.
    end if
end for
Al recibir `/stop_mpc` → finalizar bucle, liberar recursos y cerrar nodo.

El bucle se ejecuta de forma continua mientras el servicio `start_mpc` permanezca activo, y puede interrumpirse mediante el servicio `stop_mpc`. Esta arquitectura multihilo permite mantener la ejecución del lazo MPC de manera asíncrona respecto al resto de los nodos, asegurando una operación estable y sincronizada con el simulador Gazebo.

Interacción con trajectory_node

Respecto de la interacción con el nodo `trajectory_node`, como fue explicado anteriormente en 5 la misma se realiza con el objetivo de obtener la referencia futura en los siguientes N2 pasos (horizonte de predicción) a través del servicio `get_trajectory_segment`. En cada ciclo, el `manager_node` solicita un segmento de referencia a partir del tiempo actual y del horizonte de predicción configurado. El servicio devuelve matrices de posiciones y velocidades articulares discretizadas, alineadas con el paso temporal de dt. Este mecanismo desacopla la generación de trayectorias del bucle de control, permitiendo que el `trajectory_node` evolucione hacia generadores más complejos sin modificar la interfaz con el `manager_node`.

Interacción con mpc_node

El `manager_node` establece comunicación con el nodo `mpc_node` en cada iteración del bucle de control mediante el servicio `solve_mpc`. En cada llamada, se envía el estado actual del robot, formado por las posiciones y velocidades articulares, junto con la secuencia de referencia X^{ref}, que abarca el horizonte de predicción definido. El `mpc_node` procesa esta información y devuelve la respuesta compuesta por el torque óptimo inicial u_0 a aplicar en el siguiente paso de control, así como el costo total J asociado a la optimización.

La comunicación entre ambos nodos se implementa mediante mensajes en formato `float32`, lo que garantiza la compatibilidad con el entorno de ejecución basado en `TensorFlow` y la consistencia numérica de los datos a lo largo del lazo predictivo. Este intercambio periódico constituye el núcleo del esquema de control, asegurando que las decisiones del MPC se actualicen en tiempo real en función del estado actual del sistema.

Aplicación del control

Finalmente, `manager_node` publica el torque óptimo u_0 en el tópico `/joint_effort_controller`, al cual se encuentra suscrito el simulador Gazebo mediante el plugin `gazebo_ros2_control`. De esta forma, el torque calculado se aplica directamente a las articulaciones del robot. Este esquema mantiene una separación clara de responsabilidades: el `mpc_node` resuelve el problema de optimización, mientras que el `manager_node` garantiza su ejecución periódica y su aplicación sobre el modelo físico.

4.2.5. Servicios personalizados

Los servicios personalizados fueron diseñados e implementados específicamente para este proyecto con el objetivo de establecer una comunicación estructurada y eficiente entre los distintos nodos del sistema. Cada servicio define un formato de solicitud y respuesta adaptado a las necesidades particulares del lazo de control predictivo, garantizando la correcta transmisión de estados, trayectorias y comandos de control. A continuación, se describen los servicios desarrollados, detallando su estructura, propósito funcional y la forma en que intervienen dentro del ciclo de ejecución del sistema.

Definición de GetTrajectorySegment.srv

Como fue mencionado en la subsección 4.2.2, el servicio `GetTrajectorySegment.srv` fue diseñado para permitir que el `manager_node` obtenga, en cada iteración de control, un segmento de la trayectoria de referencia generada por el `trajectory_node`.

La interfaz recibe como entrada dos parámetros:

- `current_time` (float32): instante actual en segundos.
- `horizon_steps` (int32): cantidad de pasos en el horizonte de predicción.

La respuesta devuelve dos vectores aplanados en formato `float32`:

- `q_matrix`: posiciones articulares $(N+1) \times DOF$.
- `dq_matrix`: velocidades articulares $(N+1) \times DOF$.

Se garantiza que se incluyan siempre $N+1$ muestras (desde el presente hasta el horizonte solicitado). En caso de no existir suficientes muestras, se repite el último valor válido. Esto asegura que el controlador predictivo disponga de datos consistentes en cada ciclo.

Definición de SolveMPC.srv

El servicio `SolveMPC.srv` explicado en 4.2.3 constituye la interfaz principal del `mpc_node`. Su propósito es resolver el problema de control predictivo a partir del estado actual y de la trayectoria de referencia.

La entrada del servicio incluye:

- `current_state` (float32[]): vector de tamaño $2 \cdot DOF$ con posiciones y velocidades articulares actuales.
- `ref_trajectory` (float32[]): matriz aplanada con la trayectoria de referencia en posiciones y velocidades para todo el horizonte.

La salida devuelve:

- `u0` (float32): primer torque óptimo calculado.
- `cost` (float32): valor de la función de costo optimizada.

De esta manera, se mantiene la separación entre la resolución del problema predictivo (realizada en `mpc_node`) y la aplicación física de la acción de control (realizada en `manager_node`).

Definición de StartMPC.srv

El servicio `StartMPC.srv` complementado con `StopMPC.srv` controla la inicialización del del ciclo de control dentro del `manager_node`. A través de esta interfaz, el sistema puede inicializar o detener la simulación del lazo MPC.

La implementación se basa en una arquitectura sencilla de tipo *trigger*, donde el servicio de inicio lanza un hilo de ejecución que ejecuta el bucle de control, mientras que el servicio de detención activa una bandera de parada. Este esquema desacopla la inicialización del lazo de control de la configuración de los nodos, lo que simplifica las pruebas y la integración en entornos simulados.

Detalles de diseño e implementación

Cada servicio fue diseñado siguiendo principios de simplicidad y modularidad, pero también con objetivos específicos dentro de la arquitectura:

- **StartMPC.srv**: se implementó para permitir que el bucle de control predictivo se inicie de manera explícita cuando se requiera. De esta forma, con una única llamada al servicio desde código o consola, es posible comenzar la ejecución del lazo MPC sin necesidad de reiniciar nodos ni lanzar nuevamente la simulación.
- **GetTrajectorySegment.srv**: se diseñó con un carácter escalable. Si bien en la implementación actual la trayectoria es predeterminada y se calcula al inicio, la interfaz del servicio permite que en el futuro el `trajectory_node` pueda ser actualizado para entregar trayectorias dinámicas, ajustadas en tiempo real en función de estímulos externos o de objetivos cambiantes. Así, el contrato de comunicación se mantiene inalterado, mientras que la complejidad del generador de trayectorias puede crecer sin afectar al resto del sistema.
- **SolveMPC.srv**: su objetivo principal es independizar la resolución del problema de control predictivo del bucle de gestión. De esta manera, el `mpc_node` se encarga exclusivamente de la optimización, mientras que el `manager_node` se limita a orquestar la secuencia de pasos y aplicar los torques.

Este diseño modular asegura que cada servicio pueda evolucionar de manera independiente, preservando la coherencia de la arquitectura y favoreciendo la integración de mejoras futuras.

4.2.6. Consideraciones de diseño

Modularidad

El sistema se diseñó siguiendo un enfoque modular como se observa en la Figura 4.7, en el cual cada nodo cumple una función específica y bien delimitada: generación de trayectorias, resolución del problema de control predictivo, coordinación y gestión del ciclo de control, y simulación física. Esta separación de responsabilidades permite aislar los posibles errores, facilita el proceso de depuración y posibilita la sustitución o mejora de componentes individuales sin alterar el resto de la arquitectura.

Fallos de IK

El cálculo de cinemática inversa, implementado en `trajectory_node` mediante el servicio `/compute_ik` de MoveIt2, puede fallar en situaciones donde la pose deseada se encuentra fuera del espacio de trabajo alcanzable o en configuraciones cercanas a singularidades. En la versión actual, estos fallos se gestionan mediante la asignación de una solución por defecto y la emisión de advertencias en el registro.

Un problema adicional se relaciona con la coherencia de las soluciones: al no imponerse restricciones de continuidad, dos soluciones consecutivas de cinemática inversa pueden diferir de manera significativa, generando trayectorias articulares poco realistas o con saltos bruscos. Para mitigar este efecto, el sistema incorpora verificaciones que permiten identificar discrepancias marcadas entre soluciones consecutivas, asegurando que la trayectoria resultante sea suave y físicamente consistente. Este aspecto se reconoce como un área de mejora futura, donde podrían integrarse algoritmos de optimización que seleccionen la

solución más cercana a la configuración previa o mecanismos de validación basados en restricciones dinámicas.

Frecuencia de operación

El ciclo de control predictivo se diseñó para operar con un tiempo de muestreo de $dt = 0{,}1$ s (10 Hz), acorde a los requerimientos del robot simulado. El nodo `manager_node` regula la temporización de cada iteración, de modo que si las operaciones se completan antes del periodo de muestreo, el sistema espera hasta alcanzar el tiempo correspondiente antes de iniciar el siguiente ciclo.

El nodo `trajectory_node` genera inicialmente toda la trayectoria de referencia y provee al `manager_node` únicamente el segmento correspondiente al horizonte de predicción solicitado. Esta organización permite mantener escalabilidad en el futuro, habilitando la posibilidad de trayectorias dinámicas o dependientes de estímulos externos, como el seguimiento de objetos en movimiento.

El nodo `mpc_node` resuelve la optimización empleando el optimizador `RMSprop` de Keras/TensorFlow. Si bien el problema de optimización puede resultar costoso computacionalmente, se ajustaron los horizontes de predicción y los parámetros del optimizador de manera tal que el cálculo de u_0 se encuentra acotado dentro del intervalo de muestreo establecido. De esta manera, el sistema logra mantener la ejecución del ciclo de control en tiempo real dentro de la simulación.

Sincronización temporal

La coherencia temporal entre nodos constituye un aspecto crítico en el lazo MPC. Todo el sistema fue configurado para operar con un periodo de muestreo fijo de $dt = 0{,}1$ s, lo que garantiza que la discretización de trayectorias, la ejecución del optimizador y la aplicación de torques se mantengan alineadas. El `manager_node` regula la temporización de cada iteración, asegurando que, incluso cuando los cálculos finalizan antes del tiempo límite, el ciclo no se adelanta. De este modo se preserva la estabilidad del sistema y se evita la acumulación de errores de sincronización a lo largo de la simulación.

Gestión centralizada de parámetros

El sistema utiliza un archivo global de configuración, `parameters.py`, que define los parámetros principales del controlador y del entorno de simulación. Este módulo es importado por todos los nodos y contiene los valores de muestreo (dt), horizontes (N_2, N_u), matrices de costo (Q, R), límites de torque, así como las opciones de ejecución y visualización en Gazebo y Meshcat. La gestión centralizada de parámetros garantiza la coherencia entre nodos, facilita la replicabilidad de experimentos y permite modificar el comportamiento del sistema desde un único punto de configuración.

Escalabilidad y mantenibilidad

Además de modular y consistente en tiempo, la arquitectura fue diseñada con criterios de escalabilidad. El `trajectory_node`, por ejemplo, expone un servicio que en el futuro podrá entregar trayectorias dinámicas generadas en línea, sin necesidad de modificar la interfaz de comunicación. El `mpc_node`, a su vez, permite intercambiar el modelo de predicción entre un modelo analítico (Pinocchio) y una PINN, sin que ello afecte la lógica

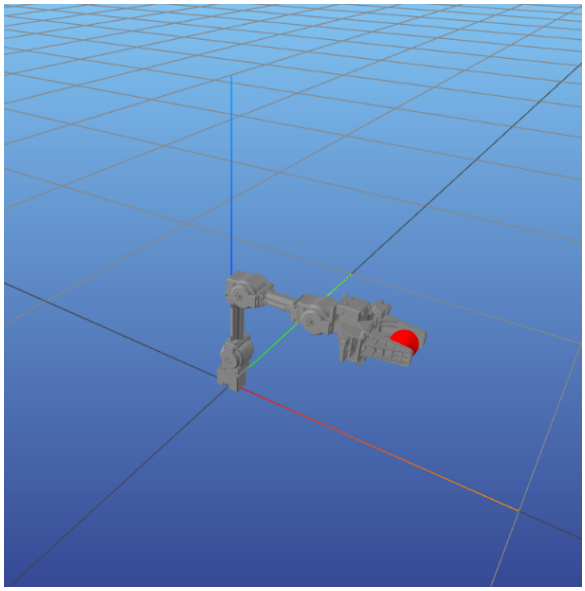
de orquestación del `manager_node`. Estas decisiones de diseño aseguran que el sistema pueda crecer en complejidad, incorporar nuevas funcionalidades o adaptarse a hardware real manteniendo la misma base de comunicación.

4.2.7. Validación en ROS 2

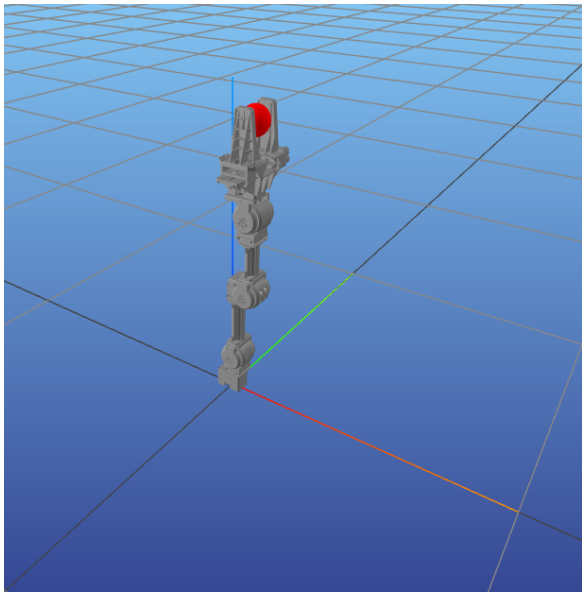
Para garantizar el correcto funcionamiento de la arquitectura, se llevaron a cabo pruebas de comunicación diferenciadas según la interfaz utilizada (servicio o tópico).

Pruebas en /compute_ik:

La validez del servicio de cinemática inversa se verificó analizando la coherencia de las soluciones devueltas. Se realizaron pruebas en configuraciones de referencia, como la posición neutral del brazo y el estiramiento completo de las articulaciones como se muestra en la Figura 4.9. Con la ayuda de la herramienta Meshcat, se visualizó en tiempo real la respuesta del solver de IK, confirmando que las soluciones eran físicamente plausibles.



Brazo en posición neutral.



Brazo completamente extendido.

Figura 4.9: Validación del servicio `/compute_ik` en *Meshcat*. En cada imagen, la esfera roja indica la posición cartesiana objetivo del efector final. La coincidencia entre el marcador y la posición alcanzada confirma la correcta resolución del solver `/compute_ik`.

Pruebas en /get_trajectory_segment:

La validación del servicio implementado en el nodo `trajectory_node` se realizó inspeccionando el tamaño y consistencia de las listas devueltas en cada llamada al serivicio. Los segmentos generados se graficaron con la ayuda de Meshcat, lo que permitió confirmar la discretización uniforme en $dt = 0{,}1$ s y la coherencia entre posiciones y velocidades articulares.

Tabla 4.1: Comparación entre valores ideales y obtenidos del servicio `/compute_ik`.

Configuración	**Referencia**	**Resultados**
Posición neutral	Pos. cartesiana (m)	(0.274, 0.000, 0.205)
	q_1 [rad]	Ideal: 0.0000 Obtenido: 0.0002
	q_2 [rad]	Ideal: 0.0000 Obtenido: −0,0018
	q_3 [rad]	Ideal: 0.0000 Obtenido: 0.0015
	q_4 [rad]	Ideal: 0.0000 Obtenido: −0,0080
Brazo extendido	Pos. cartesiana (m)	(0.024, 0.000, 0.455)
	q_1 [rad]	Ideal: 0.0000 Obtenido: 0.0006
	q_2 [rad]	Ideal: −1,5708 Obtenido: −1,5684
	q_3 [rad]	Ideal: 0.0000 Obtenido: 0.0011
	q_4 [rad]	Ideal: 0.0000 Obtenido: −0,0013

Cada configuración fue validada visualmente en *Meshcat*, verificando que el efector final alcanzara la posición cartesiana objetivo. Las ligeras variaciones respecto a valores ideales corresponden al error numérico del solver de cinemática inversa.

Pruebas en /solve_mpc:

El servicio ofrecido por el `mpc_node` se probó comparando los datos enviados y recibidos en los logs. Se verificó que el vector de estado inicial y la trayectoria de referencia coincidieran en tamaño y formato con lo esperado, y que la respuesta incluyera tanto el torque inicial u_0 como el costo de la optimización.

Pruebas en /joint_effort_controller:

La validación de la publicación de torques resultó más compleja, dado que el modelo del OpenManipulator-X debió ser adaptado para aceptar control por esfuerzo. Como es explicado en profundidad más adelante, fue necesario modificar la configuración de los controladores y la descripción del robot para habilitar esta modalidad. Una vez realizada la adaptación, se comprobó que los torques enviados por el `manager_node` eran correctamente interpretados por el simulador, aplicándose de forma consistente sobre las articulaciones y lográndose ver en gazebo.

Logs de operación

La validación del sistema se apoyó fuertemente en los registros generados por los distintos nodos durante la ejecución. Estos *logs* constituyen una fuente esencial de información, ya que permiten analizar el comportamiento temporal del sistema, detectar inconsistencias en la comunicación y evaluar el desempeño computacional de cada componente.

El proceso de depuración (*debug*) en ROS 2 presenta una complejidad considerable, dado que la ejecución simultánea de múltiples nodos, servicios y tópicos distribuidos puede dificultar la identificación del origen de un error o de un comportamiento inesperado. Por esta razón, los mensajes de registro adquieren un rol importante durante la integración del sistema completo, al proporcionar una trazabilidad detallada de las operaciones realizadas y los tiempos asociados a cada paso del ciclo predictivo.

A modo de ejemplo, como se muestra en la Figura 4.10, los mensajes generados por el `manager_node` documentan la secuencia completa de eventos en el lazo de control incluyen-

do la solicitud de trayectorias, la resolución del MPC y la aplicación de torques junto con los tiempos medidos en cada iteración. Por su parte, el `mpc_node` y el `trajectory_node` reportan los resultados de optimización y la validez de los segmentos de referencia, respectivamente, permitiendo una trazabilidad integral del ciclo de control predictivo.

```
[manager_node-4] [INFO] [1761621583.031240795] [manager_node]: Iniciando nodo Manager...
[manager_node-4] [INFO] [1761621583.295418409] [manager_node]: solve_mpc visible? False
[manager_node-4] [INFO] [1761621583.301620939] [manager_node]: Nodo MPC listo. Llamá /start_mpc para iniciar.
[manager_node-4] [INFO] [1761621583.301882690] [manager_node]: Nodo MPC listo. Esperando start...
[mpc_node-3] [INFO] [1761621584.396890509] [mpc_node]: Nodo MPC iniciado
[mpc_node-3] [INFO] [1761621584.397207579] [mpc_node]: Backend: usando PINN.
[mpc_node-3] [INFO] [1761621584.583501222] [mpc_node]: [MPC] PINN TF cargada. layers=[7, 128, 128, 128, 128, 4], prefix=backbone.linears
[mpc_node-3] [INFO] [1761621584.584061641] [mpc_node]: [PINN] var dtypes: ['float32']
[mpc_node-3] [INFO] [1761621584.587212105] [mpc_node]: SolveMPC server listo en /solve_mpc (ns=/)
[mpc_node-3] [INFO] [1761621584.588012853] [mpc_node]: Nodo MPC listo para recibir solicitudes.
[trajectory_node-2] [INFO] [1761621587.923637793] [trajectory_node]: [traj] Marcadores spawneados en Gazebo.
[trajectory_node-2] [INFO] [1761621587.923897784] [trajectory_node]: [FK] frame=world link=end_effector_link joints=['joint1', 'joint2']
[trajectory_node-2] [INFO] [1761621587.924118452] [trajectory_node]: Servicio listo.
[manager_node-4] [INFO] [1761621654.790889481] [manager_node]: Iniciando loop MPC...
[manager_node-4] [INFO] [1761621654.869584985] [manager_node]: Paso 1/200 - Tiempo: 0.00s
[manager_node-4] [INFO] [1761621654.902911528] [manager_node]: [Mgr] Paso 1: 83.9 ms
[manager_node-4] [INFO] [1761621654.906408380] [manager_node]: [Mgr] GetTrajectorySegment en 3.1 ms
[manager_node-4] [INFO] [1761621654.906860286] [manager_node]: [Mgr][dq] ||srv||=9.32e-02 (max=3.84e-02) | ||fd||=9.27e-02 (max=3.71e-02)
[manager_node-4] [INFO] [1761621654.907410874] [manager_node]: [Mgr] Paso 2: 115.1 ms
[mpc_node-3] [INFO] [1761621657.105490890] [mpc_node]: [opt] J0=3.799e+04 Jf=3.799e+04 dJ=+0.000e+00 iter=1 t=2195.5 ms
[manager_node-4] [INFO] [1761621657.108429357] [manager_node]: [Mgr] /solve_mpc en 2200.2 ms | u0=[-0.032 -0.782]
[manager_node-4] [INFO] [1761621657.108734188] [manager_node]: [Mgr] Paso 3: 2316.6 ms
[manager_node-4] [INFO] [1761621657.109096376] [manager_node]: [Mgr] Paso 4: 2317.0 ms
[manager_node-4] [INFO] [1761621657.109325900] [manager_node]: [Mgr] Paso 5: 2317.2 ms
[manager_node-4] [INFO] [1761621657.109591413] [manager_node]: [Mgr] Paso del loop en 2317.5 ms
[manager_node-4] [INFO] [1761621657.109789976] [manager_node]: [Mgr] Paso 6: 2317.7 ms
```

Figura 4.10: Ejemplo de registro de operación (*log*) durante la ejecución del lazo MPC. Se observan los mensajes de inicialización, tiempos de servicio de cada nodo, y torques óptimos calculados por el `mpc_node`. Estos registros permiten verificar la coherencia del flujo de ejecución y la estabilidad temporal del sistema.

En el `trajectory_node`, los registros reportan la correcta construcción de la trayectoria articular, incluyendo el número de pasos generados, el número de grados de libertad, la detección de valores inválidos (`NaN`) y las configuraciones iniciales y finales de la trayectoria. Además, en cada intento de cinemática inversa se registran advertencias cuando el solver no encuentra una solución válida, lo que permite identificar rápidamente posibles problemas de alcance o singularidad.

En el ciclo MPC, coordinado por el `manager_node`, se registran los pasos de la simulación junto con la marca temporal correspondiente. Cada interacción con `get_trajectory_segment` informa el tiempo de respuesta y el tamaño de las matrices recibidas. Posteriormente, la llamada al servicio `solve_mpc` reporta el tiempo de resolución de la optimización, el torque inicial u_0 calculado y el costo de la función objetivo. Estos datos permiten cuantificar de manera precisa el desempeño del sistema y confirmar que la resolución del problema de control se encuentra dentro de los márgenes temporales estipulados de 0,1 s por ciclo.

En conjunto, los logs constituyen una herramienta clave para la depuración y validación, asegurando que los resultados numéricos, los tiempos de cómputo y el comportamiento global del sistema se encuentren alineados con los objetivos de diseño.

4.3. Simulación en Gazebo

4.3.1. Entorno de simulación

El entorno de simulación se implementó en Gazebo, integrado con ROS 2 Humble, con el objetivo de reproducir de manera realista la dinámica del robot OpenManipulator-X. Este simulador permite modelar tanto las propiedades físicas de los actuadores como la interacción con el entorno, considerando gravedad, fricción y contactos. La utilización de Gazebo, junto con los paquetes de control provistos por `gazebo_ros2_control`, facilita

la experimentación en un entorno seguro y reproducible, antes de trasladar la solución al hardware físico.

4.3.2. Modelado del robot

El modelo del manipulador empleado en la simulación se basa en los archivos oficiales del *OpenMANIPULATOR-X*, provistos por el fabricante ROBOTIS [29]. A partir de esta base, se obtuvo un modelo URDF (explicado en la sección 4.1) derivado de macros `xacro`, que define la geometría de los enlaces, las articulaciones y los parámetros dinámicos asociados (masas, momentos de inercia, límites de posición y par). El modelo incluye los cuatro grados de libertad principales y el efector final, reproduciendo la estructura cinemática completa del brazo real.

Asimismo, se incorporaron términos de:

- Fricción viscosa.
- Estática en cada articulación.
- Límites máximos de torque.

de acuerdo con las especificaciones del actuador DYNAMIXEL XM430-W350-T.

Estas modificaciones permiten que el comportamiento dinámico observado en Gazebo sea coherente con la respuesta física del hardware real, garantizando una representación fiel del sistema mecánico y su interacción con el entorno.

En la Figura 4.11 se muestra el modelo del robot cargado en Gazebo, donde pueden observarse las longitudes de los eslabones, la disposición de las articulaciones y los ejes de rotación correspondientes.

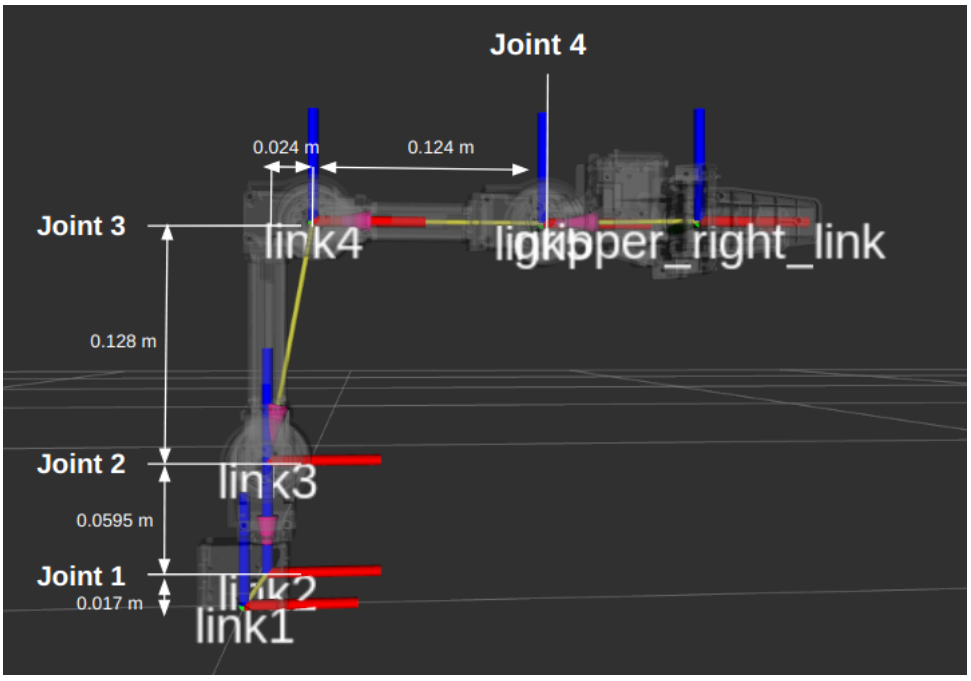


Figura 4.11: Modelo del OpenMANIPULATOR-X en Gazebo. Se muestran las articulaciones (*Joint 1–4*), los enlaces estructurales y las longitudes principales del manipulador. Los vectores en color azul indican los ejes de rotación, mientras que las cotas reflejan las distancias entre los centros de articulación. Este modelo URDF fue extendido con parámetros dinámicos y de fricción para mejorar la fidelidad física en la simulación. Extraído de [29].

4.3.3. Configuración y paquetes de OpenManipulator-X simulado

Para la integración del OpenManipulator-X en Gazebo fue necesario realizar varias modificaciones sobre los paquetes originales y generar nuevos archivos de configuración orientados al control por esfuerzo. Estas adaptaciones permitieron que el esquema MPC pudiera interactuar correctamente con el simulador.

Archivo YAML de effort_controllers: Se ha definido un nuevo archivo `.yaml` específico para definir un controlador de esfuerzos. Dicho archivo establece los parámetros de cada articulación. De esta manera, es posible sustituir el control por posición que traen los paquetes originales por controladores capaces de aceptar valores de torque publicados en el tópico `/joint_effort_controller/commands` a continuación se meustra el archivo donde se define el controlador.

```yaml
controller_manager:
  ros__parameters:
    use_sim_time: true
    update_rate: 1000 # Hz

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    joint_effort_controller:
      type: effort_controllers/JointGroupEffortController

joint_effort_controller:
  ros__parameters:
    use_sim_time: true
    joints:
      - joint1
      - joint2

    command_interfaces:
      - effort
    state_interfaces:
      - position
      - velocity
      - effort
```

Figura 4.12: Esquema de configuración YAML del controlador de esfuerzos. Se muestra un fragmento del archivo `omx_effort_controllers.yaml`, donde se definen los parámetros principales del controlador por torque, las articulaciones activas. Esta configuración es cargada automáticamente por `ros2_control` al iniciar la simulación en Gazebo.

URDF con parámetros de fricción y torque máximo: En primer lugar, el modelo URDF original fue modificado para incluir parámetros de fricción en las articulaciones. De esta manera se logró reproducir de forma más realista la disipación de energía y la resistencia al movimiento dentro del simulador Gazebo.

Por otro lado, se actualizaron los valores de esfuerzo máximo (τ_{max}) que cada articulación puede ejercer. Estos valores fueron extraídos de las especificaciones reales de los servomotores Dynamixel XM430-W350 [28], permitiendo que el modelo simulado se acerque al comportamiento esperado en el hardware real.

```
<joint name="${prefix}joint1" type="revolute">
  <parent link="${prefix}link1"/>
  <child link="${prefix}link2"/>
  <origin xyz="0.012 0.0 0.017" rpy="0 0 0"/>
  <axis xyz="0 0 1"/>
  <limit velocity="4.8" effort="3.5" lower="${-pi}" upper="${pi}" />
  <dynamics damping="0.1" friction="0.008"/>

</joint>

<link name="${prefix}link2">···
</link>

<joint name="${prefix}joint2" type="revolute">
  <parent link="${prefix}link2"/>
  <child link="${prefix}link3"/>
  <origin xyz="0.0 0.0 0.0595" rpy="0 0 0"/>
  <axis xyz="0 1 0"/>
  <limit velocity="4.8" effort="3.5" lower="${-1.5}" upper="${1.5}" />
  <dynamics damping="0.1" friction="0.008"/>
</joint>
```

Figura 4.13: Extracto del URDF mostrando los parámetros de fricción y el torque máximo (τ_{max}) definidos para cada articulación del robot.

Paquete ros2_control para Gazebo: Se empleó el uso del paquete `gazebo_ros2_control` [3], encargado de vincular las descripciones de los controladores definidas en los archivos YAML con las articulaciones del URDF. Este paquete actúa como puente entre el simulador y ROS 2, asegurando que los comandos publicados desde los nodos de control se traduzcan en acciones físicas aplicadas al modelo en Gazebo.

Controladores de esfuerzo: La configuración se completó con la definición de controladores de esfuerzo (`effort_controllers`). Estos controladores reciben vectores de torque como entrada y los aplican directamente a las articulaciones definidas en el URDF. Su correcta parametrización resultó crítica para garantizar que los torques calculados por el `manager_node` se aplicaran de forma estable y coherente durante la simulación.

4.3.4. Resultados y observaciones

4.3.5. Comportamiento del robot simulado

Durante la simulación se evaluó el comportamiento del robot al seguir trayectorias articulares generadas por el `trajectory_node` y controladas mediante el esquema MPC. Se observó que el manipulador lograba reproducir trayectorias de referencia con un seguimiento adecuado, aunque la precisión dependía de la correcta sintonización de las ganancias y de los parámetros del modelo dinámico. Los resultados mostraron la capacidad del sistema para aplicar torques consistentes a cada articulación y mantener la estabilidad en movimientos continuos.

4.3.6. Realismo de la simulación

El realismo de la simulación se evaluó en términos de la fidelidad de los efectos físicos modelados. Se comprobó que el modelo en Gazebo reproduce adecuadamente la acción de la gravedad, así como los efectos de fricción en las articulaciones. No obstante, se identificaron ciertas diferencias respecto al comportamiento esperado en hardware real, principalmente relacionadas con la ausencia de modelos de fricción más avanzados y con

las limitaciones de los actuadores simulados. Aún con estas diferencias, el entorno resultó suficientemente representativo para validar el funcionamiento del esquema MPC en condiciones cercanas a la realidad.

4.3.7. Limitaciones encontradas

Durante la integración se identificaron limitaciones derivadas de la arquitectura de los paquetes oficiales del OpenManipulator-X. Estos paquetes, diseñados para trabajar con el hardware real, incluyen controladores orientados a los motores Dynamixel XM430 [28], en los que las señales de control corresponden a *corriente* y no directamente a torques. En la simulación, sin embargo, no existen equivalentes que emulen los controladores propietarios de dichos motores, por lo que fue necesario implementar controladores de esfuerzo genéricos.

Esta diferencia plantea un desafío en caso de trasladar la solución del entorno simulado al robot real, ya que en hardware deben considerarse los paquetes del fabricante, la comunicación con los drivers y la conversión entre torque y corriente de los actuadores. En consecuencia, la implementación del control predictivo requiere ajustes adicionales al pasar de la simulación a la ejecución sobre el manipulador físico.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 5

Brazo robótico

En este capítulo se describe el modelo físico y computacional del manipulador utilizado en este trabajo, estableciendo el marco teórico necesario para el desarrollo de la red neuronal informada por física (PINN). Se introducen los conceptos fundamentales del modelado de manipuladores: articulaciones, eslabones, grados de libertad, espacio de trabajo y cinemática. Luego se detalla el funcionamiento de los algoritmos recursivos de dinámica (*RNEA* y *ABA*) y su implementación mediante la biblioteca Pinocchio. A continuación, se presenta el modelo específico del OpenMANIPULATOR-X, incluyendo su descripción mecánica, parámetros y simplificaciones adoptadas para este estudio. También se muestra la formulación de la PINN aplicada al brazo robótico, abarcando su estructura, funciones de pérdida, entrenamiento y validación.

5.1. Conceptos Teóricos

El brazo se describe a partir del modelo multi-joint que provee el fabricante en el archivo *.urdf*, en el se detallan las componentes que definen al brazo como sistema mecánico. Se verá a continuación el modelado de manipuladores para entender que contiene este archivo y como se usa.

5.1.1. Modelado de manipuladores

Los manipuladores robóticos están formados por un conjunto de cuerpos rígidos conectados entre sí mediante articulaciones, conformando una *cadena cinemática*. Cada articulación permite un tipo específico de movimiento relativo entre los cuerpos que une, y se clasifica según el tipo de desplazamiento que posibilita. A cada articulación se le asocia una variable articular q_i, la cual parametriza el movimiento permitido. A continuación se describen aquellas relevantes para el modelado de brazos robóticos en general:

- **Revoluta:** permite la rotación relativa entre los cuerpos rígidos unidos a ella. Tiene un solo grado de libertad que se expresa en la variable articular $q_i = \theta_i$ (un ángulo).
- **Prismatica:** permite la traslación relativa a lo largo de un eje fijo, también tiene un grado de libertad y se expresa $q_i = d_i$ (una distancia).

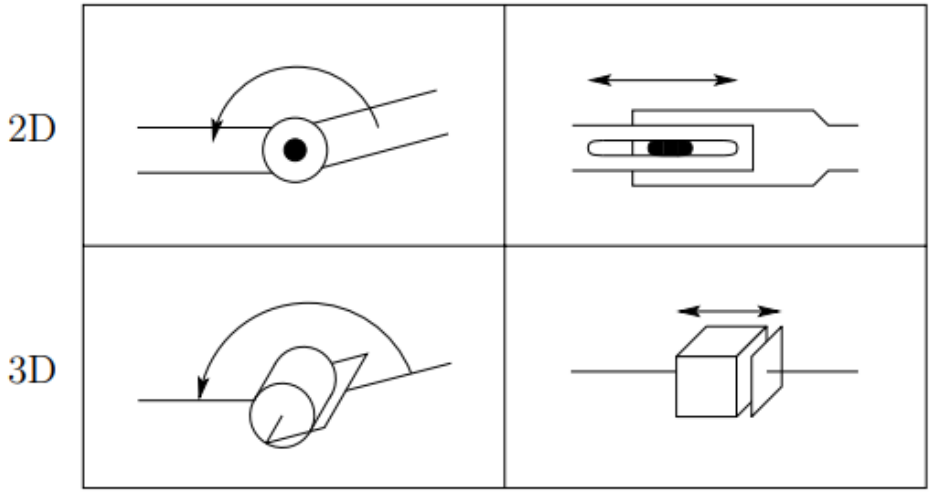


Figura 5.1: Movimientos según tipo de articulación, a la derecha prismática y a la izquierda de revolución.

Existen otros tipos de articulaciones que tienen mayor cantidad de grados de libertad (**esférica** de 3 grados de libertad, **rótula** que tiene dos, etc ..) pero las dos mencionadas son suficientes para describir los manipuladores básicos. En la imagen 5.1 se puede ver cómo es el movimiento de cada tipo. Estas articulaciones tienen **eslabones** (o *links*) acoplados. Los *links* son cuerpos rígidos que conectan articulaciones consecutivas, estos cuerpos rígidos se definen mediante sus parámetros inerciales. Para cada link i:

- **Masa** m_i
- **Frame** eje de coordenadas solidario al rígido conformado por el link
- **Centro de masas** (CoM) $c_i \in \mathbb{R}^3$, expresado en el frame del eslabón
- **Tensor de inercia** en el CoM, $\bar{I}_i \in \mathbb{R}^{3\times3}$ (simétrico)

Entonces, cada *link* queda definido por un parámetro de masa, tres coordenadas para el centro de masa en su frame y los tensores de inercia, otros seis parámetros. Un manipulador *multi-joint* puede verse como una secuencia ordenada de eslabones rígidos conectados mediante articulaciones (*cadena cinemática* o *kinematic chain*), donde cada par (link_i, joint_i) aporta una transformación cinemática entre cuerpos consecutivos. La composición de todas ellas define la configuración del sistema y determina cómo se propagan las velocidades, aceleraciones y fuerzas a lo largo de la cadena.

Esta estructura en cadena, junto con los parámetros inerciales de cada eslabón (m_i, c_i, $\bar{I}_i$), constituye la base del modelado dinámico de manipuladores robóticos. En la Figura 5.2 puede verse, esquemáticamente, una *cadena cinemática*.

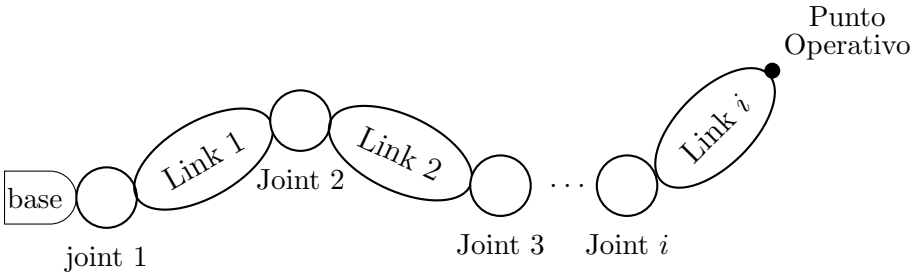


Figura 5.2: Cadena cinemática de largo i, en la que pueden verse todos los elementos que la componen. Se agrego en esta imagen el *punto operativo*, del cual se profundiza más adelante en 5.1.2. También se especifica acerca de la base.

En esta estructura, queda implícito que el *joint* i conecta el *link* $(i-1)$ con el *link* i, estableciendo la transformación relativa entre ambos. Al link $i-1$ se le llama *parent link*, mientras que el link i se denomina *child link*, estos nombres provienen de la jerarquia que tiene cada link sobre el marco de referencia (frame) del siguiente. El *parent link* define el marco de referencia desde el cual se describe la posición y orientación del *child*, y el movimiento permitido por la articulación se parametriza mediante la variable articular q_i. Esta relación jerárquica (*parent–child*) permite construir las ecuaciones de cinemática y dinámica de forma recursiva, propagando posiciones, velocidades y fuerzas a lo largo de la cadena, concepto muy utilizado en los algoritmos que se detallan más adelante en el documento.

5.1.2. Punto Operativo

Como se puede ver en la Figura 5.2, al final del último link se incluye el *punto operativo* del manipulador. Este punto representa el extremo activo del robot, es decir, la ubicación en la cual el manipulador interactúa físicamente con su entorno o ejecuta una tarea, ya sea sujetar un objeto, soldar, medir o posicionar una herramienta.

El punto operativo se encuentra fijado rígidamente al último eslabón y constituye la referencia desde la cual se describe la posición y orientación del manipulador en el espacio cartesiano. Su ubicación puede coincidir con el marco del último link o desplazarse respecto a él mediante una transformación rígida definida por el diseño mecánico de la herramienta acoplada. Las trayectorias que ejecuta un manipulador se definen normalmente en torno a este punto operativo, ya que es el que refleja la acción física del robot sobre el entorno. En consecuencia, una trayectoria robótica especifica la evolución temporal deseada de la posición y orientación del punto operativo, expresada en el espacio cartesiano. Durante la planificación de movimiento, dichas trayectorias se representan como una función continua del tiempo:

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{p}(t) \\ \boldsymbol{\phi}(t) \end{bmatrix}, \tag{5.1}$$

donde $\mathbf{p}(t)$ describe la posición y $\boldsymbol{\phi}(t)$ la orientación del punto operativo. A partir de esta referencia cartesiana, se determinan las trayectorias articulares equivalentes $\mathbf{q}(t)$ mediante la resolución de la cinemática inversa (en este caso, $\phi(t)$ es una constante ya que el último link del modelo del robot es *prismático*).

5.1.3. Grados de libertad

Cada articulación introduce un grado de libertad (*Degree of Freedom*, DOF) al sistema, asociado a la variable articular q_i. En general, un manipulador con n articulaciones independientes posee n grados de libertad, lo que determina la dimensión de su espacio articular (*joint space*). Cada una de estas variables articulares describe una coordenada generalizada que parametriza la posición relativa entre dos eslabones consecutivos. El conjunto de todas las variables articulares se agrupa en el vector generalizado:

$$\mathbf{q} = [q_1,\, q_2,\, \ldots,\, q_n]^T \tag{5.2}$$

Este vector define el estado articular del robot y, por ende, su configuración instantánea. A partir de $\mathbf{q}$ pueden determinarse las transformaciones cinemáticas que relacionan los marcos de referencia de cada eslabón, así como la posición y orientación de su *punto operativo* en el espacio cartesiano. En otras palabras, conocer la configuración articular $\mathbf{q}$ es

suficiente para reconstruir completamente la geometría del manipulador en un instante dado. Por esta razón, las ecuaciones cinemáticas del sistema se formulan en función de $\mathbf{q}$, sus derivadas temporales $\dot{\mathbf{q}}$ (velocidades articulares) y $\ddot{\mathbf{q}}$ (aceleraciones articulares), que constituyen las variables de estado fundamentales en el modelado dinámico. Además, el número de grados de libertad define la *capacidad de movimiento* del robot: un manipulador con $n = 6$ DOF, por ejemplo, puede posicionar y orientar su *punto operativo* arbitrariamente en el espacio tridimensional, mientras que manipuladores de menor número de DOF sólo pueden alcanzar subconjuntos de ese espacio. El número y disposición de los grados de libertad determinan también el *espacio de trabajo* del robot, es decir, la región del espacio que el punto operativo puede alcanzar.

5.1.4. Espacio de trabajo

El conjunto de variables articulares $\mathbf{q}$ define el *espacio articular* (*joint space*), en el cual se describen las configuraciones internas del robot. Por otro lado, el conjunto de variables que describen la posición y orientación del punto operativo conforman el *espacio cartesiano* o *espacio de trabajo* (*task space*). La relación entre ambos espacios está dada por la cinemática directa e inversa del manipulador:

$$\mathbf{x} = f(\mathbf{q}), \qquad \mathbf{q} = f^{-1}(\mathbf{x}) \tag{5.3}$$

donde $f(\cdot)$ es, en general, una función no lineal que depende de la geometría y configuración del robot, además la función $f^{-1}(x)$ puede no ser única. En la práctica, tanto el control como la planificación pueden formularse en cualquiera de estos dos espacios, según si se desea especificar trayectorias articulares o trayectorias del punto operativo en el espacio cartesiano. Por lo tanto, en el espacio articular las trayectorias se expresan directamente en términos de las variables $\mathbf{q}_i$, mientras que en el espacio de tarea se describen en términos de la posición y orientación del punto operativo, la variable $\mathbf{x}$.

5.1.5. Cinemática

El objetivo de esta sección es establecer los conceptos necesarios para la correcta interpretación de los algoritmos fundamentales empleados en el estudio de manipuladores robóticos. Se presentan de manera breve los principios teóricos que permiten comprender el funcionamiento general de dichos algoritmos. El propósito principal es analizar **la relación entre las variables articulares (q), sus velocidades y aceleraciones ($\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}$), y los torques $\boldsymbol{\tau}$ aplicados sobre cada articulación**.

El desarrollo que se expone a continuación se basa principalmente en la formulación presentada en el libro *Robot Modeling and Control* de Spong et al. [32], complementada con aportes de Featherstone [6]. En esta sección se introducen los fundamentos cinemáticos y dinámicos necesarios para comprender los algoritmos recursivos de cálculo dinámico utilizados en este trabajo: el *RNEA* (Recursive Newton-Euler Algorithm) y el *ABA* (Articulated Body Algorithm), ambos descritos en [6] e implementados en la biblioteca **Pinocchio**. Asimismo, se presentan los conceptos básicos relacionados con la cinemática directa e inversa aplicados a *cadenas cinemáticas*.

5.1.6. Cinemática directa

La cinemática directa describe la relación entre las variables articulares $\mathbf{q} = [q_1, q_2, \ldots, q_n]^T$ y la posición y orientación del punto operativo del manipulador en el espacio cartesiano.

Siguiendo a [32], cada articulación y eslabón del robot se asocia con un marco de referencia (frame), de modo que la transformación ${}^{i-1}T_i(q_i)$ expresa la posición y orientación del eslabón i respecto al anterior. Estas transformaciones se representan como matrices

$$ {}^{i-1}T_i = \begin{bmatrix} {}^{i-1}R_i & {}^{i-1}p_i \\ 0 & 1 \end{bmatrix} \tag{5.4} $$

donde ${}^{i-1}R_i$ describe la orientación del frame i y ${}^{i-1}p_i$ el vector de posición del origen i en coordenadas del frame $i-1$. A partir del producto encadenado de estas transformaciones (homogéneas) se obtiene la posición y orientación del eslabón final expresadas en el frame de la base, acumulando recursivamente las transformaciones de cada articulación anterior.

$$ {}^{0}T_n(\mathbf{q}) = {}^{0}T_1(q_1)\,{}^{1}T_2(q_2)\,\cdots\,{}^{n-1}T_n(q_n). \tag{5.5} $$

Entonces ${}^{0}T_n(\mathbf{q})$ nos da la posición y orientación del último link de la cadena cinemática. Esto es lo que se llama cinemática directa, **dado un vector de posiciones articulares, obtener las coordenadas cartesianas del punto operativo** ($\mathbf{x} = f(\mathbf{q})$). Las transformaciones ${}^{i-1}T_i(q)$ no son más que rotaciones y traslaciones, que llevan de un marco de referencia a otro. Luego ${}^{i-1}R_i$ representa la rotación de i con respecto a $i-1$ y ${}^{i-1}p_i$ representa el desplazamiento del origen del eje i con respecto al $i-1$. En [32](cap. 3) se ahonda sobre estas matrices y se presenta una convención para definir estas matrices como el producto de 4 matrices básicas que se expresan a partir de los parámetros de cada link. A esta convención se le conoce como *Denavit-Hartenberg convention* y refuerza la intuición sobre las transformaciones ${}^{i-1}T_n(q)$:

$$ {}^{i-1}T_i = Rot_z(\theta_i)Trans_z(d_i)Trans_x(a_i)Rot_x(\alpha_i) \tag{5.6} $$

Donde las transformaciones quedan expresadas a partir de rotaciones y traslaciones contra los ejes x y z. Es importante mantener esta intuición, y recordar que la *cinemática directa* busca resolver el problema:

$$ \mathbf{x} = f(\mathbf{q}) \tag{5.7} $$

La concatenación de las matrices ${}^{i-1}T_i$ permite resolver esta relación, proporcionando así la función f. A continuación, se analiza el cálculo de las velocidades $\dot{\mathbf{x}}$. Dado que f se define como un producto de matrices ${}^{i-1}T_i$, y que cada una de ellas se obtiene a partir del producto de matrices básicas, resulta razonable derivar dicha expresión para obtener las velocidades.

En lo que sigue, se denotará con n al número de articulaciones de una cadena cinemática y con m a la dimensión del espacio de trabajo. En general, este espacio posee dimensión 6: las tres primeras componentes corresponden a las coordenadas del origen del marco de referencia en el espacio cartesiano (x, y, z), mientras que las tres restantes representan los ángulos que describen la orientación del eje (ángulos de Euler).

5.1.7. Jacobiano de un manipulador

Derivando la expresión del problema de cinemática directa, recordando que $f : \mathbb{R}^n \to \mathbb{R}^m$ representa la función que asocia las coordenadas articulares con la posición y orientación del punto operativo, se obtiene:

$$ \dot{\mathbf{x}} = J(\mathbf{q})\,\dot{\mathbf{q}} \tag{5.8} $$

donde $J(\mathbf{q}) = \frac{\partial f(\mathbf{q})}{\partial \mathbf{q}} \in \mathbb{R}^{m \times n}$. En este punto puede ser útil revisar las dimensiones de estas expresiones. $J(\mathbf{q})$ depende únicamente de la configuración articular del manipulador ya que su cálculo se basa en las orientaciones y posiciones relativas de los eslabones que son funciones de las coordenadas articulares $\mathbf{q}$. Por lo tanto el Jacobiano varía con la postura del robot. Para desarrollar la intuición, se verá como $J(\mathbf{q})$ *transforma* velocidades articulares ($\mathbf{q}$) en velocidades cartesianas, observando primero las dimensiones:

$$\underbrace{\dot{\mathbf{x}}}_{m\times 1} = \underbrace{J(\mathbf{q})}_{m\times n}\underbrace{\dot{\mathbf{q}}}_{n\times 1} \tag{5.9}$$

El Jacobiano es el resultado de derivar las transformaciones que relacionan la posición y orientación del link final con las coordenadas articulares. Al derivar una transformación ${}^{i-1}T_i(q_i)$ con respecto al tiempo, aparecen dos tipos de términos: unos asociados a la derivada de las traslaciones y otros a la derivada de las rotaciones. Estos términos se agrupan naturalmente en dos bloques:

$$J(\mathbf{q}) = \begin{bmatrix} J_v(\mathbf{q}) \\ J_\omega(\mathbf{q}) \end{bmatrix} \tag{5.10}$$

, donde J_v relaciona las velocidades articulares con la velocidad lineal del punto operativo y J_ω con su velocidad angular. En lugar de calcular estos bloques de forma explícita, los algoritmos recursivos de dinámica emplean formulaciones equivalentes en términos de velocidades lineales $\mathbf{v}_i$ y angulares $\boldsymbol{\omega}_i$, que se propagan recursivamente a lo largo de la cadena cinemática. Estas expresiones se presentan a continuación.

5.1.8. Cinemática diferencial

Para los algoritmos recursivos de dinámica es suficiente conocer cómo se propagan las velocidades y aceleraciones a lo largo de la cadena cinemática, sin necesidad de construir explícitamente el Jacobiano. Siguiendo [32](cap. 4), las velocidades angulares y lineales de cada eslabón pueden expresarse de forma recursiva en función del eslabón anterior:

$$\boldsymbol{\omega}_i = {}^{i}R_{i-1}\,\boldsymbol{\omega}_{i-1} + \dot{q}_i\,\mathbf{z}_i \tag{5.11}$$

,

$$\mathbf{v}_i = {}^{i}R_{i-1}\,(\mathbf{v}_{i-1} + \boldsymbol{\omega}_{i-1} \times \mathbf{p}_i) \tag{5.12}$$

, donde ${}^{i}R_{i-1}$ representa la rotación entre los frames consecutivos $i-1$ e i, $\mathbf{p}_i$ es el vector que une sus orígenes, $\mathbf{z}_i$ el eje de la articulación i, y $\dot{q}_i$ la velocidad articular correspondiente. Estas expresiones se obtienen derivando las transformaciones homogéneas que relacionan los frames consecutivos en la cadena cinemática. La primera ecuación describe cómo se propaga la velocidad angular entre eslabones consecutivos, mientras que la segunda expresa la propagación de la velocidad lineal del origen del frame i. Ambas dependen únicamente de la configuración y del tipo de articulación. Estas expresiones son recursivas, es decir, la velocidad y aceleración de cada eslabón se calculan a partir del anterior. Para iniciar la recursión, se define el estado cinemático del eslabón base (link 0). En el caso de un manipulador de base fija, se asume que la base no presenta movimiento ni aceleración:

$$\boldsymbol{\omega}_0 = \mathbf{0}, \qquad \mathbf{v}_0 = \mathbf{0}, \qquad \dot{\boldsymbol{\omega}}_0 = \mathbf{0}, \qquad \dot{\mathbf{v}}_0 = \mathbf{0}.$$

Con estas condiciones iniciales, la propagación hacia adelante permite obtener las velocidades y aceleraciones de todos los eslabones. Por ejemplo, para el primer eslabón se cumple:

$$\boldsymbol{\omega}_1 = \dot{q}_1\,\mathbf{z}_1, \qquad \mathbf{v}_1 = \mathbf{0}.$$

Estas expresiones son fundamentales para los algoritmos de resolución de dinámica inversa y directa usados durante el proyecto. Las aceleraciones se obtienen derivando las expresiones anteriores:

$$\dot{\boldsymbol{\omega}}_i = {}^iR_{i-1}\,\dot{\boldsymbol{\omega}}_{i-1} + \ddot{q}_i\,\mathbf{z}_i + \dot{q}_i\left({}^iR_{i-1}\boldsymbol{\omega}_{i-1} \times \mathbf{z}_i\right) \tag{5.13}$$

$$\dot{\mathbf{v}}_i = {}^iR_{i-1}(\dot{\mathbf{v}}_{i-1} + \dot{\boldsymbol{\omega}}_{i-1} \times \mathbf{p}_i + \boldsymbol{\omega}_{i-1} \times (\boldsymbol{\omega}_{i-1} \times \mathbf{p}_i)) \tag{5.14}$$

Estas ecuaciones recursivas describen la propagación de las aceleraciones a lo largo de la cadena cinemática. Cada término tiene una interpretación física directa: el primero corresponde a la aceleración heredada del eslabón anterior, el segundo a la aceleración producida por la junta actual ($\ddot{q}_i$), y el tercero a los efectos centrífugos y de Coriolis (movimiento relativo entre eslabones).

5.1.9. Ecuaciones de movimiento

A partir de las relaciones cinemáticas obtenidas, es posible establecer las ecuaciones que rigen la dinámica del manipulador. Estas ecuaciones describen cómo las fuerzas y torques aplicados en las articulaciones producen aceleraciones en los eslabones del sistema. Existen dos formulaciones equivalentes para obtenerlas: la formulación *Newton–Euler*, basada directamente en las leyes de movimiento de cada cuerpo, y la formulación *Euler–Lagrange*, que utiliza el principio de energía para expresar la dinámica del sistema en coordenadas generalizadas. No se ahondará en detalles sobre la derivación, pero la formulación de *Euler-Lagrange* conduce al modelo dinámico clásico de los manipuladores:

$$M(\mathbf{q})\,\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\,\dot{\mathbf{q}} + g(\mathbf{q}) = \boldsymbol{\tau} \tag{5.15}$$

. Donde $\boldsymbol{\tau} = [\,\tau_1, \tau_2, \ldots, \tau_n\,]^T$, es un vector de fuerzas sobre cada articulación n. Esta expresión relaciona directamente las variables articulares $\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}$ con los torques $\boldsymbol{\tau}$ que actúan sobre las articulaciones. Cada término posee una interpretación física:

- $M(\mathbf{q})$: matriz de inercia del manipulador, que agrupa las masas y momentos de inercia de todos los eslabones.
- $C(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}}$: vector de fuerzas centrífugas y de Coriolis que aparecen debido al movimiento relativo entre eslabones.
- $g(\mathbf{q})$: vector de torques debidos a la gravedad.

En conjunto, estos tres términos determinan la dinámica completa del sistema.

5.1.10. Importancia de los métodos recursivos

Los métodos presentados a continuación resultan fundamentales en el estudio y control de manipuladores robóticos. Hasta este punto se han introducido las herramientas necesarias para describir la cinemática y las relaciones dinámicas generales de un manipulador, culminando en el modelo dinámico:

$$M(\mathbf{q})\,\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\,\dot{\mathbf{q}} + g(\mathbf{q}) = \boldsymbol{\tau}.$$

Resolver esta ecuación de forma eficiente permite abordar los dos problemas centrales de la dinámica de manipuladores:

- **Dinámica inversa:** dado $(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$, determinar los torques $\boldsymbol{\tau}$ necesarios para producir ese movimiento.
- **Dinámica directa:** dado $(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau})$, calcular las aceleraciones articulares resultantes $\ddot{\mathbf{q}}$.

Ambos problemas son esenciales para la simulación, el control y la planificación del movimiento en robótica. En el presente caso, esta ecuación constituye el término correspondiente a la pérdida en la ecuación diferencial, siguiendo la formulación para la PINN dada en 2.2.

En la práctica el cálculo explícito de M, C y g es computacionalmente costoso, ya que hay que construir estas matrices, que contienen dependencias cruzadas entre todas las articulaciones. Por esta razón, se emplean formulaciones recursivas que aprovechan la estructura en cadena del manipulador, como los algoritmos RNEA y ABA, los cuales permiten obtener las mismas relaciones con menor costo computacional (esta es la *feature* principal de **Pinocchio**, implementar estos algoritmos en orden $O(n)$). A continuación se detalla cada algoritmo.

5.1.11. Recursive Newton–Euler Algorithm (RNEA)

Este algoritmo permite calcular las fuerzas y torques articulares requeridos para generar un movimiento dado del manipulador. En otras palabras, resuelve el problema de **dinámica inversa**: dado un conjunto de posiciones, velocidades y aceleraciones articulares $(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$, determina los torques $\boldsymbol{\tau}$ que satisfacen

$$M(\mathbf{q})\,\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\,\dot{\mathbf{q}} + g(\mathbf{q}) = \boldsymbol{\tau}.$$

RNEA no requiere construir explícitamente las matrices M, C y g. En cambio, emplea una estrategia recursiva basada en las leyes de Newton y Euler para propagar las velocidades, aceleraciones y fuerzas a lo largo de la cadena cinemática, aprovechando la estructura jerárquica del manipulador.

Estructura general del algoritmo

El algoritmo se compone de dos fases principales:

1. **Propagación hacia adelante (forward recursion):** Se calculan las velocidades y aceleraciones de cada eslabón a partir del movimiento del eslabón anterior. En este paso se aplican las expresiones vistas anteriormente. Empezando por las velocidades:

$$\boldsymbol{\omega}_i = {}^{i}R_{i-1}\,\boldsymbol{\omega}_{i-1} + \dot{q}_i\,\mathbf{z}_i, \quad \mathbf{v}_i = {}^{i}R_{i-1}(\mathbf{v}_{i-1} + \boldsymbol{\omega}_{i-1} \times \mathbf{p}_i) \tag{5.16}$$

   Y sus aceleraciones:

$$\begin{aligned} \dot{\boldsymbol{\omega}}_i &= {}^{i}R_{i-1}\,\dot{\boldsymbol{\omega}}_{i-1} + \ddot{q}_i\,\mathbf{z}_i + \dot{q}_i\left({}^{i}R_{i-1}\boldsymbol{\omega}_{i-1} \times \mathbf{z}_i\right), \\ \dot{\mathbf{v}}_i &= {}^{i}R_{i-1}(\dot{\mathbf{v}}_{i-1} + \dot{\boldsymbol{\omega}}_{i-1} \times \mathbf{p}_i + \boldsymbol{\omega}_{i-1} \times (\boldsymbol{\omega}_{i-1} \times \mathbf{p}_i)) \end{aligned} \tag{5.17}$$

   En esta etapa se incluyen los efectos de la aceleración de la gravedad, normalmente añadida a la base como $\dot{\mathbf{v}}_0 = -\mathbf{g}$.

2. **Propagación hacia atrás (backward recursion):** Una vez conocidas las aceleraciones, se calculan las fuerzas y torques transmitidos desde los links hasta la base

$$\mathbf{f}_i = I_i\,\dot{\boldsymbol{\omega}}_i + \boldsymbol{\omega}_i \times (I_i\,\boldsymbol{\omega}_i) + m_i\,\dot{\mathbf{v}}_i \tag{5.18}$$

$$\mathbf{n}_i = I_i\,\dot{\boldsymbol{\omega}}_i + \boldsymbol{\omega}_i \times (I_i\,\boldsymbol{\omega}_i) + \mathbf{p}_i \times (m_i\,\dot{\mathbf{v}}_i) \tag{5.19}$$

Donde $\mathbf{f}_i$ y $\mathbf{n}_i$ son respectivamente las fuerzas y momentos ejercidos sobre el eslabón i. Los torques articulares se obtienen proyectando estas fuerzas sobre el eje de la articulación:

$$\tau_i = \mathbf{n}_i^T\,\mathbf{z}_i \tag{5.20}$$

El algoritmo RNEA permite calcular $\boldsymbol{\tau}$ de manera eficiente sin necesidad de formar explícitamente las matrices de inercia ni los términos de Coriolis o gravedad. Además, su formulación estructurada lo hace altamente compatible con implementaciones numéricas modernas y diferenciables. **Más adelante se retomará este algoritmo para analizar el papel que desempeña dentro de la red neuronal**, ya que constituye el elemento central encargado de incorporar los términos físicos en el modelo.

5.1.12. Articulated Body Algorithm (ABA)

Introducido por Featherstone [6], permite resolver el problema de dinámica directa, es decir, calcular las aceleraciones articulares $\ddot{\mathbf{q}}$ dado el estado del sistema $(\mathbf{q}, \dot{\mathbf{q}})$ y los torques aplicados $\boldsymbol{\tau}$. Matemáticamente, ABA busca resolver de manera eficiente:

$$M(\mathbf{q})\,\ddot{\mathbf{q}} = \boldsymbol{\tau} - C(\mathbf{q}, \dot{\mathbf{q}})\,\dot{\mathbf{q}} - g(\mathbf{q}) \tag{5.21}$$

sin necesidad de invertir la matriz de inercia $M(\mathbf{q})$. El principio del algoritmo consiste en aprovechar la estructura en cadena del manipulador para propagar de manera recursiva las propiedades dinámicas equivalentes de grupos de links conectados. Cada grupo se denomina *cuerpo articulado* (*articulated body*) y tiene una **inercia equivalente** I_i^A que resume las masas y acoplamientos de todos los eslabones descendientes del eslabón i.

Aunque el algoritmo ABA comparte con RNEA la estructura recursiva y el objetivo de resolver la dinámica del manipulador, su formulación es significativamente más compleja. Esto se debe a que ABA trabaja con representaciones **espaciales unificadas** de movimiento y fuerza (velocidades y torques expresados en un mismo vector de 6 componentes), y realiza las operaciones directamente sobre el espacio de transformaciones rígidas tridimensionales ($SE(3)$), donde rotación y traslación se encuentran acopladas.

Esto se traduce en que la formulación del problema requiere un marco matemático mas general (formalmente, se trabaja en el grupo de Lie $SE(3)$, que representa transformaciones rígidas en el espacio). Esta representación unificada permite mayor generalidad, pero también vuelve las ecuaciones más complejas que en el caso del RNEA.

Por esta razón, aunque el principio conceptual del ABA es claro, (propagar dinámicamente las contribuciones de los eslabones descendientes para calcular las aceleraciones sin invertir la matriz de inercia) su desarrollo algebraico completo requiere una notación más abstracta introducida por Featherstone en [6] (dedicando un capítulo entero, cap 7). En consecuencia, en este trabajo se presenta únicamente una descripción conceptual del algoritmo, dejando su formulación matemática detallada fuera del alcance del documento.

Una vez establecido el marco teórico necesario para comprender la construcción matemática del manipulador, se procede a definir de manera concreta el manipulador utilizado en este proyecto.

5.2. Open Manipulator-X

5.2.1. Descripción general del robot

El manipulador utilizado en este proyecto es el **OpenMANIPULATOR-X** (modelo RM-X52-TNM), desarrollado por la empresa ROBOTIS. Se trata de un brazo robótico modular de código abierto, diseñado específicamente para aplicaciones educativas y de investigación en robótica. El sistema combina hardware accesible, software compatible con ROS y actuadores de la línea DYNAMIXEL, permitiendo un entorno de desarrollo completamente reproducible [29].

El robot posee un total de 5 grados de libertad (4 articulaciones y uno correspondiente a la pinza), alcanzando una extensión máxima aproximada de $380mm$ y una capacidad de carga de $500g$. Cuenta con una estructura ligera, de aproximadamente $700g$, está compuesta por eslabones y actuadores `XM430-W350-T` alimentados a $12V$.

En este proyecto no se cuenta con el brazo físicamente. Todas las pruebas y validaciones se ejecutaron en un **entorno de simulación** basado en Gazebo, utilizando modelos dinámicos equivalentes al hardware real. Además, con el objetivo de simplificar el problema de optimización del control predictivo y evaluar la dinámica aprendida por la PINN, se considerarán únicamente las **2 primeras articulaciones activas**, manteniendo las restantes fijas en configuraciones de equilibrio.

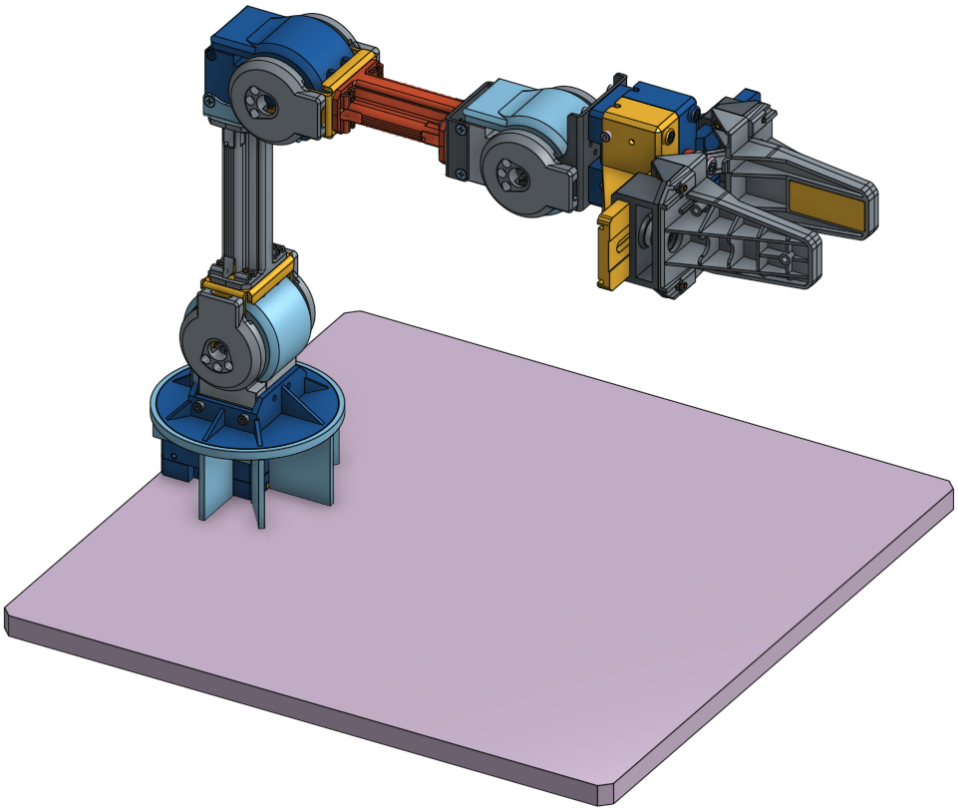


Figura 5.3: Modelo del OpenMANIPULATOR-X en el entorno de simulación Gazebo.

La elección de esta plataforma se fundamenta en su arquitectura abierta y en su alta compatibilidad con ROS 2, lo que permite una integración directa con los nodos desarrollados para el control MPC y la PINN. Si bien el modelo real emplea controladores DYNAMIXEL y comunicación serial TTL, en esta implementación dichos elementos fueron reemplazados por controladores de esfuerzo genéricos dentro del simulador, lo que

posibilita aplicar directamente los torques calculados por el nodo `manager_node` sobre las articulaciones simuladas.

Esta descripción general establece el marco físico y funcional sobre el cual se integran los algoritmos de control predictivo y de aprendizaje profundo en el entorno ROS 2.

5.2.2. Especificaciones técnicas

Las características técnicas del OpenMANIPULATOR-X dan un contexto realista para el diseño del controlador. Según el e-Manual oficial y como se observa en la Figura 5.4 el brazo tiene un alcance máximo (reach) de aproximadamente $380mm$ y un peso total de alrededor de $0{,}70kg$ [29]. Su carga útil nominal es de $500g$. Además, usa actuadores DYNAMIXEL XM430-W350-T y opera con alimentación de $12V$ [28].

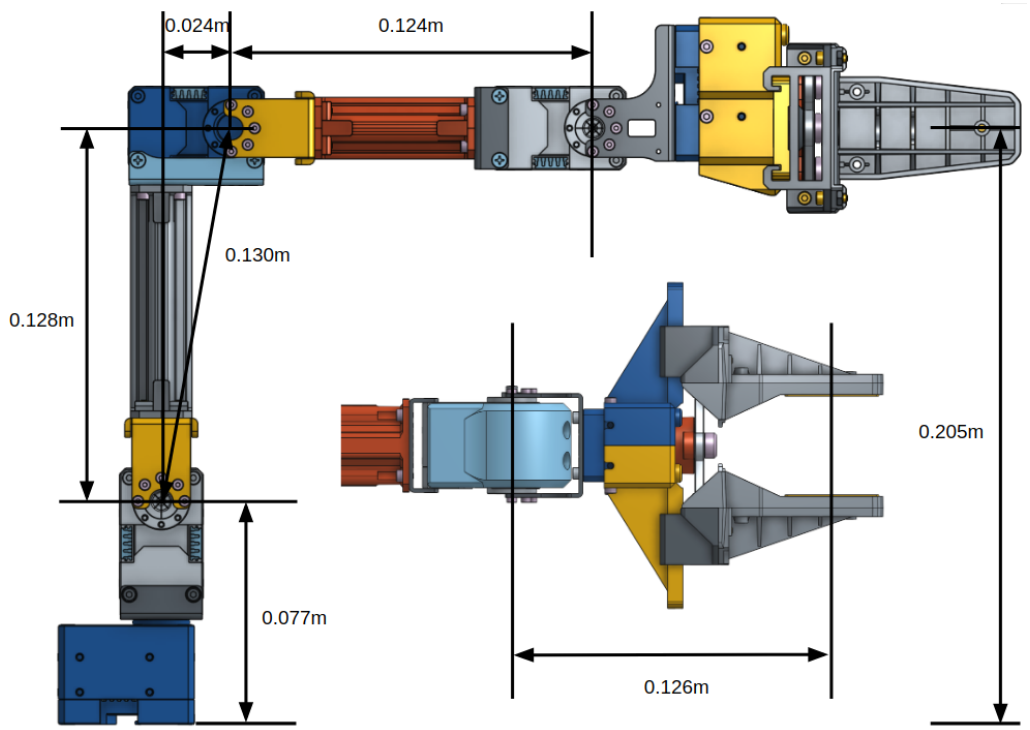


Figura 5.4: Esquema del OpenMANIPULATOR-X con las dimensiones principales de sus eslabones y límites articulares.

En el contexto de este proyecto, si bien el modelo se basa en el robot completo, únicamente se emplean **las dos primeras articulaciones activas** (como se explicará más adelante). Las medidas utilizadas en la simulación —tales como el alcance, las masas y los límites articulares— se inspiran en las especificaciones reales del manipulador, aunque se simplifican para adecuarse al dominio de control considerado.

Tabla resumen de parámetros relevantes:

Tabla 5.1: Parámetros técnicos del OpenMANIPULATOR-X utilizados como referencia en la simulación.

Parámetro	Valor nominal
Alcance máximo	380 mm
Peso total	700 g
Carga útil	500 g
Grados de libertad	5 (4 articulaciones + 1 gripper)
Voltaje de operación	12 V
Actuador empleado	XM430-W350-T
Límite articular $q_{\max}$	$\pm 180°$
Velocidad máxima $\dot{q}_{\max}$	4,8 rad/s
Torque máximo $\tau_{\max}$	3,5 N·m

Los parámetros del modelo simulado respetan los límites físicos del hardware real. En particular, los rangos articulares definidos en el archivo URDF se ajustaron a los límites mecánicos que soportan los actuadores del robot, evitando que el controlador genere comandos fuera del dominio operativo.

Por motivos de protección y estabilidad numérica, el par máximo permitido durante la simulación se limitó a **3.0 N·m**, un valor ligeramente inferior al torque nominal de los motores DYNAMIXEL XM430-W350-T. Esta restricción busca prevenir comportamientos no realistas en el modelo dinámico y garantizar que los torques calculados por el MPC se mantengan dentro de un rango físicamente seguro y coherente con las especificaciones del fabricante.

5.2.3. Modelo multi-joint

Poniendo este brazo dentro del marco de las *cadenas cinemáticas*, puede describirse como un manipulador serie conformado por una secuencia de eslabones rígidos unidos mediante articulaciones. Cada articulación introduce un grado de libertad (DOF), definido según el tipo de movimiento relativo permitido entre los eslabones consecutivos.

El manipulador OpenManipulator-X presenta cuatro articulaciones principales de tipo **revoluta**, que corresponden a los movimientos de rotación en las juntas de la base, el hombro, el codo y la muñeca, respectivamente. Estas cuatro juntas determinan completamente la posición y orientación del último eslabón antes de la pinza, conformando la parte *cinemáticamente activa* del brazo. Estas juntas y sus respectivos movimientos se pueden ver en la Figura 5.5

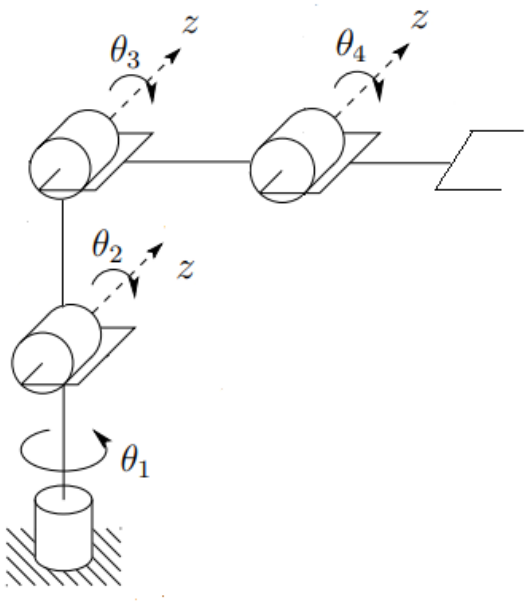


Figura 5.5: Modelo multi-joint del manipulador OpenManipulator-X, en el que se pueden ver 4 articulaciones revolutas, y una prismatica (la pinza al final de la cadena). Las uniones entre estas articulaciones son los *links* o eslabones, que tienen una descripción mecánica a tráves de los parametros ya mencionados. Este modelo mecánico es el que da sustento a todos los algoritmos usados.

En el extremo del último eslabón se encuentra una quinta articulación de tipo **prismática**, que acciona la apertura y cierre de la pinza. Esta junta prismática puede modelarse mediante un único parámetro correspondiente al desplazamiento lineal del actuador que modula la distancia entre las garras. Durante el proyecto, la última articulación se mantiene fija por simplicidad. A partir de esta estructura se define el vector de coordenadas articulares del manipulador como:

$$\mathbf{q} = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 \end{bmatrix}^\top, \tag{5.22}$$

donde cada componente θ_i representa el ángulo de rotación correspondiente a la articulación i-ésima del brazo. Asimismo, se definen las magnitudes:

$$\dot{\mathbf{q}} = \begin{bmatrix} \dot{\theta}_1 & \dot{\theta}_2 & \dot{\theta}_3 & \dot{\theta}_4 \end{bmatrix}^\top, \qquad \boldsymbol{\tau} = \begin{bmatrix} \tau_1 & \tau_2 & \tau_3 & \tau_4 \end{bmatrix}^\top, \tag{5.23}$$

donde $\dot{\mathbf{q}}$ representa las velocidades articulares y $\boldsymbol{\tau}$ el vector de torques aplicados por los actuadores en cada junta. Para cada grado de libertad i se especifican:

- El **rango de movimiento** permitido, es decir, los valores mínimos y máximos de posición articular $[\theta_i^{\text{mín}}, \theta_i^{\text{máx}}]$.
- Las **velocidades angulares máximas** $|\dot{\theta}_i|_{\text{máx}}$, definidas por las prestaciones de los servomotores.
- Los **torques máximos** $|\tau_i|_{\text{máx}}$ que pueden generar los actuadores sin exceder su régimen seguro de funcionamiento.

Estos valores constituyen las *restricciones físicas del sistema* y determinan la región de operación admisible para las variables $\mathbf{q}$, $\dot{\mathbf{q}}$ y $\boldsymbol{\tau}$ (según las especificaciones del fabricante [27]). En los experimentos y simulaciones se trabajará exclusivamente dentro de dichos límites, garantizando que los movimientos del modelo sean físicamente realizables y compatibles con las especificaciones del fabricante.

Tabla 5.2: Límites de operación de las articulaciones del manipulador

Junta	$\boldsymbol{\theta_i^{\text{mín}}}$ [rad]	$\boldsymbol{\theta_i^{\text{máx}}}$ [rad]	$\lvert\dot{\boldsymbol{\theta}}_i\rvert_{\text{máx}}$ [rad/s]	$\lvert\boldsymbol{\tau}_i\rvert_{\text{máx}}$ [Nm]
1 (Base)	$-2{,}6$	2,6	2,62	3,0
2	$-1{,}7$	1,7	2,62	3,0
3	$-1{,}5$	1,5	2,62	3,0
4	$-1{,}8$	1,8	2,62	3,0

Como ya fue mencionado y se detallará mas adelante, es posible mantener cualquiera de estas juntas bloqueada mecánicamente.

5.3. Pinocchio

Pinocchio es una biblioteca de código abierto especializada en cinemática y dinámica de cuerpos rígidos. En esta biblioteca se encuentran las implementaciones de muchos de los algoritmos usados para la simulación y diferenciación de modelos multi-body (en particular *cadenas cinemáticas*). Esta librería provee todo lo necesario para trabajar con modelos multi-joint, como el del brazo robotico. El núcleo de **Pinocchio** se organiza alrededor de tres componentes principales: `Model`, `Data` y `Algorithms`. Este diseño permite implementar los algoritmos de cinemática y dinámica de manera eficiente, modular y numéricamente estable.

El objeto `Model` contiene toda la información estructural e invariante del robot, independiente de su estado instantáneo. Incluye la lista de articulaciones y eslabones, sus relaciones jerárquicas *(parent–child)*, los parámetros inerciales de cada link (m_i, $\bar{I}_i$, c_i), las transformaciones ${}^{i-1}T_i$ que describen la geometría entre un joint i y su *parent link* ($p(i)$), y el tipo de articulación (revoluta, prismática, etc.). Cada articulación está indexada numéricamente, de modo que el conjunto completo forma un **grafo dirigido acíclico**, donde el nodo raíz representa la base del robot y las ramas los links que dependen de ella. También define el enlace entre un *parent joint* y su *child joint*, y este grafo dirigido constituye la topología estructural del manipulador. Durante la ejecución de algoritmos como RNEA o ABA, este grafo se recorre en dos fases: una *propagación hacia adelante* (*forward pass*) y una *propagación hacia atrás* (*backward pass*).

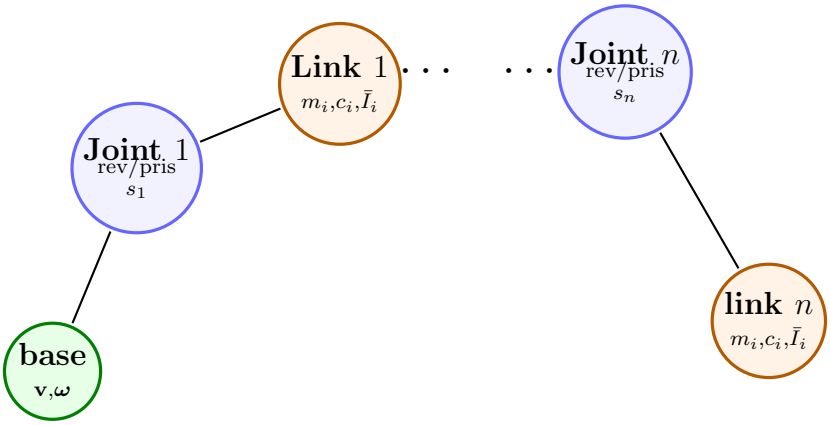


Figura 5.6: Grafo de cadena cinemática guardado en objeto `Model`. Este grafo guarda todo lo relacionado con el modelo mecánico del manipulador. s_i representa el *frame* asociado al joint i

En la Figura 5.6 se puede ver esquemáticamente el grafo guardado en el objeto `Model`, donde s_i representa el *frame* asociado al joint i. Por otro lado, el objeto `Data` se asocia directamente al `Model` y contiene las variables **dependientes del estado** ($\mathbf{q}$, $\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}$). Aquí se almacenan las velocidades y aceleraciones (articulares y espaciales), las fuerzas netas y todos los resultados intermedios generados por los algoritmos recursivos (*RNEA*, *ABA*). La separación entre `Model` y `Data` permite evaluar múltiples configuraciones articulares de un mismo robot sin reconstruir su estructura completa, reduciendo significativamente los costos computacionales y de memoria.

Finalmente, esta organización con una descripción estructural invariante (`Model`), un contenedor dinámico (`Data`) y algoritmos genéricos que operan sobre ambos, permite que cualquier robot definido mediante un archivo **URDF** se convierta automáticamente en un modelo dinámico completo, capaz de resolver cinemática, dinámica y derivadas analíticas de forma eficiente. Otra función que resultó muy útil de la biblioteca es la función `buildReducedModel` la cual recibe una lista de articulaciones a fijar y una configuración de referencia $\mathbf{q}_{\text{ref}}$, eliminando los grados de libertad asociados y fusionando los cuerpos rígidos correspondientes. De este modo, se obtienen cuerpos rígidos acoplados que conservan la dinámica global del sistema, pero con menor dimensionalidad. Esto resulta especialmente ventajoso para el entrenamiento de la PINN, ya que simplifica la dinámica del brazo sin perder coherencia física. En la Figura 5.7 se puede ver un diagrama de los módulos utilizados de la biblioteca **Pinocchio**.

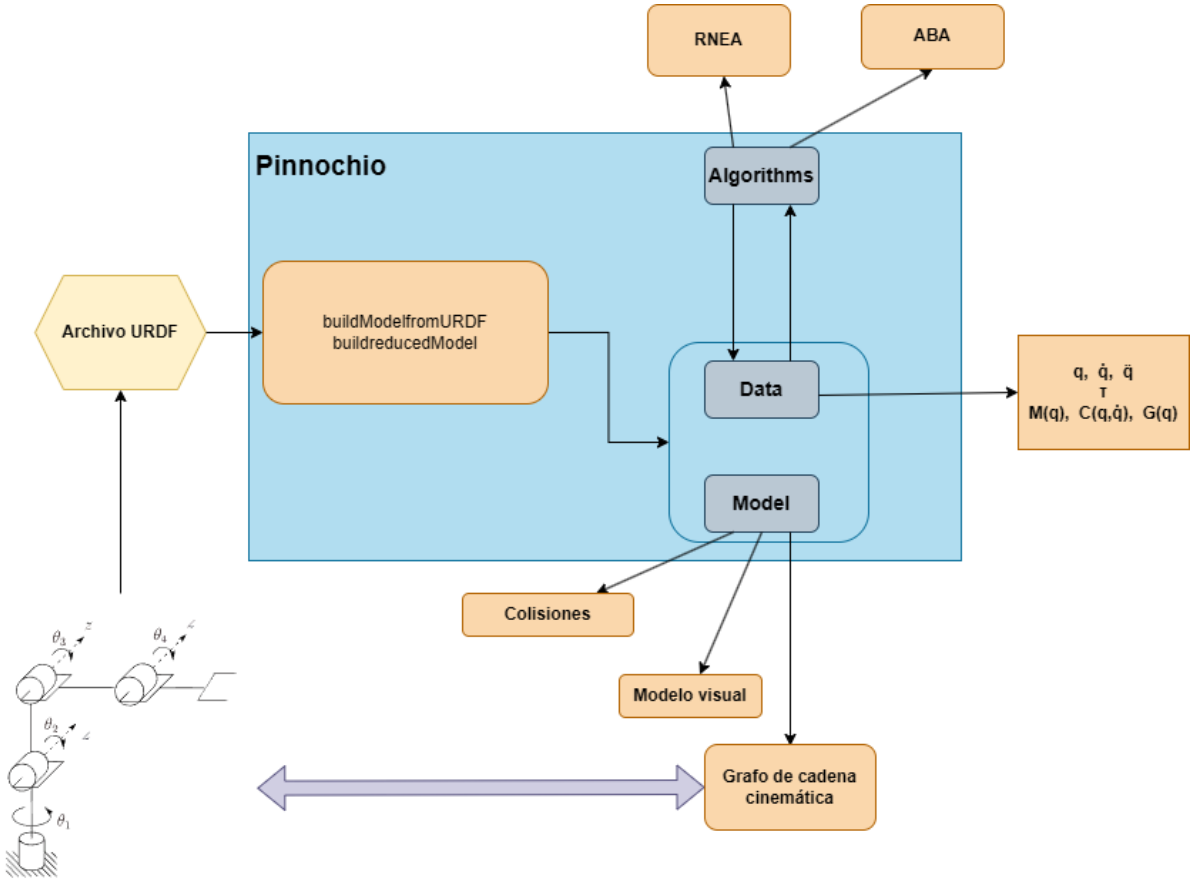


Figura 5.7: Diagrama general de los módulos usados de *Pinocchio*. Se pueden ver los módulos usados durante el proyecto y cómo interactúan entre ellos. El flujo comienza en el archivo .urdf, que describe el modelo del manipulador. A partir de este archivo, Pinocchio genera las estructuras Model y Data, que permiten interactuar con el robot. El objeto Model almacena la información del sistema como un grafo de articulaciones y cuerpos.

5.3.1. Desventajas de la librería

Si bien *Pinocchio* resuelve de forma eficiente y robusta la mayor parte de los problemas asociados al modelado cinemático y dinámico de manipuladores, presenta ciertas limitaciones derivadas de su enfoque de diseño. La biblioteca fue concebida principalmente para integrarse en sistemas de control en tiempo real, optimizando la ejecución sobre CPU y priorizando el bajo consumo de memoria por encima de la paralelización. En consecuencia, no dispone de una implementación nativa sobre GPU ni de soporte para cómputo paralelo de gran escala. Esto dificulta su utilización en contextos donde se requieren evaluaciones dinámicas en grandes batches (por ejemplo, en el entrenamiento de redes neuronales). Se verá que, integrar *Pinocchio* en un loop de optimización, introduce un cuello de botella cuando se trabaja con diferenciación automática y cálculo en GPU. La librería esta implementada en OpenMP, lo que permite, en teoría paralelizar los algoritmos implementados. No se profundizó en la paralelización de los algoritmos con el objetivo de evitar que el enfoque del proyecto se centrara en los aspectos computacionales del entrenamiento; en este caso fue suficiente con ajustar parámetros para que el cuello de botella sea tolerable (más detalles sobre esto en secciones posteriores).

5.4. PINN para el brazo robótico

A partir del marco teórico previamente desarrollado sobre manipuladores, dinámica y cinemática de *cadenas cinemáticas*, ya es posible formular el problema de aprendizaje informado por física aplicado al brazo robótico.

Retomando el problema de la *dinámica directa*, cuya formulación general se expresa como:

$$M(\mathbf{q})\,\ddot{\mathbf{q}} + C(\mathbf{q},\dot{\mathbf{q}})\,\dot{\mathbf{q}} + g(\mathbf{q}) = \boldsymbol{\tau},$$

que ya fue presentada en 5.1.9, la red neuronal buscará resolver esta dinámica a partir de los torques $\boldsymbol{\tau}$ y las *condiciones iniciales* $\mathbf{q}_0$ y $\dot{\mathbf{q}}_0$. En este punto, ya es posible hacer algunas apreciaciones sobre por qué tiene sentido abordar este problema desde una perspectiva de aprendizaje, y qué se espera que la red neuronal aprenda concretamente.

Dado que se pueden derivar las posiciones articulares $\mathbf{q}(t)$ para obtener velocidades y aceleraciones, y conociendo los torques $\boldsymbol{\tau}(t)$, el problema se reduce a identificar las funciones paramétricas que describen las matrices $M(\mathbf{q})$, $C(\mathbf{q},\dot{\mathbf{q}})$ y $g(\mathbf{q})$, que constituyen la dinámica subyacente.

Para sistemas simples, estas matrices pueden tener una forma analítica relativamente sencilla, lo que permite resolver el problema de forma directa. Sin embargo, a medida que aumentan los *grados de libertad* del manipulador, la formulación simbólica de estas expresiones se vuelve considerablemente más compleja. El número de términos crece rápidamente con la cantidad de articulaciones, y las expresiones resultantes incluyen productos cruzados y dependencias no lineales entre las posiciones y velocidades articulares.

Esta complejidad no solo incrementa el costo computacional de su cálculo en tiempo real, sino que también dificulta su análisis simbólico y la implementación eficiente de modelos dinámicos precisos. Frente a estas dificultades, un enfoque basado en redes neuronales permite aproximar directamente la dinámica a partir de datos de entrada-salida, evitando la necesidad de expresar analíticamente cada uno de estos términos.

En consecuencia, resulta útil explorar enfoques alternativos basados en aprendizaje automático, donde el modelo aprenda la dinámica directamente a partir de datos y principios físicos. Las *Physics-Informed Neural Networks* (PINNs presentadas en el capitulo 2) proporcionan un marco adecuado para este fin, combinando el conocimiento físico del sistema con la capacidad de aproximación universal de las redes neuronales. Conceptualmente este tipo de enfoques se centran en *aprender* las matrices M,C y g. La **interpretabilidad** de la red neuronal no es parte de los objetivos de este proyecto, pero tener presente esta idea ayuda a entender cómo está formulado el problema de aprendizaje.

5.4.1. Entrada/salida de la red

Se considera útil analizar inicialmente el sistema bajo un enfoque de caja negra. Bajo esta perspectiva, se definen explícitamente las variables de entrada que alimentan a la red neuronal:

$\mathbf{q_0} \in \mathbb{R}^n$	posiciones articulares
$\dot{\mathbf{q}}_\mathbf{0} \in \mathbb{R}^n$	velocidades articulares
$\boldsymbol{\tau} \in \mathbb{R}^n$	torques aplicados
$\mathbf{t_{eval}} = [t_1, t_2, \ldots, t_N]^\top,\ t_k \in [0, T]$	puntos de evaluación temporales

Entonces conceptualmente, las entradas de la red son las *condiciones iniciales*, los torques aplicados y los puntos temporales para los cuales queremos obtener el avance del sistema. La salida de la red retorna:

$$\hat{\mathbf{q}}(t_k) \in \mathbb{R}^n \quad \text{posiciones estimadas en cada } t_k$$
$$\hat{\dot{\mathbf{q}}}(t_k) \in \mathbb{R}^n \quad \text{velocidades estimadas en cada } t_k$$

La interpretación de la salida corresponde a la trayectoria resultante de aplicar los torques $\boldsymbol{\tau}$ a un manipulador, partiendo de un estado inicial definido por $\mathbf{q}_0$ y $\dot{\mathbf{q}}_0$, evaluada en los instantes de tiempo t_k. En un contexto de control, la entrada $\boldsymbol{\tau}$ representa la **acción de control**, mientras que la salida describe la evolución del estado del sistema en los distintos puntos temporales t_k.

Es importante destacar que la red neuronal predice el avance de las posiciones y velocidades articulares del manipulador, dadas las *condiciones iniciales* y los torques aplicados. Este avance es **único**, lo que refuerza la importancia de formular la predicción a partir de las variables articulares. Como se analizó previamente, si la trayectoria se definiera en función de la posición del *punto operativo* en el espacio cartesiano, podrían existir múltiples trayectorias articulares $\hat{\mathbf{q}}(t_k)$ que conduzcan a un mismo objetivo final ($f^{-1}(x)$ puede no ser única como se menciona en 5.1.4).

En la Figura 5.8 se ilustra el funcionamiento conceptual del sistema para un brazo robótico de dos articulaciones (dos *grados de libertad*) y tres puntos para la evaluación de la dinámica.

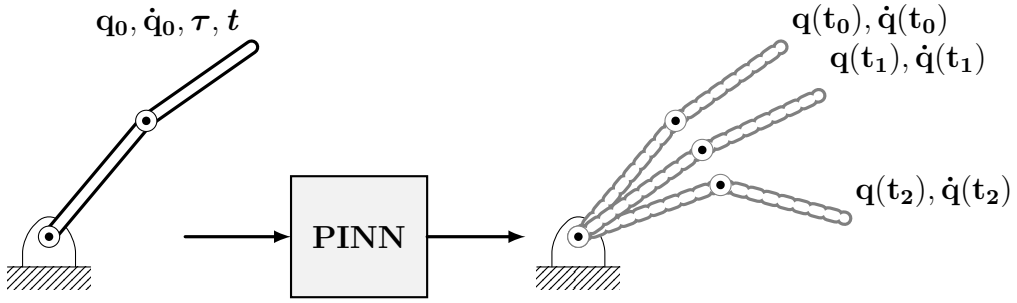


Figura 5.8: Esquema del funcionamiento de la PINN considerada como una *caja negra*. Las entradas corresponden a las *condiciones iniciales*, los torques aplicados $\boldsymbol{\tau}$ y el vector temporal $\mathbf{t}_{col}$. La dimensión de la salida de la red depende directamente de la cantidad de puntos de instantes de tiempo en los que se quiere evaluar la trayectoria, especificados en $\mathbf{t}_{col}$. En este caso particular, se le solicita a la red predecir la trayectoria completa en el intervalo definido. Más adelante se analizarán otras configuraciones posibles.

Es importante resaltar, y más adelante se detallará cómo se aborda, que la salida de la red en el instante t_0 debe coincidir con el estado inicial del brazo robótico. Es decir, la trayectoria estimada por la red debe contener explícitamente la configuración inicial del sistema. Esta condición constituye una de las *condiciones de borde* que deben ser correctamente tratadas durante el entrenamiento.

5.4.2. Funciones de pérdida

Como se ha mencionado anteriormente, las PINNs permiten incorporar conocimiento físico del sistema en el proceso de entrenamiento, integrando términos adicionales en la función de pérdida que reflejan la dinámica subyacente. En el caso del brazo robótico, la red

neuronal se entrena imponiendo explícitamente las identidades cinemáticas y dinámicas del manipulador. La pérdida total de la red se conforma de tres funciones:

$$\mathcal{L} = w_{\text{data}}\,\mathcal{L}_{\text{data}} + w_{\text{BC}}\,\mathcal{L}_{\text{BC}} + w_{phys}\,\mathcal{L}_{\text{phys}}$$

Estas tres funciones de costo buscan incorporar de forma equilibrada los distintos aspectos del entrenamiento de la PINN:

- **Pérdida de datos** ($\mathcal{L}_{\text{data}}$): Mide el error entre las predicciones del modelo y los datos de referencia (simulados o medidos). Permite que la red aprenda a reproducir trayectorias o torques conocidos cuando se dispone de ejemplos provenientes de un modelo dinámico o de un experimento real. Es útil para captar discrepancias entre un modelo y el manipulador real.
- **Condiciones de contorno** ($\mathcal{L}_{\text{BC}}$): Impone las condiciones iniciales o de frontera del sistema, asegurando que la solución predicha cumpla con los estados físicos observados al inicio de cada trayectoria (en este caso $q(t_0) = q_0$ y $\dot{q}(t_0) = \dot{q}_0$).
- **Pérdida física** ($\mathcal{L}_{\text{phys}}$): Garantiza que las predicciones de la PINN respeten las ecuaciones de movimiento del manipulador, es decir, que satisfagan el modelo dinámico del manipulador. Además se incluye un término para asegurar la consistencia física de las dos salidas de la red. Este término es el núcleo de la PINN, ya que introduce las leyes de la dinámica en el proceso de aprendizaje. Los puntos de colocación corresponden a instantes temporales en los que evaluamos esta pérdida, pudiendo agregar una gran cantidad para mejorar el entrenamiento.

En conjunto, estas tres pérdidas equilibran información proveniente de datos, restricciones físicas y condiciones iniciales, permitiendo que el modelo aprenda una representación consistente tanto con la física como con la observación empírica. Se ahondará a continuación sobre cada uno de estos términos y como fueron implementados en este proyecto.

Pérdida física

Este término es el encargado del cumplimiento de la dinámica del sistema. Modelamos la dinámica en forma de primer orden con el estado

$$\mathbf{x} = \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \end{bmatrix} \tag{5.24}$$

Luego se tiene que:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{q}} \\ \mathbf{M}(\mathbf{q})^{-1}\big(\boldsymbol{\tau} - \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\,\dot{\mathbf{q}} - \mathbf{g}(\mathbf{q})\big) \end{bmatrix} \tag{5.25}$$

Las *pérdidas físicas* surgen de los residuos de estas dos ecuaciones:

$$\mathbf{r}_{\text{kyn}}(t) = \tfrac{\partial \hat{\mathbf{q}}}{\partial t}(t) - \hat{\dot{\mathbf{q}}}(t), \qquad \mathbf{r}_{\text{dyn}}(t) = \mathbf{M}(\hat{\mathbf{q}})\,\tfrac{\partial^2 \hat{\mathbf{q}}}{\partial t^2} + \mathbf{C}(\hat{\mathbf{q}}, \tfrac{\partial \hat{\mathbf{q}}}{\partial t})\,\tfrac{\partial \hat{\mathbf{q}}}{\partial t} + \mathbf{g}(\hat{\mathbf{q}}) - \boldsymbol{\tau} \tag{5.26}$$

Las pérdidas asociadas se definen como:

$$\mathcal{L}_{\text{ODE}} \;=\; w_{\text{kyn}}\,\big\|\mathbf{r}_{\text{kyn}}\big\|_2^2 \;+\; w_{\text{dyn}}\,\big\|\mathbf{r}_{\text{dyn}}\big\|_2^2 \tag{5.27}$$

promediadas sobre los puntos de colocación en el tiempo. Entonces a partir de las predicciones de la red $(\hat{\mathbf{q}}, \hat{\dot{\mathbf{q}}})$ se obtienen las derivadas $\frac{\partial^2 \hat{\mathbf{q}}}{\partial t^2}$, $\frac{\partial \hat{\mathbf{q}}}{\partial t}$ y se calculan los residuos. El término $\mathbf{r}_{\text{kyn}}(t)$ se puede interpretar como un término que introduce la coherencia física de la red (la derivada de la salida $\hat{\mathbf{q}}$, $\frac{\partial \hat{q}}{\partial t}$ debe ser igual a la salida $\hat{\dot{\mathbf{q}}}$).

Pérdida de condiciones de borde (BC)

Este término se encarga de garantizar que las soluciones obtenidas por la red neuronal respeten las condiciones iniciales (más en general las condiciones de borde). En el contexto del manipulador robótico, las condiciones de borde corresponden a los estados articulares conocidos en los instantes iniciales:

$$\mathbf{q}(t_0) = \mathbf{q}_0, \qquad \dot{\mathbf{q}}(t_0) = \dot{\mathbf{q}}_0.$$

La red debe reproducir exactamente estos valores, asegurando que las trayectorias generadas sean consistentes con las condiciones físicas impuestas al sistema. De este modo, se penalizan las desviaciones entre las salidas de la red y los valores iniciales reales. En la práctica las condiciones de contorno pueden imponerse en la red de dos maneras conceptualmente distintas:

- **Hard constraint:** Las condiciones iniciales se imponen directamente en la arquitectura o en la formulación del modelo, de modo que la red las satisface de manera exacta. Por ejemplo, puede parametrizarse la salida para garantizar por construcción que $\hat{\mathbf{q}}(t_0) = \mathbf{q}_0$. Este enfoque asegura cumplimiento exacto de las condiciones iniciales, pero restringe la flexibilidad del modelo y puede dificultar la optimización. Las condiciones duras se pueden agregar a la red mediante:

$$\hat{\mathbf{q}}(t) = \mathbf{q}_0 + t\,\dot{\mathbf{q}}_0 + t^2\,\boldsymbol{\phi}(t;\theta), \qquad \hat{\dot{\mathbf{q}}}(t) = \dot{\mathbf{q}}_0 + t\,\boldsymbol{\psi}(t;\theta) \tag{5.28}$$

  donde $\boldsymbol{\phi}(t;\theta)$ y $\boldsymbol{\psi}(t;\theta)$ son las salidas aprendidas por la red. Esta formulación garantiza que en $t = 0$ se cumplan de manera exacta las condiciones iniciales:

$$\hat{\mathbf{q}}(0) = \mathbf{q}_0, \qquad \hat{\dot{\mathbf{q}}}(0) = \dot{\mathbf{q}}_0.$$

  En [11] se explora en detalle esta formulación para imponer condiciones de borde de este tipo en PDEs (este enfoque también sirve para PINNs y otros problemas). En general, este método es recomendable cuando resulta sencillo incorporar las condiciones iniciales en la parametrización del modelo, y cuando la restricción estructural no compromete la capacidad de la red para aproximar la dinámica subyacente.

- **Soft constraint:** Las condiciones de borde se incorporan dentro de la función de pérdida mediante un término de penalización, como en

$$\mathcal{L}_{\text{BC}} = \|\hat{\mathbf{q}}(t_0) - \mathbf{q}_0\|^2 + \left\|\hat{\dot{\mathbf{q}}}(t_0) - \dot{\mathbf{q}}_0\right\|^2 \tag{5.29}$$

  permitiendo que el modelo viole levemente la condición mientras optimiza el resto de los términos (por ejemplo, la dinámica física o los datos). Este enfoque es más flexible y puede facilitar la convergencia en entrenamientos complejos, aunque a costa de perder cumplimiento exacto de las condiciones iniciales.

En este trabajo se exploraron ambos enfoques, con especial énfasis en el enfoque *hard bc* debido a que para una tarea de control resulta muy importante que la condición inicial se cumpla. Además este enfoque tiene el agregado de alivianar la tarea de optimización con varias funciones de costo (posteriormente se detallan resultados y estrategias tenidos en cuenta). Otra razón relevante que sustenta el enfoque *Hard-BC* es que la red neuronal evalua la dinámica en los puntos de colocación (t_k), esto lleva a que naturalmente la función de pérdida $\mathcal{L}_{phys}$ sea mas influyente que $\mathcal{L}_{BC}$, simplemente porque se evalúa en más puntos. Por esto el modelo tiende a ajustar más en función de $\mathcal{L}_{ODE}$ que para $\mathcal{L}_{BC}$. Esto se puede ajustar con los pesos asociados pero el enfoque duro permite desentenderse de este balanceo de pérdidas (es complicado a priori saber cómo balancear las pérdidas por lo que resulta beneficioso poder evitarlo).

Pérdida en datos

Este término mide el error entre las trayectorias predichas por la red y aquellas obtenidas a partir de un modelo físico o simulación. En este trabajo, las trayectorias de referencia $(\mathbf{q}_{\text{sim}}(t), \dot{\mathbf{q}}_{\text{sim}}(t))$ se obtienen mediante el algoritmo *Articulated Body Algorithm* (ABA), implementado en la librería **Pinocchio**.

El ABA resuelve el problema de **dinámica directa** del manipulador, calculando las aceleraciones articulares $\ddot{\mathbf{q}}$ a partir de las variables $(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau})$ y del modelo dinámico:

$$M(\mathbf{q})\,\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\,\dot{\mathbf{q}} + g(\mathbf{q}) = \boldsymbol{\tau}$$

A partir de las aceleraciones, se integran numéricamente las ecuaciones del movimiento para obtener las trayectorias $\mathbf{q}_{\text{sim}}(t)$ y $\dot{\mathbf{q}}_{\text{sim}}(t)$, que se utilizan como datos de entrenamiento supervisado. Esta integración se realiza mediante el módulo correspondiente de **Pinocchio**, que permite integrar ecuaciones diferenciales respetando la naturaleza geométrica de cada variable (rotaciones, traslaciones o articulaciones prismáticas). Esto se conoce como integración en *variedades*, y garantiza que las configuraciones articulares permanezcan sobre sus espacios de movimiento válidos evitando errores númericos.

La pérdida se define entonces como el error cuadrático medio (MSE) entre las predicciones de la red $(\hat{\mathbf{q}}(t), \hat{\dot{\mathbf{q}}}(t))$ y las trayectorias simuladas:

$$\mathcal{L}_{\text{data}} = \frac{1}{N}\sum_{i=1}^{N}\left(\|\hat{\mathbf{q}}(t_i) - \mathbf{q}_{\text{sim}}(t_i)\|^2 + \left\|\hat{\dot{\mathbf{q}}}(t_i) - \dot{\mathbf{q}}_{\text{sim}}(t_i)\right\|^2\right) \tag{5.30}$$

Este término obliga a la red a reproducir trayectorias físicamente coherentes con la simulación del modelo dinámico del robot, sirviendo como una forma de calibración o supervisión externa. En general, no es recomendable obtener los datos para esta función de costo a partir de un modelo simulado (en este caso es la única opción) ya que este término en la función de costo está pensado para captar irregularidades del modelo o diferencias sutiles entre el modelo y el manipulador real.

5.4.3. Normalización

La red recibe como entrada el conjunto $(\mathbf{t}, \mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0}, \tau)$ y construye el vector

$$x = [\, q_n,\, \dot{q}_n,\, \tau_n,\, \tilde{t}\,] \in \mathbb{R}^{3\ \times \text{DOF}+1} \tag{5.31}$$

donde cada variable se normaliza en función de los límites físicos del robot proporcionados por el fabricante. Este preprocesamiento convierte todas las entradas en magnitudes adimensionales y acotadas dentro del intervalo $[-1, 1]$, lo que mejora la estabilidad numérica y evita que una variable domine sobre las demás durante el entrenamiento.

Por ejemplo, para las posiciones articulares se utiliza la siguiente normalización:

$$q_n = 2 \cdot \frac{q_0 - q_{\text{mín}}}{q_{\text{máx}} - q_{\text{mín}}} - 1 \tag{5.32}$$

donde $q_{\text{mín}}$ y $q_{\text{máx}}$ son los valores mínimo y máximo permitidos para cada articulación. De este modo, cuando $q_0 = q_{\text{mín}}$ el valor normalizado es -1, cuando $q_0 = q_{\text{máx}}$ es 1, y el punto medio del rango se corresponde con 0. El mismo criterio se aplica a las velocidades $\dot{q}_0$ y los torques τ. El tiempo t se normaliza en el intervalo $[0, 1]$

5.4.4. Generación de condiciones iniciales

Cobertura del espacio de trabajo

Un aspecto clave en la generación de condiciones iniciales es garantizar una cobertura adecuada del *espacio de trabajo* del robot, es decir, del conjunto de posiciones y orientaciones que puede alcanzar el manipulador dadas las limitaciones geométricas y articulares.

En un manipulador de múltiples grados de libertad, diferentes combinaciones de ángulos articulares pueden producir trayectorias similares, mientras que otras regiones del espacio cartesiano pueden ser inaccesibles debido a restricciones mecánicas, límites de giro o autocolisiones. Por tanto, una cobertura deficiente del espacio de trabajo provoca que el modelo sólo vea una fracción del comportamiento posible, deteriorando su capacidad de generalización. En el contexto de la PINN, esto se traduce en un sesgo del entrenamiento hacia ciertas zonas del espacio articular, con una pobre representación de las trayectorias extremas o de configuraciones menos frecuentes.

En el OpenManipulator-X completo (cuatro grados de libertad activos) el espacio articular es de dimensión elevada, lo que hace inviable recorrerlo de forma densa: una grilla de N puntos por articulación crece como N^4. Por ejemplo, una resolución moderada de $N = 50$ implicaría $50^4 = 6{,}25 \times 10^6$ combinaciones posibles, lo cual resulta computacionalmente prohibitivo para validar con chequeos de colisión y singularidad. Además esto último se traduce en un entrenamiento más largo, debido a que sumado a la cantidad de *condiciones iniciales* se debe contemplar que la PINN debe generalizar también en diferentes rangos de torques. Este es uno de los principales desafíos del entrenamiento, desde un punto de vista práctico esto significa que la red debe capturar la dinámica completa del sistema bajo cualquier combinación físicamente posible de $(q_0, \dot{q}_0, \tau)$ dentro del dominio de interés. Esto exige más puntos de entrenamiento, un mayor número de *collocation points* y, en general, un tiempo de convergencia más largo, dado que la red necesita ajustar sus pesos para representar adecuadamente las distintas regiones de ese espacio. La PINN debe aproximar una función con fuerte variabilidad en varios ejes (estados y fuerzas), lo que incrementa significativamente la complejidad del aprendizaje. Para mitigar esta explosión combinatoria que resulta en tiempos de entrenamientos excesivamente largos, se decidió **bloquear dos grados de libertad proximales** del brazo. Esta simplificación reduce el espacio articular a dos variables principales, manteniendo la movilidad esencial en el plano de trabajo del manipulador. **En particular, en este documento se presenta el modelo con las dos últimas articulaciones bloqueadas.** El bloqueo de estas articulaciones se hizo mediante la función *buildReducedModel* de **Pinocchio**, que ya fue explicada en 5.3.

Gracias a esta reducción, fue posible generar una grilla fina de condiciones iniciales para las posiciones $\mathbf{q} = [q_1, q_2]^T$ (recordar que la *articulación 0* es la base, que no forma parte del modelo del manipulador a efectos prácticos).

Esta grilla se recorre en su totalidad y, para cada punto $[q_1, q_2]$, se sortean uniformemente las velocidades $[\dot{q}_1, \dot{q}_2]$. Esta estrategia permite cubrir de manera sistemática el espacio de configuraciones, que es donde las matrices dinámicas $\boldsymbol{M(q)}$, $\boldsymbol{C(q, \dot{q})}$ y $\boldsymbol{G(q)}$ presentan mayor variabilidad, mientras que las velocidades, al influir principalmente de forma lineal en los términos de Coriolis, pueden ser muestreadas aleatoriamente sin perder tanta representatividad. De este modo se obtiene un conjunto de estados suficientemente diverso para capturar las principales no linealidades del sistema sin necesidad de una exploración exhaustiva del espacio completo $(\boldsymbol{q}, \dot{\boldsymbol{q}})$. Cada muestra $(\boldsymbol{q_0}, \dot{\boldsymbol{q}}_{\boldsymbol{0}})$ se valida a través de los siguientes filtros:

- **Límites físicos:** el estado debe permanecer dentro de $[q_{\text{mín}}, q_{\text{máx}}]$.

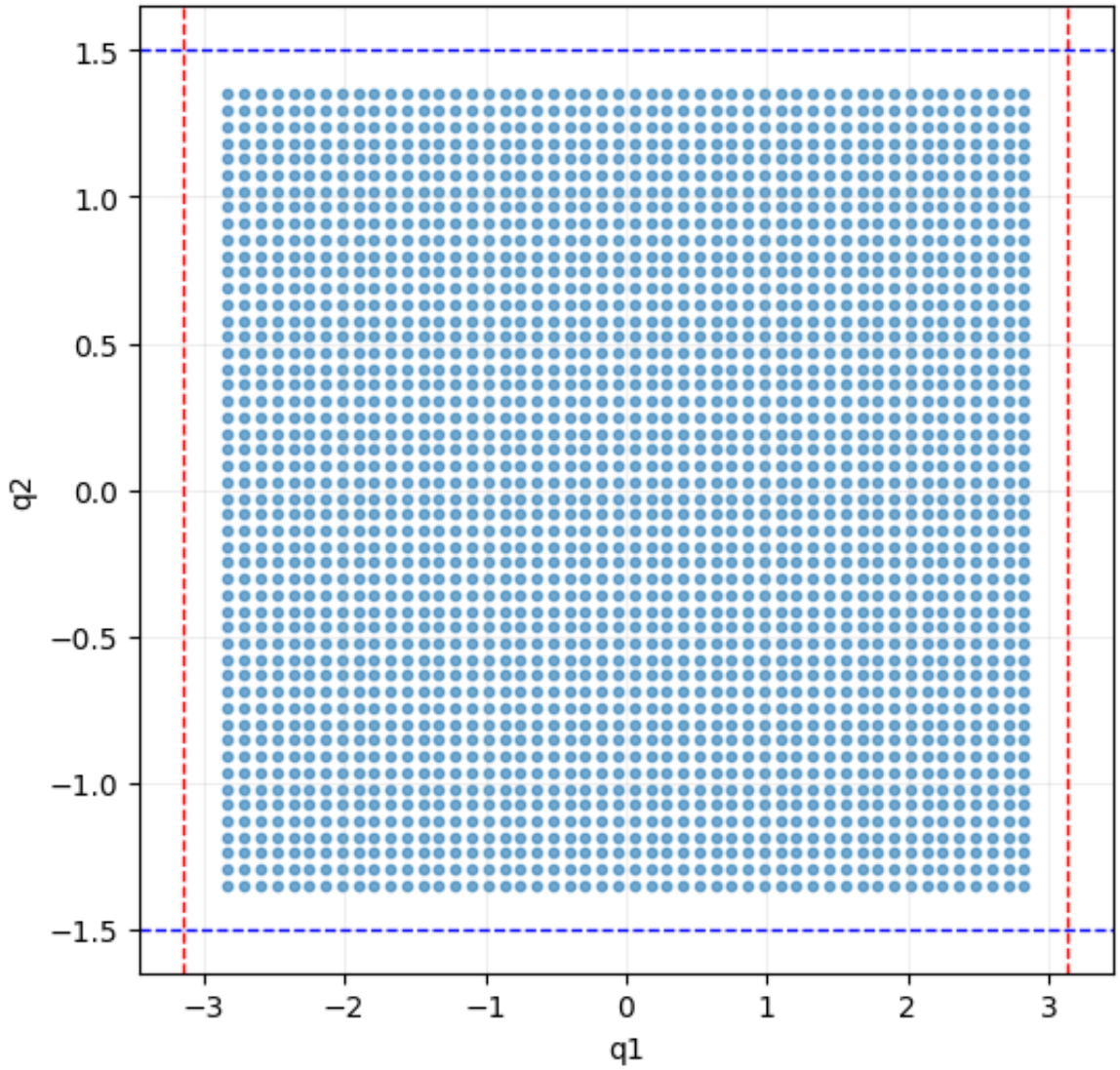


Figura 5.9: Grilla de puntos tomada para $\mathbf{q} = [q_1, q_2]^T$. Las lineas punteadas marcan los límites físicos del robot. La resolución se tomó con fines ilustrativos como N = 50. Esta grilla genera alrededor de 1800 puntos

- **Colisión:** se descartan configuraciones en contacto con el piso o con otros enlaces.
- **Singularidades:** se evalúa la condición del Jacobiano del manipulador, rechazando configuraciones con número de condición elevado ($\kappa(J) > \kappa_{\text{máx}}$).
- **Guarda dinámico:** se comprueba que el estado adelantado $q^+ = q_0 + \dot{q}_0\,\Delta t$ siga siendo válido dentro de los límites.

Este proceso garantiza que cada condición inicial sea físicamente coherente. El resultado final es un conjunto de pares $(\boldsymbol{q}, \dot{\boldsymbol{q}})$ que cubren uniformemente el dominio operativo del manipulador y pueden combinarse con torques constantes para formar las entradas del entrenamiento (*condiciones iniciales*) de entrenamiento.

5.4.5. Muestreo en **t**

Para el muestreo temporal se empleó la técnica *Latin Hypercube Sampling* (LHS), la cual garantiza una cobertura uniforme del dominio $[0, T]$. Como fue explicado en 2, este método resulta especialmente útil cuando se requiere evaluar la dinámica en múltiples puntos de colocación (*collocation points*), ya que asegura que las muestras en **t** exploren todo el rango temporal de forma balanceada. Estos puntos también son remuestreados cada epoca para asegurar el cubrimiento del espacio temporal lo cual también contribuye a que la red no sobreajuste a un conjunto de puntos específico y generalice para todo el espacio temporal.

5.4.6. Optimización

Se evaluaron dos métodos de optimización: **Adam** y **L-BFGS**. Aunque L-BFGS puede lograr buenas convergencias en problemas suaves, la PINN del brazo robótico combina distintas pérdidas (condiciones iniciales y dinámica), lo que la hace sensible a diferencias de escala y curvatura. En la práctica, **Adam** mostró un entrenamiento más estable, rápido y con mejor capacidad de generalización, mientras que L-BFGS presentó oscilaciones e inestabilidad numérica al calcular derivadas de orden superior. Por este motivo, se adoptó **Adam** como optimizador principal en los resultados presentados. Se exploró la variante Adam + LBFGS, lo que que no dio resultados convincentes y aumentó considerablemente el tiempo de optimización.

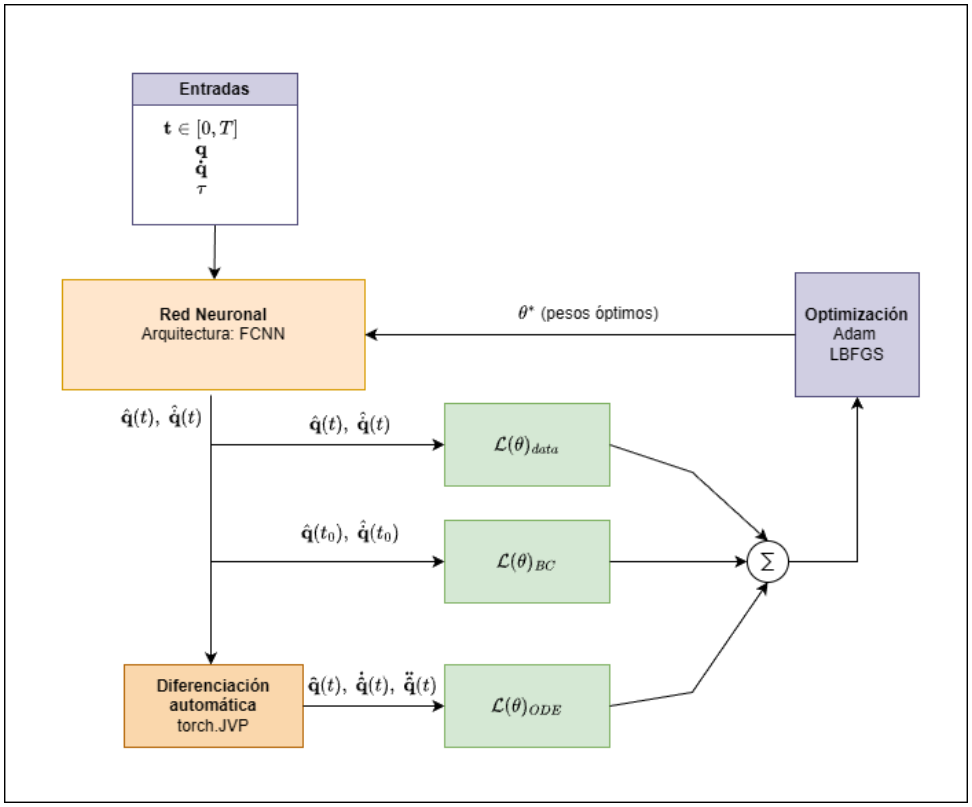


Figura 5.10: Diagrama esquemático de la implementación. Se puede ver que el diagrama no difiere mucho del diagrama para Van der Pol en la Figura 2.5. En este caso todas las entradas y salidas son vectoriales, por ende se cambia el método de derivación automática (JVP)

5.4.7. Esquema general

Con todos los componentes previamente definidos, es posible comprender el funcionamiento general del sistema propuesto. En particular, se puede visualizar cómo interactúan los distintos términos de la función de pérdida con la red neuronal y el optimizador durante el proceso de entrenamiento. En la figura 5.10 se presenta el diagrama completo del sistema que se implementó.

5.5. Detalles de la implementación

En esta sección se abordan algunos detalles relevantes de la implementación del modelo. Se destacan, principalmente, aquellos aspectos que definen la estructura del modelo, su interpretación y la forma en que se plantea el entrenamiento. Se han omitido partes de la implementación que no resultan pertinentes para este propósito.

5.5.1. Diferenciabilidad y backpropagation.

Para entrenar por propagación hacia atrás (backpropagation), la función de costo debe ser diferenciable *con respecto a los parámetros de la red*. Por la regla de la cadena, esto exige que cada componente sea (al menos) diferenciable respecto de las *salidas intermedias* que la red produce. En este caso, la PINN predice $\mathbf{q}(t)$ y $\dot{\mathbf{q}}(t)$; de allí obtenemos $\ddot{q}(t)$ mediante diferenciación automática. Cualquier bloque que mapee $(q, \dot{q}, \ddot{q})$ a cantidades usadas en la función de costo *también* debe ser diferenciable. Operaciones no diferenciables rompen la cadena de gradientes, haciendo que el aprendizaje no sea posible.

5.5.2. Requisito de diferenciabilidad a RNEA.

El término dinámico más natural de la pérdida es el *residuo de la ecuación de dinámica*:

$$r(q, \dot{q}, \ddot{q}, \tau) = M(q)\,\ddot{q} + C(q, \dot{q})\,\dot{q} + G(q) - \tau, \qquad \mathcal{L}_{\text{ODE}} = \|r\|_2^2 \tag{5.33}$$

que penaliza violar la dinámica del manipulador. Computar $r(\cdot)$ eficientemente y con precisión conduce al algoritmo RNEA (Recursive Newton–Euler Algorithm), tal como lo implementa Pinocchio. La llamada directa a `pin.rnea` ejecuta C++ sobre CPU y devuelve un arreglo `numpy` *desacoplado* del grafo de autograd de PyTorch. Es decir: PyTorch no conoce $\partial\tau/\partial q$, $\partial\tau/\partial\dot{q}$ ni $\partial\tau/\partial\ddot{q}$, por lo que no puede propagar gradientes hacia las salidas de la red que generaron $(q, \dot{q}, \ddot{q})$.

5.5.3. Uso de RNEA dentro de la función de costo.

Se define un operador $\tau(q, \dot{q}, \ddot{q})$ diferenciable y se agrega en la pérdida:

$$\mathcal{L}_{\text{ODE}}(\theta) = \left\| \tau\big(q, \dot{q}, \ddot{q}\big) - \tau_{\text{ref}} \right\|^2 \quad \text{con} \quad \tau\big(q, \dot{q}, \ddot{q}\big) = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) \tag{5.34}$$

Se observa que $\mathcal{L}_{ODE}$ depende de los parámetros θ, estos son los parámetros de la red neuronal (pesos y bias en cada capa). La dependencia sobre estos parámetros viene de $\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}$. Esto es inmediato para $\mathbf{q}$, ya que esta es una salida de la red, luego $\dot{\mathbf{q}}$ y $\ddot{\mathbf{q}}$ se obtienen mediante diferenciación automática de la salida $\mathbf{q}$. Entonces, para que la función de costo sea compatible con la propagación hacia atrás, es necesario que sea diferenciable con

respecto a los parámetros de la red, que es equivalente a que sea diferenciable con respecto a $\mathbf{q}_{\boldsymbol{\theta}}$, $\dot{\mathbf{q}}_{\boldsymbol{\theta}}$, $\ddot{\mathbf{q}}_{\boldsymbol{\theta}}$ (donde el subíndice θ hace explícita la dependencia con los parámetros de la red). De la formulación anterior, esto significa que es necesario algún mecanismo para derivar la expresión $\tau\big(q, \dot{q}, \ddot{q}\big)$. Como fue explicado en secciones anteriores, el vector $\tau\big(q, \dot{q}, \ddot{q}\big)$ es exactamente lo que retorna el algoritmo RNEA. Ahora, para volver RNEA *diferenciable* en PyTorch, se implementó una función (wrapper, basada en [1]) utilizando la función *computeRNEADerivatives* de Pinocchio(que retorna las derivadas necesarias para un τ calculado mediante RNEA). La función implementada tiene dos etapas:

1. En el **forward**, para una muestra con valores $(q, \dot{q}, \ddot{q})$, se calcula el torque resultante

$$\tau = \text{RNEA}(q, \dot{q}, \ddot{q}) \tag{5.35}$$

   y, además, se obtienen las derivadas analíticas mediante `pin.computeRNEADerivatives`:

$$\frac{\partial \tau}{\partial q}, \qquad \frac{\partial \tau}{\partial \dot{q}}, \qquad \frac{\partial \tau}{\partial \ddot{q}} \text{ (esto es } M(q)) \tag{5.36}$$

   Estas tres matrices jacobianas se guardan para ser utilizadas durante el paso backward.

2. En el **backward**, la función recibe el gradiente:

$$g_\tau = \frac{\partial \mathcal{L}}{\partial \tau} \tag{5.37}$$

   Luego aplica la regla de la cadena para propagar el gradiente hacia las entradas:

$$\frac{\partial \mathcal{L}}{\partial q} = \Big(\frac{\partial \tau}{\partial q}\Big)^\top g_\tau, \qquad \frac{\partial \mathcal{L}}{\partial \dot{q}} = \Big(\frac{\partial \tau}{\partial \dot{q}}\Big)^\top g_\tau, \qquad \frac{\partial \mathcal{L}}{\partial \ddot{q}} = \Big(\frac{\partial \tau}{\partial \ddot{q}}\Big)^\top \tag{5.38}$$

3. Finalmente, los tres gradientes obtenidos se devuelven al grafo computacional de PyTorch, permitiendo que la propagación hacia atrás continúe de forma automática a través del resto de la PINN.

En la Figura 5.11 se puede ver un diagrama de cómo funciona la implementación del wrapper de PyTorch para el algoritmo RNEA.

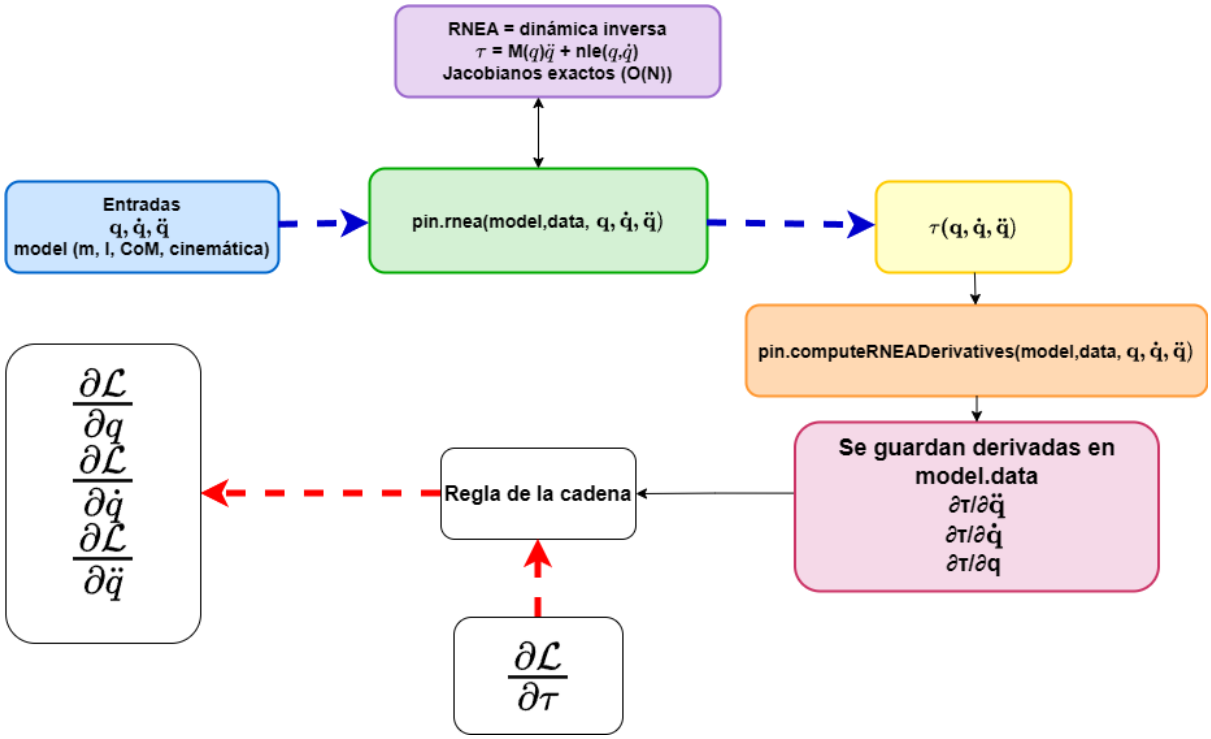


Figura 5.11: Diagrama de la función implementada para que el algoritmo RNEA propague correctamente los gradientes y permita que la *backpropagation* de PyTorch pase por la función de pérdida asociada a la dinámica y pueda derivar hasta los parámetros de la red. En rojo puede verse el camino *backward* y en azul el *forward*

5.5.4. Balance de pérdidas

En la teoría resulta sencillo definir una *función de costo compuesta* del tipo:

$$\mathcal{J}(\theta) = \sum_{i=1}^{m} w_i \, \mathcal{L}_i(\theta) \tag{5.39}$$

donde cada término $\mathcal{L}_i$ busca optimizar un objetivo distinto (por ejemplo, ajustar datos, respetar la dinámica, mantener suavidad o cumplir límites). Sin embargo, en la práctica este enfoque presenta varias dificultades. Por un lado, los distintos objetivos pueden generar **gradientes que compiten entre sí**, lo que ralentiza o incluso impide la convergencia. Además, si los términos tienen **escalas numéricas muy diferentes**, el entrenamiento se vuelve mal condicionado y algunos objetivos dominan sobre otros. También es necesario ajustar con cuidado los **pesos** w_i para equilibrar la contribución de cada término, lo que suele requerir mucha experimentación. Por último, al combinar varios objetivos, la superficie de optimización se vuelve más **no convexa**, haciendo el descenso por gradiente menos estable. Esto último es algo que afecta particularmente al optimizador L-BFGS (este optimizador está basado en la estimación de la hessiana y el supuesto de que la *superficie de optimización* es regular) y una de las razones por las que se cree que este último no tuvo los resultados conseguidos con Adam. En resumen, aunque la idea de una función de costo compuesta es conceptualmente simple, su aplicación práctica exige una buena normalización y un diseño cuidadoso de cada término para que los gradientes sean útiles y conduzcan a una convergencia estable. Más allá de los problemas mencionados, es importante destacar que las funciones de pérdida que se emplean en este trabajo son coherentes entre sí. Esto significa que todas persiguen un mismo objetivo físico y

numérico: lograr que la red aprenda a predecir trayectorias $(q, \dot{q})$ que sean dinámicamente consistentes con el modelo del manipulador.

Las pérdidas no compiten entre sí sino que se **complementan**. Cada término refuerza un aspecto distinto del mismo comportamiento físico, contribuyendo al aprendizaje global de una representación que sea tanto precisa como físicamente válida. Para evitar que un término en la función de costo domine debido a la escala númerica de las variables involucradas, se normalizaron todas las funciones de costo de manera que estén todas en la misma escala. La aplicación de esta normalización resulta sencilla, aunque requiere precaución respecto a qué términos deben ser normalizados. En esencia, no es posible normalizar directamente la salida de la red, sino únicamente los términos involucrados en las funciones de costo. El diagrama 5.12 ilustra el flujo de información desde las salidas de la red neuronal hasta las distintas funciones de pérdida.

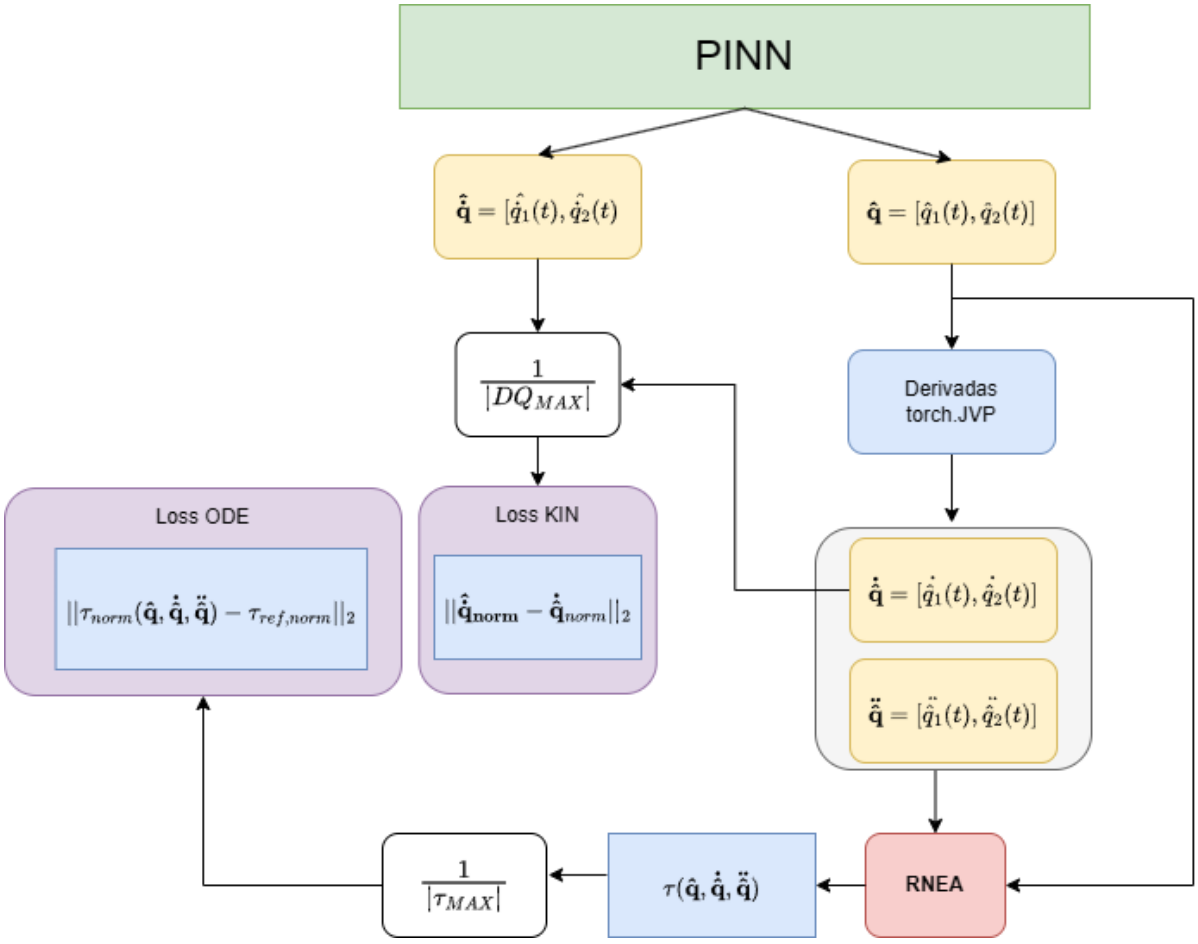


Figura 5.12: Esquema del manejo de las entradas a las funciones de costo. La normalización se hace previo al cálculo de las mismas usando los limites físicos del robot. Estos limites físicos se almacenan en las variables globales τ_{MAX} y DQ_{MAX}. Recordar que τ_{ref} es el torque de entrada a la red. Observar que $\hat{q}(t)$ no es lo mismo que $\hat{\dot{q}}(t)$ siendo esta última la derivada de una salida de la red.

El objetivo principal de este esquema es mostrar cómo se implementa una normalización adecuada de dichas funciones, con el fin de evitar que la escala de los términos involucrados se convierta en un factor determinante durante la etapa de optimización. La necesidad de dicha normalización surge a partir del monitoreo de los gradientes de las funciones de pérdida, donde se pudo observar que ciertas componentes dominaban sobre otras debido a diferencias de escala. Sin la normalización, los términos dependientes de $\ddot{\mathbf{q}}$ y $\boldsymbol{\tau}$ tienden a dominar. Se omitió la función $\mathcal{L}_{data}$ por razones que se explican en la sección 5.5.5

5.5.5. Pérdida en datos

Este trabajo no contó con el manipulador físico, sino que se emplearon simulaciones. En caso de haber contado con el brazo real, este término de pérdida consistiría en comparar las trayectorias medidas $(\mathbf{q}(t), \dot{\mathbf{q}}(t))$ bajo un torque aplicado $\boldsymbol{\tau}(t)$ con las predicciones de la red para distintas condiciones iniciales. Al no disponer de esos datos, la única alternativa fue resolver la *dinámica directa* del manipulador para generar trayectorias sintéticas de referencia. Esta integración se realizó mediante el algoritmo ABA (*Articulated Body Algorithm*), que entrega $\ddot{\mathbf{q}}$ a partir de $(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau})$ y de ahí se integran velocidades y posiciones a lo largo del tiempo, usando el módulo integrador de **Pinocchio**.

Sin embargo, desde el punto de vista físico y del aprendizaje, este enfoque impone esencialmente las mismas restricciones que la ecuación de movimiento usada en RNEA (inversa): ambos algoritmos provienen del mismo modelo dinámico de Euler–Lagrange, sólo que aplican la ecuación en sentido opuesto. Por tanto, al entrenar con trayectorias generadas por ABA, la red está aprendiendo sobre los mismos residuos dinámicos que ya se penalizan con el término $\mathcal{L}_{\text{ODE}}$ basado en RNEA. Además, el uso de ABA introduce dificultades numéricas adicionales: la integración paso a paso requiere elegir un paso temporal Δt suficientemente pequeño para mantener estabilidad, lo que amplifica errores de redondeo y puede hacer el gradiente más ruidoso o inestable durante el entrenamiento. En otras palabras, si el objetivo de la PINN es satisfacer directamente la ecuación de movimiento a través de RNEA, resolver la dinámica por ABA aporta poca información nueva y añade complejidad computacional y de integración. Esta función de costo se implementó pero no ofreció mejoras claras al modelo, mientras que introdujo otras complicaciones a la hora de entrenar. Por esta razón en la mayoría de los modelos que entrenados **no se tomó en cuenta esta función de costo**. En 5.13 se puede ver como se implementó.

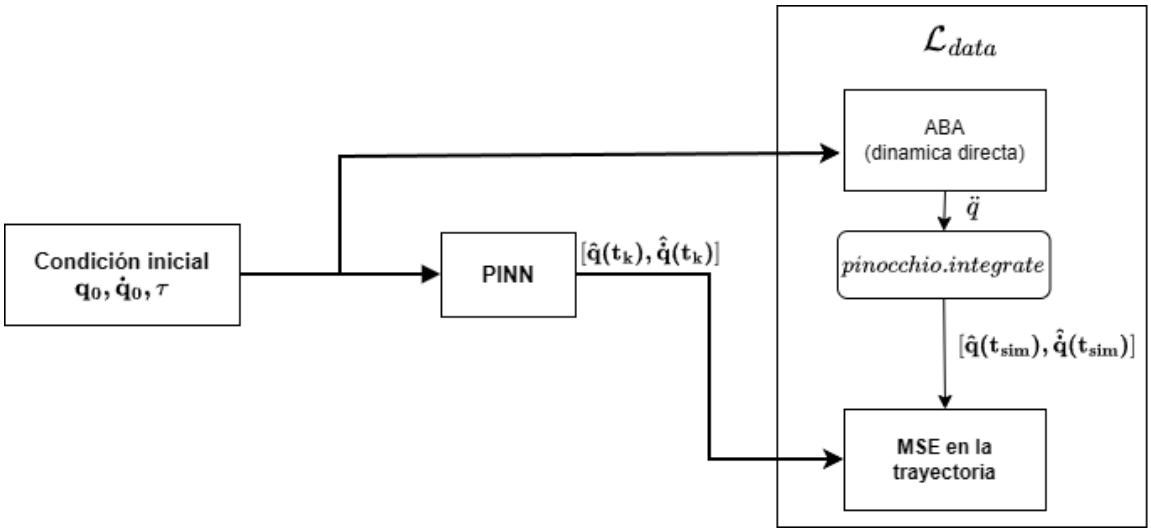


Figura 5.13: Diagrama de implementación de la función de pérdida en los datos simulados.

5.6. Datos

Esta sección refiere a los datos utilizados para el entrenamiento, teniendo en cuenta que no se incluye la función $\mathcal{L}_{DATA}$. Estos datos corresponden a los puntos que se alimentan a la red para la evaluación de las funciones de pérdida. Ya que la red no se entrena con trayectorias obtenidas mediante simulación o medición, el proceso de entrenamiento es *no supervisado*: la red ajusta sus parámetros únicamente en función de las condiciones

iniciales y de las restricciones físicas impuestas por la ecuación de movimiento. Por eso cuando se menciona a los *datos*, esto refiere al conjunto de puntos con el que se alimenta la red durante el entrenamiento.

Como ya fue mencionado, el conjunto de datos se genera a partir de una grilla de puntos $\mathbf{q_0} = [q_{0_1}, q_{0_2}]$ y luego se sortean las velocidades $\dot{\mathbf{q_0}} = [\dot{q}_{0_1}, \dot{q}_{0_2}]$ uniformemente. Posteriormente, para este conjunto $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0})$ se hacen todos los chequeos de validez mencionados. Los torques $\boldsymbol{\tau} = [\tau_1, \tau_2]$ se sortean también uniformemente y los puntos de colocación $\mathbf{t_{col}} \in [\mathbf{0}, \mathbf{T}]$ se sortean usando LHS. Es importante hacer la separación entre el conjunto de los puntos $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0})$ y los $\boldsymbol{\tau}$ y $\mathbf{t_{col}}$ ya que los primeros permanecen fijos, debido a que constituyen configuraciones validas del manipulador. Entonces para un mismo $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0})_i$ pueden existir diferentes $\mathbf{t_{col}}$ y $\boldsymbol{\tau}$. Recordemos que el objetivo es abarcar el mayor espacio de trabajo posible; por tanto, resulta fundamental generar la mayor cantidad de configuraciones $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0}, \boldsymbol{\tau})$ posibles, dado el conjunto de configuraciones válidas $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0})$. Por esta razón, durante el entrenamiento, para cada *batch* de condiciones iniciales $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0})$ se sortean nuevamente los valores asociados de $\boldsymbol{\tau}$ y $\mathbf{t_{col}}$.

Es crucial que la red neuronal aprenda a responder correctamente frente a un amplio conjunto de torques posibles, ya que de ello dependerá la flexibilidad y robustez del modelo al integrarlo posteriormente en un esquema de control basado en MPC. A su vez, re-muestrear los puntos de colocación permite asegurar una adecuada cobertura del espacio temporal, siendo el muestreo LHS un complemento importante en esta estrategia. Los puntos de colocación pueden sortearse incluso en cada pasada por la red, aumentando la diversidad de puntos de colocación vistos durante el entrenamiento.

5.6.1. Batcheo de datos

El entrenamiento se realiza por *batches* de condiciones iniciales (CIs) y se evalúa la PINN en múltiples puntos de colocación por cada CI. Notación:

$$B = \text{tamaño de batch}, \qquad N = \text{número de collocation points}, \qquad n = \text{DOF}.$$

5.6.2. Muestreo por batch.

En cada época se llama a `utils.generate_batch_from_pool` para construir

$$q_0 \in \mathbb{R}^{B\times n}, \quad \dot{q}_0 \in \mathbb{R}^{B\times n}, \quad \tau \in \mathbb{R}^{B\times n}, \quad t_{\text{col}} \in \mathbb{R}^{N}.$$

Las CIs $(q_0, \dot{q}_0)$ provienen del *pool* de configuraciones válidas; en cambio, τ y t_{col} se re-muestrean en cada época para cubrir distintos esfuerzos y ubicaciones temporales como ya fue mencionado.

5.6.3. Validación y Test

Del conjunto total de configuraciones válidas $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0})$ se reserva un 20 % para el conjunto de validación. Para estas configuraciones se sortean torques $\boldsymbol{\tau}_{\text{val}}$ y puntos $\mathbf{t}_{\text{col,val}}$ que se mantienen fijos durante todo el entrenamiento.

5.6.4. Independencia y generalidad.

Aún cuando algunas combinaciones de torques $\boldsymbol{\tau}$ aparezcan en los conjuntos de entrenamiento y validación, se considera que no se pierde generalidad, ya que cada batch genera

nuevas combinaciones de estados y tiempos $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0}, \boldsymbol{\tau}, \mathbf{t}_{\text{col}})$. El modelo no memoriza torques individuales, sino que aprende a resolver la dinámica del sistema completa, es decir, la relación funcional:

$$(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) \longmapsto \boldsymbol{\tau} = M(\mathbf{q})\,\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\,\dot{\mathbf{q}} + G(\mathbf{q}).$$

La información relevante para el aprendizaje proviene principalmente de la diversidad de configuraciones articulares $(\mathbf{q}, \dot{\mathbf{q}})$, que determinan las matrices M, C y G, mientras que los torques $\boldsymbol{\tau}$ intervienen como término de excitación que condiciona las aceleraciones resultantes $\ddot{\mathbf{q}}$.

Al reservar un subconjunto de configuraciones iniciales, la red nunca ve todos los estados posibles del manipulador durante el entrenamiento. Esto fuerza a la PINN a generalizar la dinámica aprendida hacia regiones del espacio articular que no fueron observadas directamente.

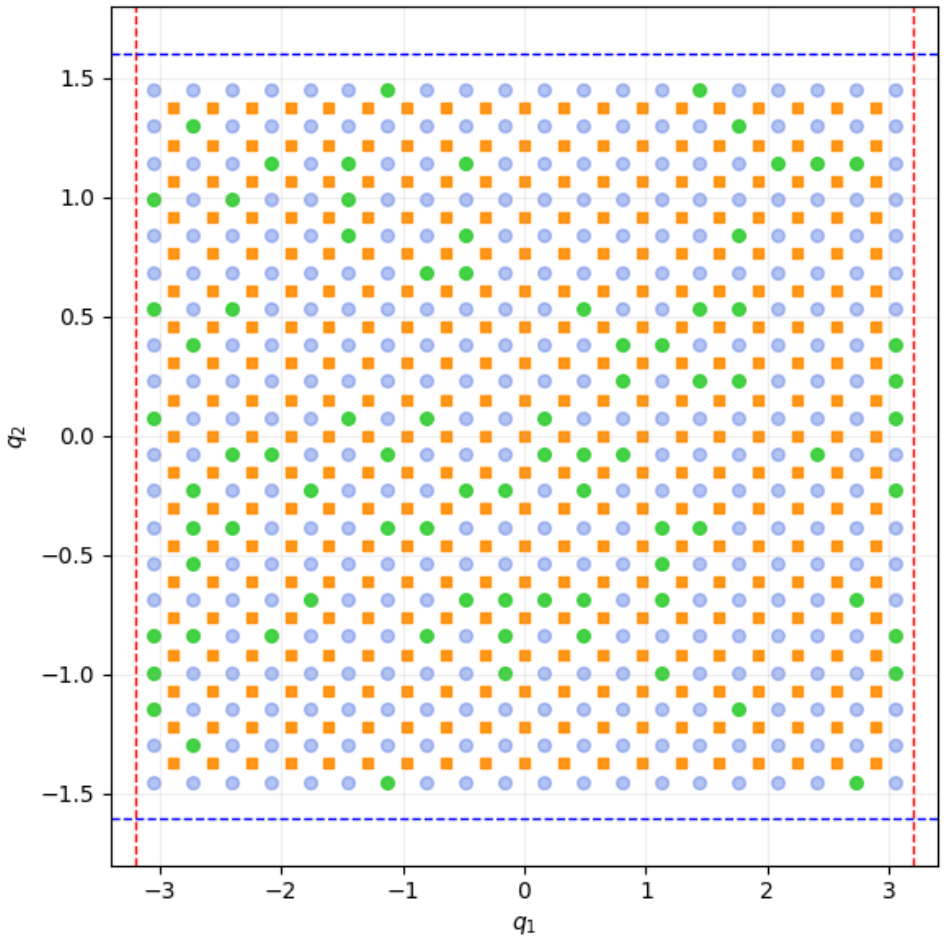


Figura 5.14: Selección de configuraciones del manipulador para training, test y validación. En azul, el conjunto de entrenamiento, en verde el de validación y en naranja el de test. Se tomó una grilla con un resolución baja para visibilizar los conjuntos correctamente

Para el conjunto de *test*, se genera una grilla nueva de configuraciones articulares $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0})$. Esta grilla es idéntica a la anterior, pero dándole un desfase para que los puntos no coincidan. De esta manera se logra evaluar la red en puntos en los que nunca se entrenó ni validó. En la Figura 5.14 se muestra cómo se tomaron los diferentes conjuntos.

5.7. Experimentos y pruebas

Las primeras pruebas al modelo se realizaron con el motivo de confirmar el funcionamiento básico del mismo y la correcta implementación de las funciones de pérdida. Para esto se fijaron los pesos de manera que en cada prueba quede solo una función de costo activa, con el fin de ver el ajuste a través de descenso por gradiente. A lo largo de estos experimentos se variaron algunos parámetros, dejando fijo únicamente el conjunto de configuraciones válidas de validación y training.

5.7.1. Validación de las pérdidas

Los hiperparámetros utilizados fueron:

$n_{\text{col}} = 2000$ (cantidad de puntos de colocación)
epochs $= 3000$ (épocas de entrenamiento)
batch_size $= 50$ (50 condiciones iniciales por batch)
optimizer $=$ Adam

Recordar que la red, en cada época evaluará las funciones de pérdida para cada *condición inicial* en cada punto de colocación, y que cada batch es diferente en $\boldsymbol{\tau}$ y $\mathbf{t_{col}}$. Por lo tanto la red estaría evaluando (en el caso de las pérdidas ODE y Kin) alrededor de $(n_{col} \times batch\_size) \times epochs$ (300 millones) puntos. Se utilizaron 2000 puntos de colocación con el objetivo de asegurar una cobertura temporal suficiente del dominio de entrenamiento y una buena aproximación de las derivadas temporales mediante autograd. Este valor se determinó empíricamente como un compromiso entre la precisión en la evaluación del residuo dinámico y el costo computacional del entrenamiento.

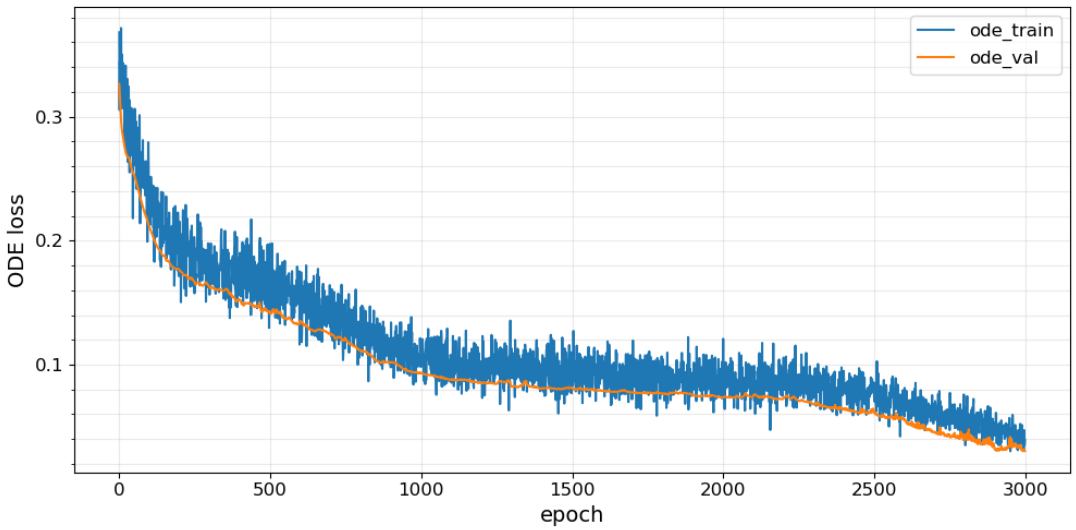


Figura 5.15: $\mathcal{L}_{ODE}$

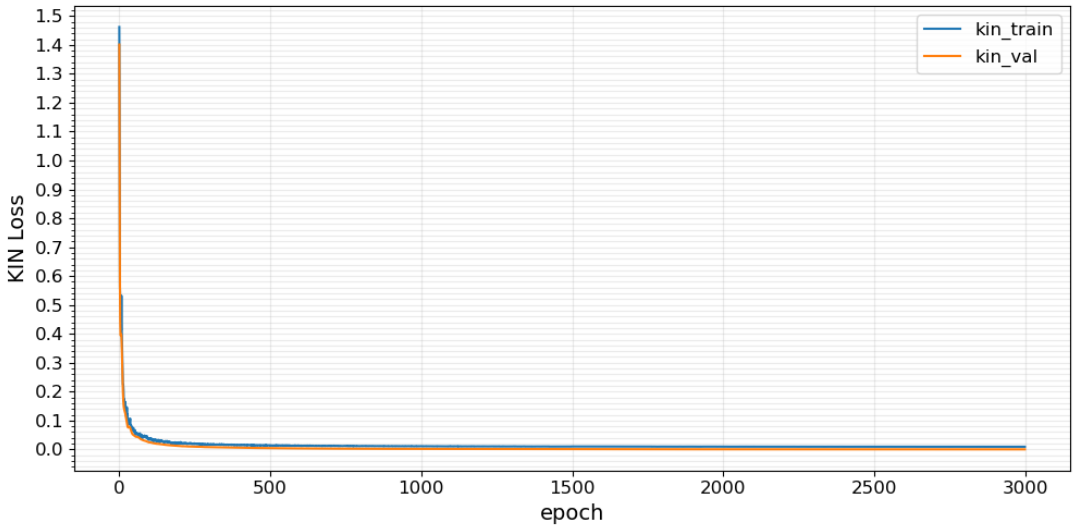


Figura 5.16: $\mathcal{L}_{kin}$

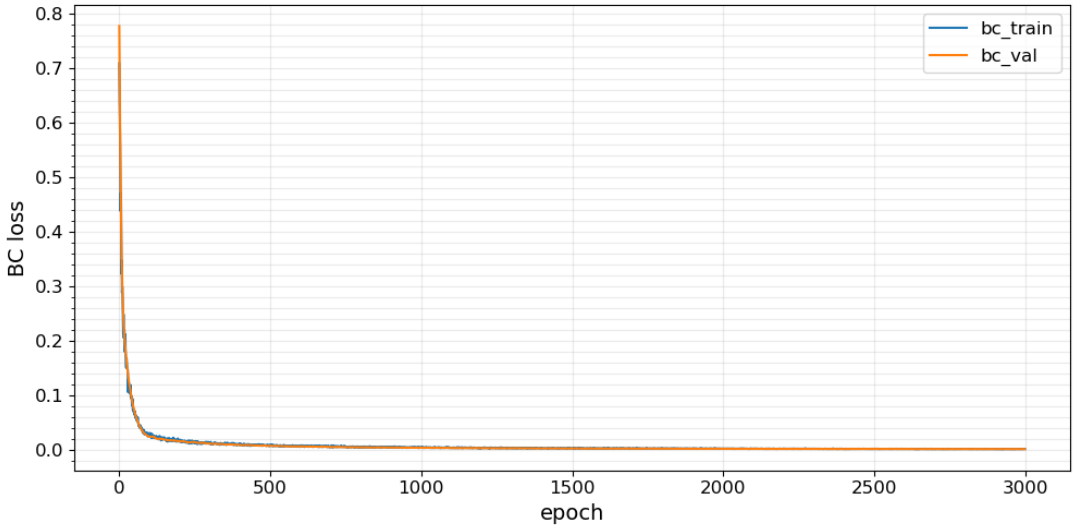


Figura 5.17: $\mathcal{L}_{BC}$

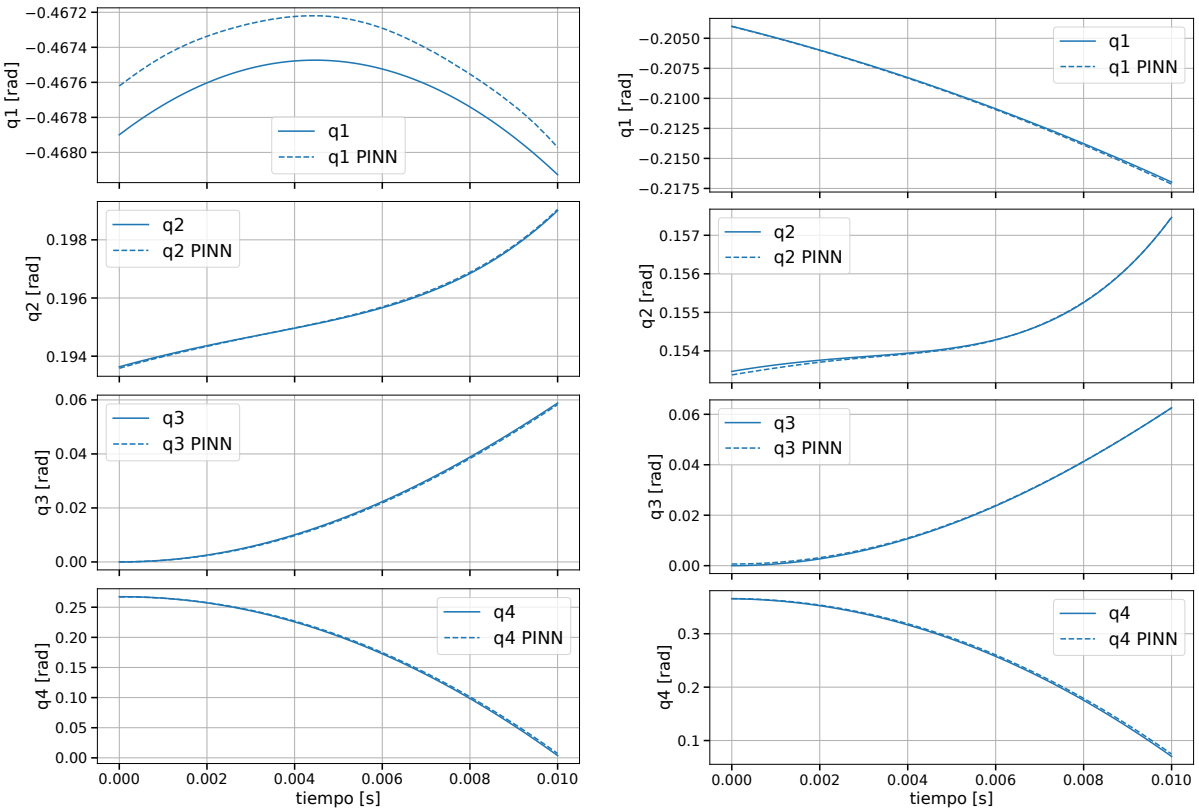
Figura 5.18: Funciones de costo tras 3000 épocas de optimización. Para lograr *apagar* una de las componentes del costo compuesto se elijen, por ejemplo $w_{BC} = w_{kin} = 0$ y se aisla la función que se quiere probar. Todas las funciones son optimizadas de manera satisfactoria.

Como se puede ver en la Figura 5.18, Adam logra satisfactoriamente minimizar estas funciones de costo individualmente. Esta prueba sirve para ver de dónde proviene la complejidad en la optimización, que en este caso evidencia que la función de costo compuesta podría ser el obstáculo (individualmente, se minimizan todas las funciones de costo. Se verá luego que esto no es tan sencillo cuando se componen las tres).

5.7.2. Validez física de los resultados

Si bien se consigue minimizar los residuos físicos, existe la posibilidad de que por algún error en la implementación, estas funciones no tengan relación con la física del sistema y que se esten minimizando residuos que pueden ser inconsistentes con la dinámica, llegando a soluciones triviales del sistema o trayectorias degeneradas. Para chequear esto se toma el caso del manipulador de cuatro grados de libertad **sin bloquear joints** para ver si la

minimización del residuo físico lleva a la correcta predicción de las trayectorias a partir del estado $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0}, \boldsymbol{\tau})$. Para que esta prueba fuera posible, se entrenó la red neuronal para pocas *condiciones iniciales* y la se la evaluó en los mismos puntos de entrenamiento, con el objetivo de ver si la trayectoria predicha por la PINN es consistente con el avance real del sistema. Esto puede resultar confuso, pero es crucial recordar que la PINN **no "ve" las trayectorias enteras**, solo considera $(\mathbf{q_0}, \dot{\mathbf{q}}_\mathbf{0}, \boldsymbol{\tau})$ y minimiza las funciones de pérdida (**no se toma en cuenta $\mathcal{L}_{DATA}$, por que lo explicado, si este fuera el caso la siguiente prueba no tendría sentido**).



(a) c_0 (b) c_1

Figura 5.19: Evaluación del desempeño del modelo tras minimizar el residuo físico. Es importante entender que la red no "vió" las trayectorias completas sino que solo ajustó las funciones de pérdida $\mathcal{L}_{BC}$ y $\mathcal{L}_{ODE}$. Con estos resultados se comprueba que la red al minimizar estas funciones efectivamente aprende la dinámica del sistema, sin caer en soluciones triviales o degeneradas

Los hiperparametros considerados en este caso fueron:

$$n_{\text{col}} = 200 \quad \text{(cantidad de puntos de colocación)}$$
$$epochs = 15k \quad \text{Épocas de entrenamiento}$$
$$batch\_size = 10 \quad \text{(10 } \textit{condiciones iniciales} \text{ por batch)}$$
$$opt = Adam$$

También en este caso se redujo el horizonte de predicción T, para que la predicción sea más sencilla. Se puede ver en la Figura 5.19, los resultados de evaluar la predicción de la red para dos condiciones iniciales (c_i) de las diez para las que fue entrenada, cabe destacar que los $\boldsymbol{\tau}_{val}$ son sorteados y podrían ser diferentes a los de entrenamiento. En 5.19 se puede ver que la red, con solo minimizar el residuo físico y la pérdida asociada a las

condiciones iniciales, logra predecir con claridad las trayectorias. Estas pruebas también demostraron la dificultad de entrenar un modelo para 4 grados de libertad, ya que incluso para puntos vistos durante el entrenamiento el ajuste no es perfecto, lo que dio indicios de que el modelo puede ser muy complejo cuando se trabaja con todos los grados de libertad disponibles.

5.8. Entrenamiento

Una vez verificada la capacidad del modelo para predecir trayectorias completas con el comportamiento esperado, el siguiente paso consiste en refinar el entrenamiento con el objetivo de obtener una red robusta y precisa, apta para ser integrada dentro de un esquema de control MPC.

5.8.1. Uso de ClusterUy

El entrenamiento y evaluación de los modelos se realizaron utilizando recursos de cómputo del clúster nacional Cluster-UY [19]. Esto permitió acelerar considerablemente el entrenamiento y sistematizar las pruebas realizadas y los modelos entrenados.

5.8.2. Modelos entrenados

Durante el desarrollo del proyecto se entrenaron muchos modelos de la PINN con distintas configuraciones de hiperparámetros. No todos los experimentos fueron documentados, ya que muchos no presentaron mejoras cuantitativas ni comportamientos cualitativos relevantes respecto a versiones previas. Se registraron y analizaron múltiples modelos que mostraron resultados estables o con alguna mejora significativa, los cuales se encuentran documentados en una planilla disponible en el repositorio del proyecto [1]. Los principales hiperparámetros que se variaron durante los experimentos fueron:

- **Número de puntos de colocación** (N_{col}): entre 100 y 4000, afectando la resolución temporal con la que se evalúa la dinámica.
- **Tamaño del batch de condiciones iniciales** (CI_{batch}): entre 8 y 64, determinando cuántas configuraciones simultáneas se utilizan por iteración.
- **Arquitectura de la red**: cantidad de capas ocultas y neuronas por capa.
- **Pesos de pérdida**: $w_{\text{BC}}, w_{\text{kin}}, w_{\text{dyn}}$
- **Optimizador**: Adam variando lr, β_1, β_2.
- **Tamaño de grilla de muestreo**: N_{grid}
- **Épocas**: Cantidad de epocas

Los modelos que lograron mejores resultados fueron aquellos que presentaron una mayor minimización de los residuos físico y cinemático, es decir, los términos $\mathcal{L}_{\text{ODE}}$ y $\mathcal{L}_{\text{kin}}$ de la función de costo. Estos modelos mostraron una buena coherencia entre la derivada temporal de la posición y la velocidad estimada, y un cumplimiento consistente de la ecuación de movimiento del manipulador.

[1] https://gitlab.fing.edu.uy/rodrigo.baliosian/pfc-control-mpc-basado-en-pinns

5.8.3. Métricas

Para evaluar el desempeño de los modelos se utilizó el **error cuadrático medio (RMSE)** entre las trayectorias predichas por la PINN y las trayectorias simuladas mediante el algoritmo ABA, tanto para las posiciones articulares $\mathbf{q}(t)$ como para las velocidades $\dot{\mathbf{q}}(t)$.

$$\mathrm{RMSE}_q = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \|\hat{\mathbf{q}}(t_i) - \mathbf{q}_{\mathrm{sim}}(t_i)\|^2}, \qquad \mathrm{RMSE}_{\dot{q}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \|\hat{\dot{q}}(t_i) - \dot{\mathbf{q}}_{\mathrm{sim}}(t_i)\|^2} \tag{5.40}$$

Estas métricas cuantifican la distancia promedio entre las trayectorias predichas por la red y las trayectorias obtenidas a partir de la dinámica directa del manipulador. Las posiciones $\mathbf{q}$ se expresan en radianes y las velocidades $\dot{\mathbf{q}}$ en rad/s, por lo que sus valores absolutos pueden interpretarse directamente como errores angulares o cinemáticos.

En términos físicos, un RMSE_q de 0,1 rad equivale aproximadamente a un error angular de 6°, lo que representa un seguimiento razonablemente preciso de la dinámica, mientras que un $\mathrm{RMSE}_{\dot{q}}$ de 0,2 rad/s (aprox 11,5°/s) indica un buen seguimiento de la velocidad articular.

Cabe destacar que en el *modo autorregresivo* (AR), el modelo tiende a acumular errores debido a que la salida de un paso es la entrada del anterior. Por esto es esperable que el error RMSE aumente respecto al modo directo.

Modelo	**RMSE_q (directo)**	**$\mathrm{RMSE}_{\dot{q}}$ (directo)**	**RMSE_q (AR)**	**$\mathrm{RMSE}_{\dot{q}}$ (AR)**
`job_4898343`	0.068	0.574	0.081	1.157
`job_4882301`	0.080	0.891	0.089	1.209
`job_4884309`	0.082	0.985	0.092	1.212

Tabla 5.3: Resultados de los mejores modelos evaluados en el conjunto de test. Se muestran los errores promedio de posición (RMSE_q) y velocidad ($\mathrm{RMSE}_{\dot{q}}$) tanto en evaluación directa como en modo autoregresivo.

Como se observa en la Tabla 5.3, los errores en posición articular (RMSE_q) se mantienen en valores bajos, indicando que la red logra capturar correctamente la evolución de las configuraciones del manipulador. Sin embargo, el error en velocidad ($\mathrm{RMSE}_{\dot{q}}$) resulta considerablemente mayor, especialmente en el modo autoregresivo, donde las pequeñas desviaciones se acumulan a lo largo de los pasos de predicción. Esto sugiere que, si bien la PINN reproduce con buena precisión las trayectorias en el espacio de configuración, tiene mayores dificultades para mantener la coherencia en la dinámica de velocidades.

En el contexto de control predictivo (MPC), pequeños errores en las velocidades articulares no resultan críticos. El controlador utiliza la PINN principalmente para predecir la evolución de las posiciones y generar las acciones de control $\boldsymbol{\tau}$. Mientras el error en posición $\mathbf{q}$ se mantenga acotado, el MPC puede corregir desviaciones en la velocidad $\dot{\mathbf{q}}$ mediante la realimentación. En consecuencia, un ligero error en la predicción de $\dot{\mathbf{q}}$ no compromete la estabilidad ni el desempeño general del esquema de control.

5.8.4. Funcionamiento del modelo

Luego de haber seleccionado el modelo, los resultados RMSE dan una idea de que está funcionando correctamente y que puede predecir la dinámica del manipulador. En esta parte se harán algunas demostraciones más concretas para apreciar cámo funciona el modelo. Primero, para analizar visualmente la calidad de las trayectorias, se simularon tres variantes del movimiento del brazo: la referencia obtenida mediante el algoritmo ABA, la predicción directa de la PINN y la predicción autoregresiva (PINN-AR).

A partir de las trayectorias articulares $\mathbf{q}(t)$ obtenidas en cada caso, se aplicó cinemática directa para calcular la posición del punto operativo en el espacio cartesiano.

Estas posiciones se proyectaron sobre el plano (x, y), permitiendo comparar la forma general del movimiento y el seguimiento espacial entre la simulación física (ABA) y las trayectorias aprendidas por la PINN. En la Figura 5.20 se pueden ver los resultados para una entrada $(\mathbf{q}_{0_{test}}, \dot{\mathbf{q}}_{0_{test}}, \boldsymbol{\tau}_{0_{test}})$

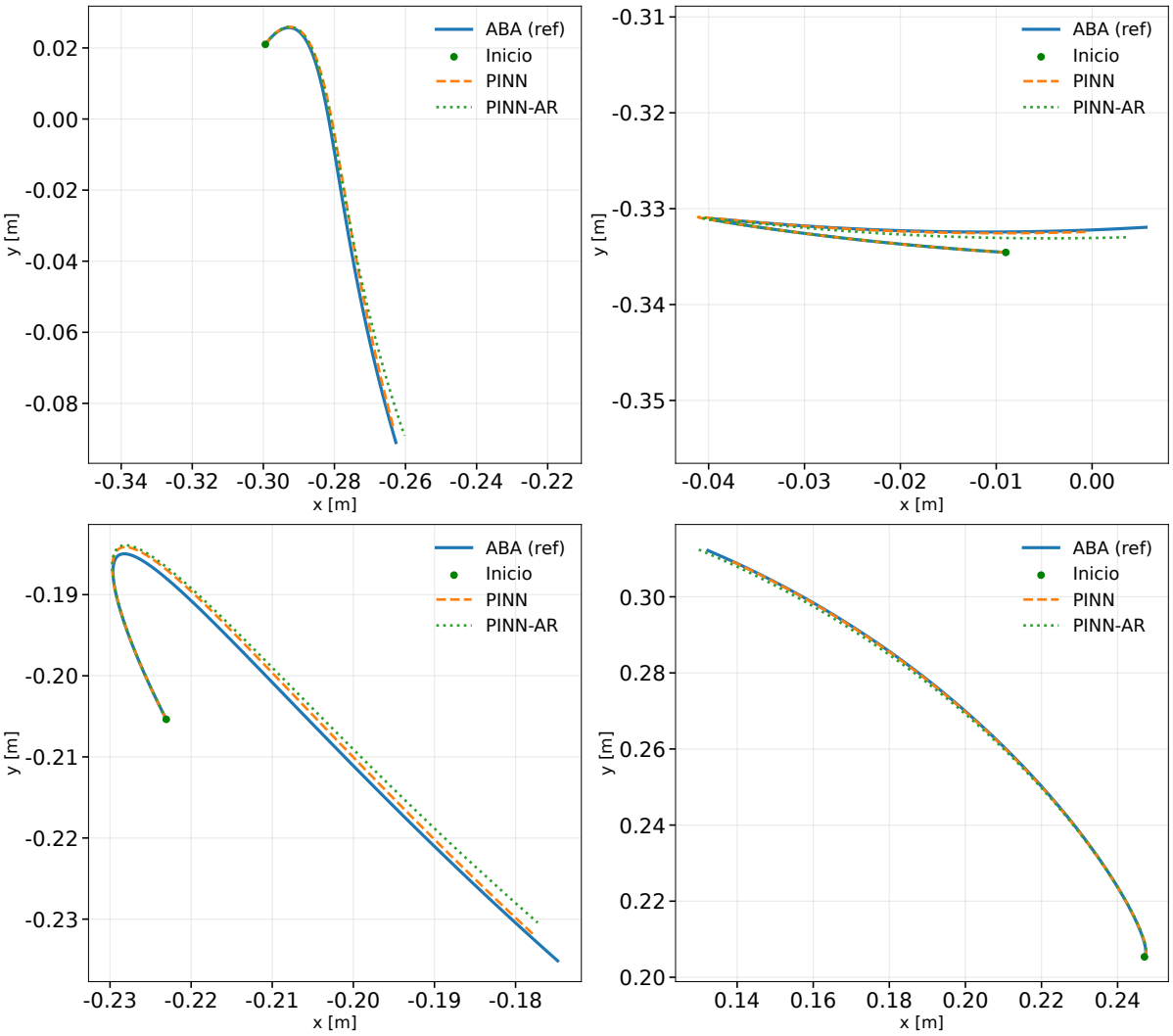


Figura 5.20: Trayectorias en coordenadas cartesianas del punto operativo del manipulador, proyectadas sobre el plano x,y. Se presenta solo un punto de entrada a la red. Se seleccionaron 4 casos al azar, como se puede ver, el comportamiento es variado.

En el repositorio se puede encontrar un archivo mp4 (OMX_PINN_SIM.mp4) en el que se muestran, para el conjunto de test, los movimientos del OMX y la predicción de la PINN

(en azul). Finalmente, se presentan algunos resultados de funcionamiento, comparando trayectorias articulares predichas contra las simuladas mediante ABA 5.21.

5.8.5. Conclusiones sobre el entrenamiento

Se logró obtener un modelo que cumple satisfactoriamente con la predicción de la trayectoria, lo que ya permite avanzar a una implementación punta a punta del sistema de control MPC. Los errores en velocidad, pueden tener que ver con que el término ∇_{kin} fuerza la coherencia interna pero no necesariamente con la simulación a partir de ABA. Recordar que al tener el manipulador puramente simulado, ésta es la única referencia disponible.

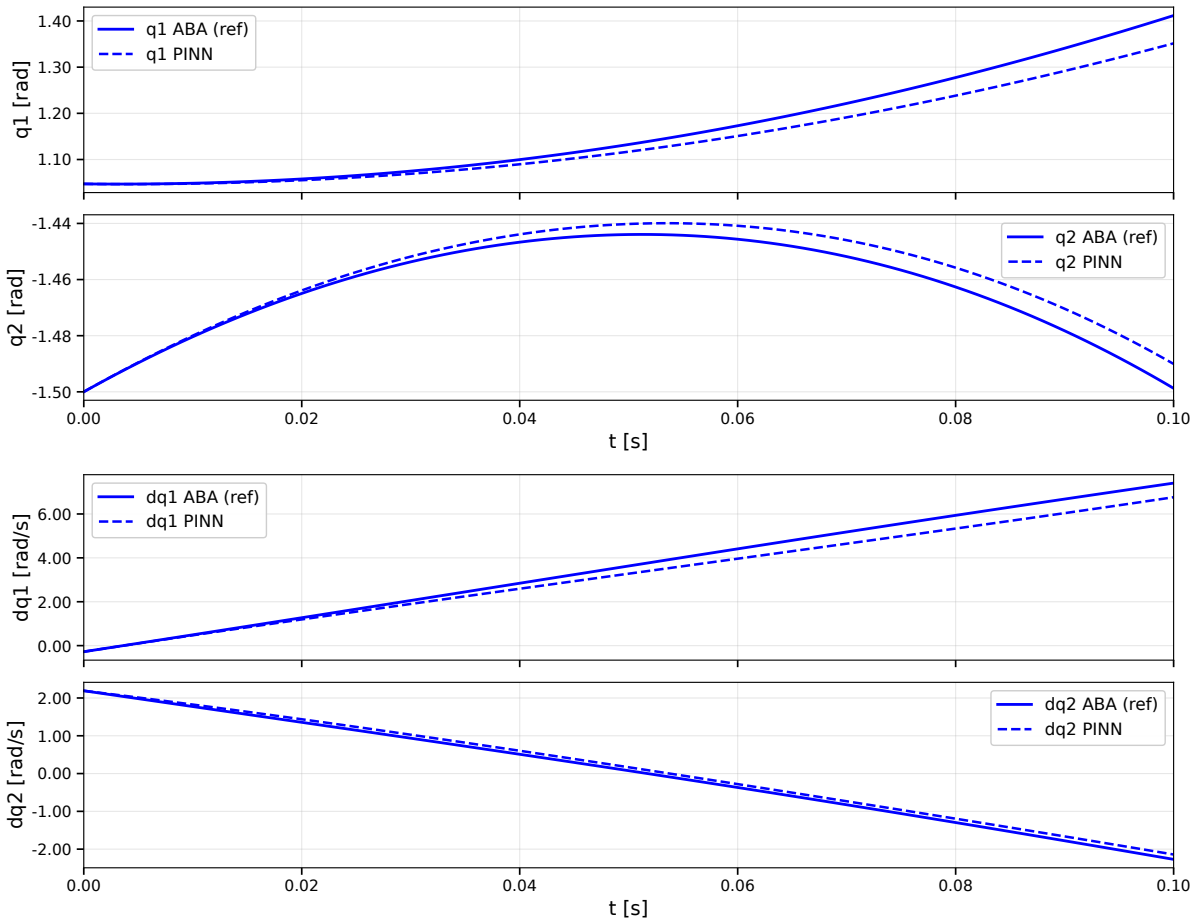


Figura 5.21: Para una configuración al azar del conjunto de testing, se observa la predicción de la red en las trayectorias y velocidades articulares.

5.8.6. Modelo final

Los parámetros de entrenamiento usados en el modelo seleccionado fueron:

Hiperparámetro	**Valor**
$epochs$	100000
ci_batch	10
w_{ode}	1,0
w_{kin}	0,75
T	0,1
N_{col}	2000
n_{hidden}	4
$n_{neurons}$	128

Tabla 5.4: Parámetros de entrenamiento usados para el modelo con mejor desempeño. n_{hidden} corresponde a la cantidad de capas ocultas y $n_{neurons}$ a la cantidad de neuronas por capa oculta. Notar que el número de *condiciones iniciales* vistas es 1e

5.8.7. Diagnóstico de fallas

Durante el entrenamiento se monitorearon los gradientes de cada término de la función de costo, con el objetivo de detectar posibles desbalances o conflictos entre pérdidas. Observar los gradientes es útil porque permite entender qué componentes están realmente guiando la actualización de los parámetros: si una pérdida tiene gradientes mucho más grandes que las otras, dominará el aprendizaje, y los demás términos tendrán poca influencia.

El análisis de gradientes ayudó a verificar que los términos asociados a la dinámica y la consistencia cinemática tuvieran una contribución equilibrada. Esto permitió ajustar los pesos relativos de cada pérdida y detectar cuándo la red aprendía trayectorias de posición correctas pero con velocidades sesgadas.

5.8.8. Coste computacional y escalabilidad

El entrenamiento del modelo requirió un uso intensivo de recursos de cómputo debido a la naturaleza del problema y al tamaño de los *batches* a ser procesados.

En términos de escalabilidad, la complejidad del modelo aumenta rápidamente con los grados de libertad (DOF) del manipulador. Cada articulación adicional introduce nuevas variables de estado $(q_i, \dot{q}_i, \tau_i)$, incrementando el tamaño de entrada y salida de la red, así como la dimensión de las matrices dinámicas M, C y G. Esta *explosión dimensional* se traduce en un tiempo de entrenamiento más prolongado.

El uso de tensores en PyTorch es paralelo por naturaleza, pero al integrarlo con **Pinocchio** se pierde esa ventaja porque parte del código solo se ejecuta en CPU. Este cuello de botella puede ser sorteable mediante paralelización en CPU de los algoritmos usados.

5.9. Implementación del sistema completo

Esta sección presenta la implementación completa del sistema de control predictivo (MPC) aplicado al brazo robótico *OpenManipulator-X*.

El objetivo principal de esta sección es validar el funcionamiento del controlador en un entorno simulado, evaluando su desempeño en términos de seguimiento, estabilidad y eficiencia computacional.

5.9.1. Integración general del sistema

El sistema implementado integra todos los módulos desarrollados en capítulos previos, combinando el modelo dinámico del brazo, el controlador predictivo y la simulación dentro del entorno ROS 2.

La comunicación entre los nodos `trajectory_node`, `mpc_node` y `manager_node`, junto con su interacción con el simulador Gazebo, sigue la estructura previamente presentada en la Figura 4.7 del Capítulo 4.

Para la implementación del sistema completo, se utilizaron dos predicciones distintas del modelo dinámico:

- Mediante la librería *Pinocchio*.
- Mediante la red neuronal informada por la física (PINN) entrenada, comentada en la sección 5.8.

Ambas configuraciones se evalúan en el mismo entorno de simulación para comparar su desempeño en términos de precisión de seguimiento y consistencia dinámica.

5.9.2. Caso 1: Predicción de la dinámica mediante Pinocchio

Predicción del modelo dinámico con Pinocchio

Con el fin de disponer de una referencia física verificable para el controlador predictivo, se implementó un modelo dinámico explícito del robot empleando la librería `Pinocchio`. Esta herramienta permite calcular de forma eficiente las ecuaciones del movimiento a partir del modelo multicuerpo descrito en el archivo URDF, incluyendo el cómputo exacto de las aceleraciones articulares, fuerzas generalizadas y sus derivadas parciales.

A diferencia del caso basado en redes neuronales, donde la diferenciación se realiza de forma automática dentro de `TensorFlow`, la integración con `Pinocchio` requiere un tratamiento distinto: esta librería está escrita en `C++` y vinculada a Python mediante `NumPy`, por lo que no participa del grafo de cómputo de `TensorFlow`. En consecuencia, los gradientes no pueden obtenerse mediante *backpropagation*, y deben calcularse de manera explícita a partir de las derivadas analíticas de la dinámica. Esto garantiza compatibilidad con el optimizador `RMSprop` empleado en el MPC por el nodo `mpc_node` desarrollado en la Subsección 4.2.3.

La dinámica del robot se calcula utilizando el algoritmo de cuerpo articulado explicado en la Sección 5.1.12 (*Articulated-Body Algorithm*, ABA):

$$\ddot{q} = \mathrm{ABA}(q, \dot{q}, \tau), \tag{5.41}$$

donde $q \in \mathbb{R}^n$ y $\tau \in \mathbb{R}^n$ representan las posiciones articulares y los torques aplicados, respectivamente. A partir de esta aceleración, se integran las ecuaciones de movimiento de manera explícita, respetando el periodo de muestreo dt:

$$\begin{aligned} \dot{q}_{k+1} &= \dot{q}_k + dt\,\ddot{q}_k, \\ q_{k+1} &= \text{integrate}\big(q_k,\, dt\,\dot{q}_{k+1}\big), \end{aligned} \tag{5.42}$$

normalizando q tras cada paso para conservar la consistencia geométrica en el espacio de configuración. Cada intervalo dt puede subdividirse en M subpasos para mejorar la estabilidad numérica, aplicando saturaciones suaves en torques y velocidades antes de almacenar el estado del sistema:

$$X_k = \begin{bmatrix} q_k^\top & \dot{q}_k^\top \end{bmatrix}^\top \in \mathbb{R}^{2n}.$$

Cálculo del gradiente analítico: El vector de decisión del MPC corresponde a la secuencia de torques $U = \{u_k, \ldots, u_{k+N_2-1}\}$, y el objetivo es minimizar la función de costo cuadrática:

$$J(U) = \sum_{j=1}^{N_2} \|X_{k+j} - X_{k+j}^{\text{ref}}\|_Q^2 + \sum_{i=0}^{N_u-1} \|\Delta u_{k+i}\|_R^2, \tag{5.43}$$

donde $\Delta u_k = u_k - u_{k-1}$. El gradiente de J con respecto a los torques no puede obtenerse mediante el grafo de `TensorFlow`, por lo que se calcula explícitamente a partir de las derivadas parciales de la dinámica. Estas se obtienen mediante la función `computeABADerivatives(model, data, q, dq, tau)`, que proporciona las jacobianas locales de la aceleración:

$$\frac{\partial \ddot{q}}{\partial q}, \qquad \frac{\partial \ddot{q}}{\partial \dot{q}}, \qquad \frac{\partial \ddot{q}}{\partial \tau}.$$

Con estas derivadas se construye una linealización local del modelo dinámico:

$$x_{k+1} = A_k\, x_k + B_k\, u_k, \tag{5.44}$$

donde

$$A_k = \begin{bmatrix} I + dt\,\frac{\partial \dot{q}_{k+1}}{\partial q_k} & dt\,\frac{\partial \dot{q}_{k+1}}{\partial \dot{q}_k} \\ dt\,\frac{\partial \ddot{q}_k}{\partial q_k} & I + dt\,\frac{\partial \ddot{q}_k}{\partial \dot{q}_k} \end{bmatrix}, \qquad B_k = \begin{bmatrix} dt\,\frac{\partial \ddot{q}_k}{\partial \tau_k} \\ \frac{\partial \ddot{q}_k}{\partial \tau_k}\,dt \end{bmatrix}.$$

A partir de esta linealización, se implementa un esquema de retropropagación tipo adjunto:

$$\lambda_{N_2} = \mathbf{0}, \qquad \lambda_k = A_k^\top \lambda_{k+1} + 2Q(x_{k+1} - x_{k+1}^{\text{ref}}), \tag{5.45}$$

y el gradiente del costo respecto a cada torque resulta:

$$\frac{\partial J}{\partial u_k} = B_k^\top \lambda_{k+1} + 2R\Delta u_k - 2R\Delta u_{k+1}, \qquad \text{si } k+1 < N_2. \tag{5.46}$$

Optimización: El gradiente $\nabla_U J$ se emplea en un esquema de descenso con el optimizador `RMSprop` de `Keras/TensorFlow`. Tras cada actualización, los torques se proyectan a los límites físicos $[u_{\text{mín}}, u_{\text{máx}}]$ y se aplica la política de *horizonte constante* para $N_u < N_2$:

$$U \leftarrow \text{clip}(U, u_{\text{mín}}, u_{\text{máx}}), \qquad U[j \geq N_u] \leftarrow U[N_u - 1]. \tag{5.47}$$

El procedimiento se repite hasta que se cumple una de las siguientes condiciones:

I la mejora relativa del costo cae por debajo de un umbral,

II se alcanza el límite temporal de 0,1 s por ciclo, o

III se completa el número máximo de iteraciones configurado.

Resumen. En síntesis, el modelo dinámico basado en `Pinocchio` permite predecir la evolución física del robot de forma analítica, utilizando derivadas exactas de la dinámica para calcular los gradientes requeridos por el MPC. Este enfoque preserva la interpretabilidad física del modelo y mantiene tiempos de cómputo competitivos, al evitar la retropropagación numérica completa a través del integrador dinámico.

Parámetros de la implementación

La Tabla 5.5 resume los parámetros utilizados en la configuración experimental del controlador en conjunto con la predicción del modelo dinámico realizado por la librería `Pinocchio`.

Tabla 5.5: Parámetros generales del controlador MPC utilizados en la simulación del brazo robótico con predicción mediante Pinocchio.

Parámetro	Símbolo	Valor
Cantidad de articulaciones	n_{joints}	2
Grados de libertad	$q, \dot{q}$	2 (posiciones) + 2 (velocidades)
Paso de muestreo	dt	0.1 s
Tiempo total de simulación	T	20.0 s
Cantidad total de pasos	$N_{\text{total}} = T/dt$	200
Horizonte de predicción	N_2	5
Horizonte de control	N_u	4
Matriz de penalización de estados	$Q_{2n\times 2n}$	diag(200, 2000, 2, 2)
Matriz de penalización de control	$R_{n\times n}$	diag(10^{-4}, 10^{-4})
Límite inferior de torque	$u_{\text{mín}}$	-3.0 N·m
Límite superior de torque	$u_{\text{máx}}$	+3.0 N·m
Optimizador empleado	–	RMSprop (lr = 10^{-2})

Ejecución del sistema

La ejecución del sistema se llevó a cabo íntegramente en el entorno de simulación de Gazebo, utilizando el modelo dinámico del robot OpenMANIPULATOR-X y la predicción basada en la librería `Pinocchio`.

Durante la prueba, el controlador predictivo debió seguir una trayectoria cartesiana generada por el nodo `trajectory_node`, diseñada para evaluar la capacidad del sistema de

mantener un seguimiento continuo y estable en presencia de movimientos acoplados entre articulaciones.

La trayectoria se definió en el plano XY, manteniendo constante la altura del efector final en el eje Z, de modo que el extremo del brazo describe un arco suave en el espacio de trabajo. Este movimiento corresponde a un barrido angular de la articulación base (q_1) desde 0 hasta 120° durante los 20 s de simulación, mientras que la segunda articulación q_2 varía de 90° a 60°. El resultado es un desplazamiento coordinado que combina rotación en la base y extensión progresiva del brazo, manteniendo constante la orientación del efector final, como se observa en la Figura 5.22.

La trayectoria fue discretizada en 200 puntos equiespaciados en el tiempo, correspondientes a un periodo de muestreo de $dt = 0{,}1\,\text{s}$. Cada punto representa una configuración cartesiana (x_d, y_d, z_d) del efector final, asociada a un instante temporal dentro del horizonte total de simulación. Esta discretización garantiza la coherencia temporal entre las distintas etapas del sistema: la generación de la referencia por el `trajectory_node`, la resolución de la cinemática inversa mediante el servicio `/compute_ik`, y la aplicación del control óptimo calculado por el `mpc_node`.

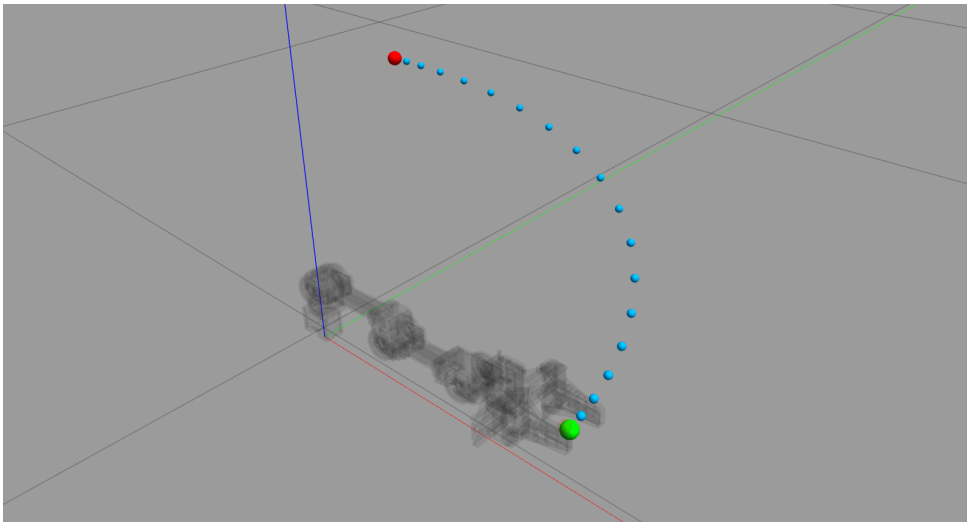


Figura 5.22: Trayectoria cartesiana semicircular definida para la validación del sistema. El efector final se desplaza en el plano XY manteniendo constante la coordenada Z.

Resultados de simulación

En la Figura 5.23 se muestran las posiciones articulares $q_1(t)$ y $q_2(t)$ junto con los torques aplicados por el controlador MPC. Las líneas de color **naranja** representan las referencias generadas por el nodo `trajectory_node`, mientras que las líneas **azules** indican los valores obtenidos en la simulación.

Se observa que ambas articulaciones siguen sus trayectorias con alta precisión general, presentando una respuesta estable y sin oscilaciones significativas. Sin embargo, durante los primeros 4 s de simulación, la articulación q_2 permanece prácticamente inmóvil. Este comportamiento se explica porque el torque inicial parte desde $u = 0{,}0$ N·m, y el controlador incrementa gradualmente la acción de control en sentido negativo para vencer el peso del propio brazo. De este modo, se evita un sobreesfuerzo inicial, manteniendo bajo el valor de la función de costo J mientras la articulación acumula el par suficiente para superar la gravedad y comenzar el movimiento ascendente.

Los torques calculados (u_1, u_2), representados en los ejes secundarios en color **verde**, muestran una evolución escalonada característica del horizonte de control discreto. Los valores permanecen dentro de los límites definidos ($[-3, 3]\,\mathrm{N{\cdot}m}$) sin evidenciar saturaciones, lo que refleja una correcta ponderación del término $\Delta u^T R \Delta u$ en la función de costo y un equilibrio entre suavidad y precisión de control.

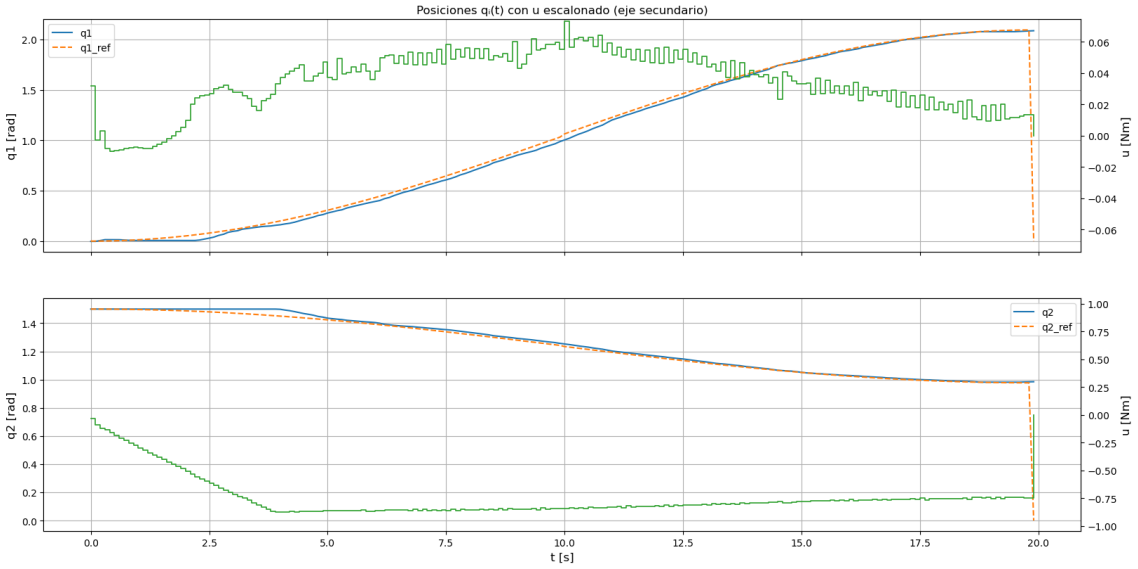


Figura 5.23: Evolución temporal de las posiciones articulares $q_1(t)$, $q_2(t)$ y torques de control u_1, u_2 durante la simulación del MPC con modelo dinámico de `Pinocchio`. Las líneas naranjas representan las referencias deseadas y las líneas azules los valores obtenidos en la simulación. En verde se muestra el torque de control aplicado en cada paso.

En la Figura 5.24 se presenta la evolución temporal de las velocidades articulares $\dot{q}_1(t)$ y $\dot{q}_2(t)$. Ambas articulaciones reproducen la tendencia de las trayectorias de referencia, aunque con menor precisión y presencia de pequeñas oscilaciones y ruido numérico. Esto puede atribuirse, en primer lugar, al efecto de discretización temporal, pero también a la ponderación empleada en la función de costo: la matriz de pesos utilizada,

$$Q = \mathrm{diag}(200,\ 200,\ 2,\ 2),$$

penaliza fuertemente los errores en las posiciones q_1, q_2, mientras que las velocidades $\dot{q}_1, \dot{q}_2$ tienen una influencia mucho menor sobre el valor total del costo. Como resultado, el controlador prioriza la precisión de las posiciones sobre la suavidad del perfil de velocidad, lo que explica las fluctuaciones observadas en los gráficos.

Por otro lado, la Figura 5.25 muestra la evolución de la función de costo J y los tiempos de resolución del controlador MPC durante toda la simulación. En la parte superior se observa cómo el valor de J decrece progresivamente a lo largo de los pasos de simulación. Durante los primeros segundos, el costo inicial de orden 10^4, desciende abruptamente, lo que indica la fase de ajuste del controlador en la que las acciones de control comienzan a corregir la diferencia entre el estado inicial y la trayectoria de referencia. Una vez alcanzado el régimen estacionario (alrededor de $t \approx 5\,\mathrm{s}$), el costo se estabiliza próximo a cero, reflejando un seguimiento preciso y una acción de control eficiente. Este comportamiento confirma la correcta sintonización de las matrices de penalización Q y R, donde los pesos altos sobre las posiciones permiten reducir el error de seguimiento sin introducir oscilaciones ni saturaciones en los torques.

En la parte inferior de la Figura se presenta el tiempo de cómputo del controlador MPC por iteración.

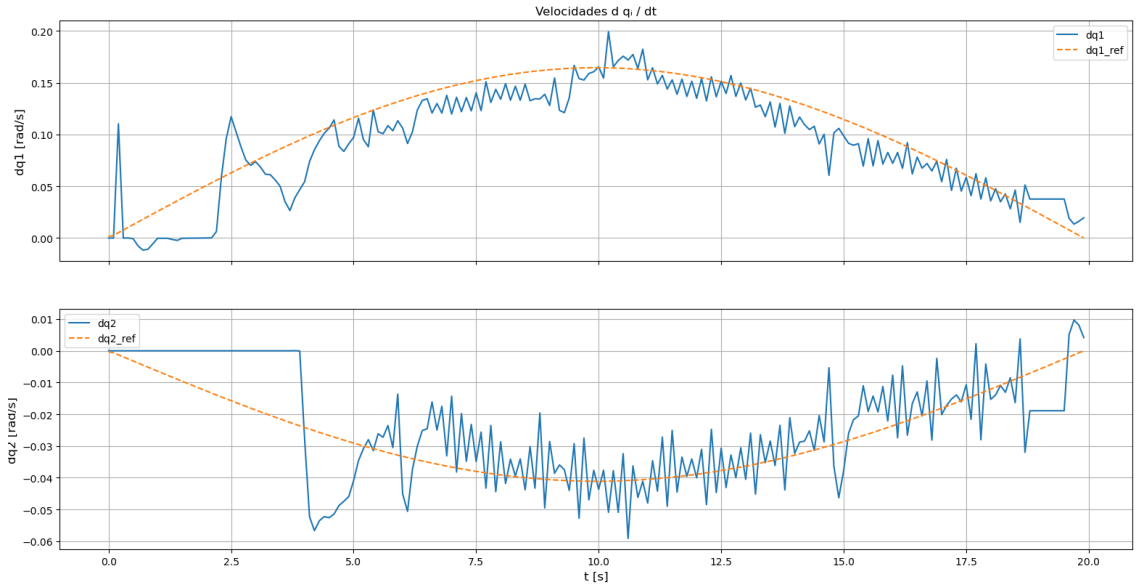


Figura 5.24: Evolución temporal de las velocidades articulares $\dot{q}_1(t)$ y $\dot{q}_2(t)$ durante la simulación del MPC con modelo dinámico de `Pinocchio`. Las líneas naranjas corresponden a las referencias deseadas y las azules a los valores simulados.

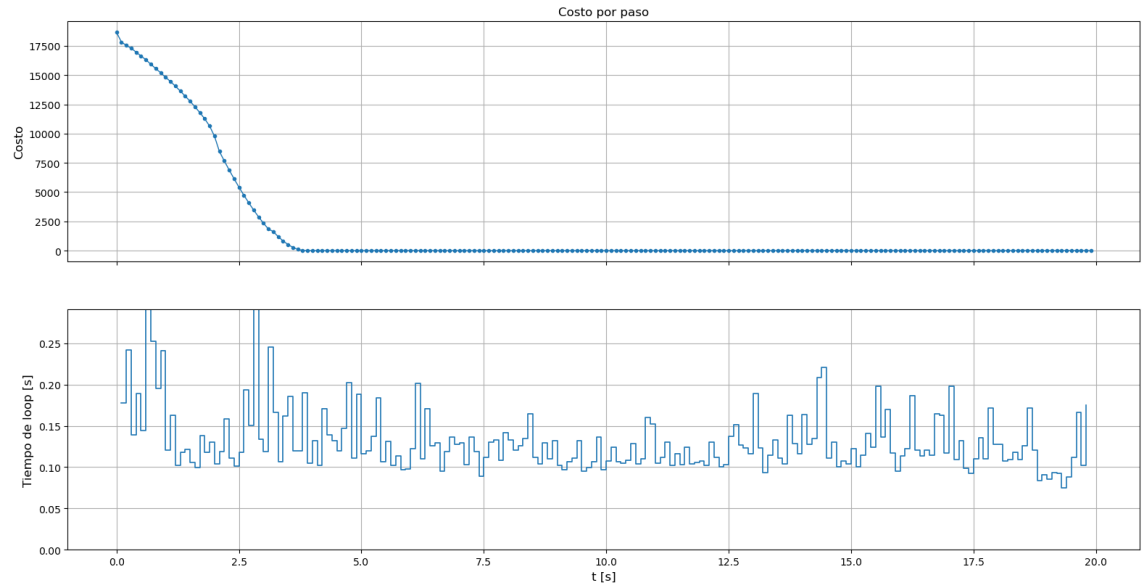


Figura 5.25: Evolución de la función de costo J (arriba) y tiempos de resolución del lazo MPC (abajo) durante la simulación con modelo dinámico de `Pinocchio`. Se observa una rápida disminución del costo en los primeros segundos, indicando la convergencia del controlador hacia el seguimiento deseado. En la gráfica inferior se aprecia que la mayoría de los pasos superan el tiempo de muestreo de $dt = 0{,}1\,\text{s}$, con un tiempo medio de resolución de aproximadamente $138\,\text{ms}$ por iteración.

De acuerdo con los registros mostrados, de los 200 pasos totales, **179 superaron el tiempo de muestreo configurado de** $dt = 0{,}1\,\mathbf{s}$. Si bien existen algunos picos que superan los 250 ms, se obtiene un tiempo promedio de resolución de **144 ms por paso**. Esto evidencia que, si bien el controlador logra resolver el problema de optimización en cada iteración, el costo computacional del enfoque actual, basado en la integración dinámica con `Pinocchio` y en la optimización por gradiente RMSprop, resulta elevado para ejecución en tiempo real.

Este comportamiento es coherente con lo discutido en la Sección 3.5: la integración explícita del modelo físico, junto con el cálculo de la optimización de la función de costo, esto introduce una sobrecarga significativa en cada ciclo de control. No obstante, la consistencia en los resultados y la estabilidad del costo validan el correcto funcionamiento del esquema predictivo.

5.9.3. Caso 2: Predicción de la dinámica mediante PINN

En este segundo caso de implementación, el modelo dinámico utilizado por el controlador predictivo no se obtiene a partir de una formulación analítica explícita, sino mediante una red neuronal informada por la física (*Physics-Informed Neural Network*, PINN). Este enfoque permite reemplazar el modelo rígido derivado del URDF por una aproximación aprendida que captura directamente las relaciones no lineales entre las variables de estado y los torques de control, manteniendo coherencia con las ecuaciones diferenciales subyacentes al sistema.

A diferencia del caso anterior, donde la dinámica estaba implementada en `Pinocchio` y los gradientes debían calcularse de forma analítica, en este caso toda la formulación del modelo y de la función de costo se encuentra implementada dentro del entorno `TensorFlow`. Por este motivo, no es necesario derivar manualmente las expresiones del gradiente: el propio marco de `TensorFlow` permite obtener de manera automática las derivadas parciales requeridas mediante el mecanismo de diferenciación automática (*autograd*). Esto simplifica la estructura del controlador y permite un flujo de optimización completamente diferenciable, en el que tanto la predicción del modelo dinámico $f_\theta(\cdot)$ como la evaluación y minimización del costo J se integran de forma directa dentro del grafo computacional.

Parámetros de la implementación

La Tabla 5.6 resume los parámetros utilizados en la configuración experimental del controlador en conjunto con la predicción del modelo dinámico aprendida mediante la red *Physics-Informed Neural Network* (PINN), previamente entrenada y cuyos hiperparámetros se presentan en la Tabla 5.4.

En este caso, tanto el modelo dinámico $f_\theta(\cdot)$ como la función de costo J se encuentran implementados dentro del entorno de `TensorFlow`, lo que permite aprovechar la diferenciación automática (*autograd*) para la actualización directa de los torques de control a lo largo del horizonte de predicción. De esta forma, no es necesario calcular derivadas analíticas de la dinámica como en el caso anterior, simplificando el flujo de optimización y reduciendo la latencia computacional del lazo MPC.

En esta configuración, la red PINN predice la evolución dinámica del sistema de manera autoregresiva respetando el paso de muestreo dt, y a partir del estado actual $(q, \dot{q})$, los torques aplicados u y el tiempo t. Su integración completa dentro del grafo de cómputo permite que el optimizador `RMSprop` actualice directamente los torques sin requerir cálculos de gradiente externos, manteniendo una ejecución diferenciable, coherente y eficiente dentro del lazo MPC.

Tabla 5.6: Parámetros generales del controlador MPC utilizados en la simulación del brazo robótico con predicción mediante PINN.

Parámetro	Símbolo	Valor
Cantidad de articulaciones	n_{joints}	2
Grados de libertad	$q, \dot{q}$	2 (posiciones) + 2 (velocidades)
Paso de muestreo	dt	0.1 s
Tiempo total de simulación	T	20.0 s
Cantidad total de pasos	$N_{\text{total}} = T/dt$	200
Horizonte de predicción	N_2	7
Horizonte de control	N_u	4
Matriz de penalización estados	$Q_{2n\times 2n}$	diag(200, 200, 2, 2)
Matriz de penalización control	$R_{n\times n}$	diag(10^{-4}, 10^{-4})
Límite inferior de torque	$u_{\text{mín}}$	-3.0 N·m
Límite superior de torque	$u_{\text{máx}}$	+3.0 N·m
Optimizador empleado	–	RMSprop (lr = 10^{-2})

Ejecución del sistema

Al igual que en el Caso 1 descrito en la Subsección 5.9.2, la ejecución del sistema se llevó a cabo íntegramente en el entorno de simulación *Gazebo*, utilizando el modelo del brazo *OpenMANIPULATOR-X* provisto por el fabricante [29]. La principal diferencia respecto al caso anterior se da en el modelo dinámico empleado: en este escenario, la predicción de la evolución del sistema es realizada completamente por la red *Physics-Informed Neural Network* (PINN) entrenada previamente, tal como se detalla en 5.8.6. De esta manera, la red neuronal reemplaza el cálculo analítico de la dinámica proporcionado por `Pinocchio`, manteniendo la estructura general del controlador MPC y el mismo esquema de optimización.

La trayectoria cartesiana a seguir por el efector final es la misma utilizada en la simulación del Caso 1, representada en la Figura 5.22. Esta elección permite realizar una comparación directa entre ambos enfoques, analítico y basado en aprendizaje, bajo condiciones idénticas de referencia, horizonte de predicción y parámetros del controlador, garantizando así una evaluación objetiva del desempeño del MPC en función del modelo dinámico empleado.

Resultados de simulación

En la Figura 5.26 se muestran las posiciones articulares $q_1(t)$ y $q_2(t)$ junto con los torques calculados por el MPC basado en la PINN. Las trayectorias de referencia (en **naranja**) son seguidas con buena precisión general, mostrando un comportamiento más suave respecto al caso anterior. En particular, la articulación q_1 reproduce fielmente el barrido angular previsto entre 0 y 2,1 rad, mientras que q_2 mantiene una evolución continua y sin retrasos significativos. Se aprecia que la red logra predecir correctamente las dinámicas inerciales del sistema, permitiendo al optimizador ajustar los torques con menor fluctuación.

Los torques u_1 y u_2, representados en color **verde** sobre el eje secundario, presentan una evolución más suave y de menor amplitud que en el caso con modelo analítico. Los valores permanecen dentro del rango $[-1{,}5,\ 1{,}5]$ N·m, sin indicios de saturación, lo que evidencia la estabilidad del controlador y la capacidad de la red para aproximar adecuadamente la dinámica no lineal del brazo.

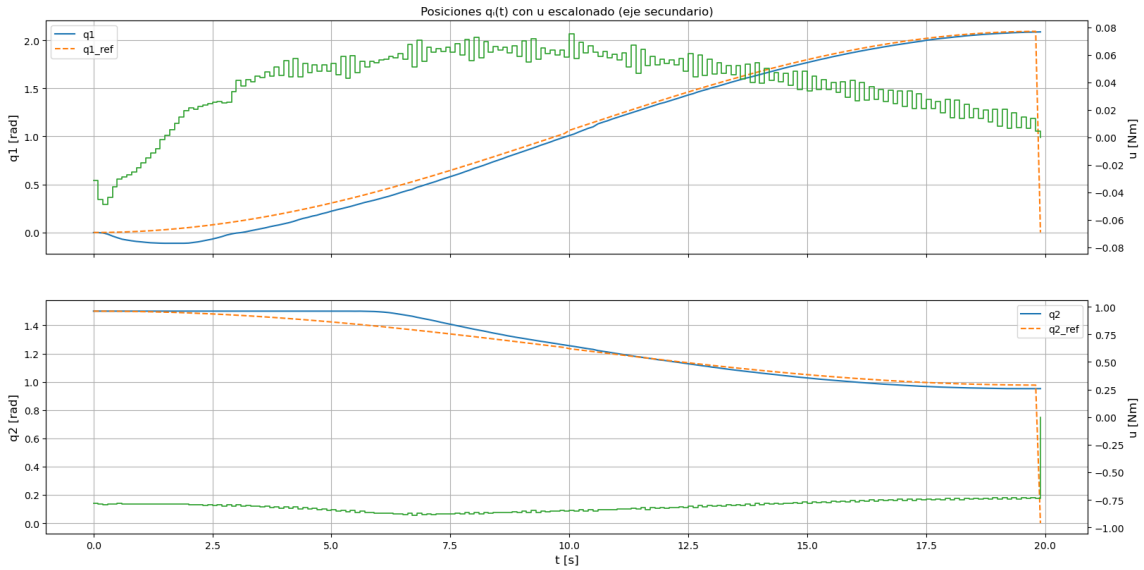


Figura 5.26: Evolución temporal de las posiciones articulares $q_1(t)$, $q_2(t)$ y torques de control u_1, u_2 durante la simulación del MPC con modelo dinámico aprendido mediante PINN. Las líneas naranjas representan las referencias deseadas, las azules los valores simulados, y las verdes los torques de control aplicados.

En la Figura 5.27 se presentan las velocidades articulares $\dot{q}_1(t)$ y $\dot{q}_2(t)$. La respuesta dinámica muestra una buena concordancia con las trayectorias de referencia, con un error menor al observado en el caso anterior y una clara reducción del ruido numérico. Este comportamiento más estable se debe a la naturaleza continua del modelo neuronal, que evita los efectos de discretización inherentes al cálculo de las derivadas en el modelo físico, resultando en una dinámica más regular y suave.

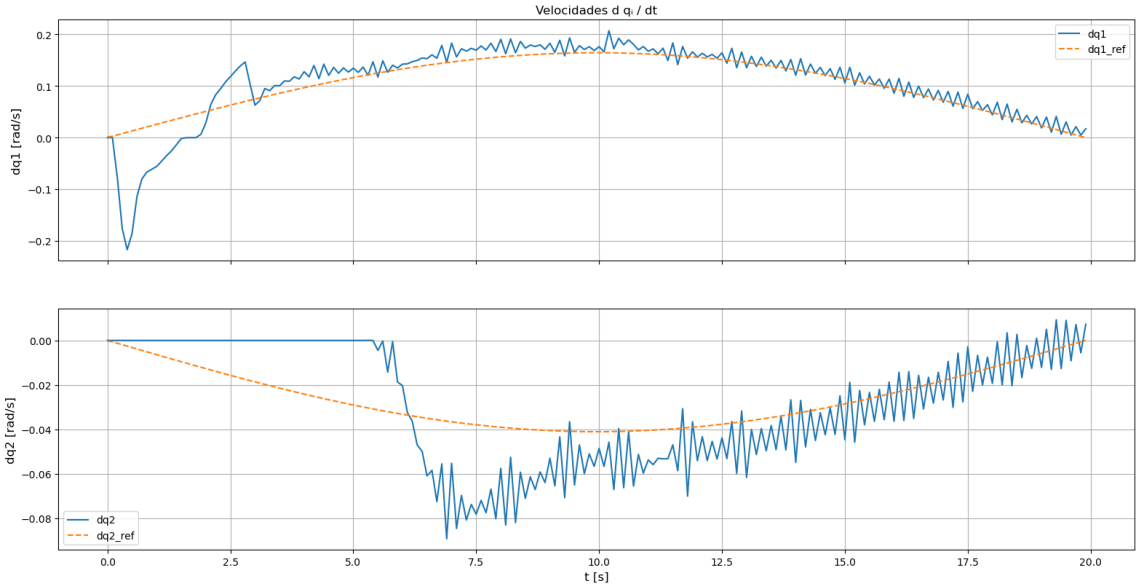


Figura 5.27: Evolución temporal de las velocidades articulares $\dot{q}_1(t)$ y $\dot{q}_2(t)$ obtenidas mediante el controlador MPC con modelo PINN.

Finalmente, la Figura 5.28 muestra la evolución de la función de costo J y los tiempos de cómputo del controlador a lo largo de la simulación. El valor del costo presenta un descenso rápido durante las primeras iteraciones, estabilizándose en torno a valores próximos a cero, lo que indica que el controlador logra un seguimiento eficiente. Los tiempos de cómputo,

presentados en la gráfica inferior de la Figura 5.28, se mantienen por debajo del umbral 100 ms en prácticamente todas las iteraciones, solo **33/200 superaron el tiempo de muestreo configurado**, evidenciando la mayor eficiencia computacional del enfoque basado en PINN.

En promedio, el lazo MPC alcanzó un tiempo de resolución de aproximadamente 92 ms por paso. Esto representa una mejora sustancial respecto al caso anterior con modelo de `Pinocchio`, cuyo tiempo medio fue de 144 ms por iteración. Por tanto, el uso del modelo aprendido mediante PINN no solo mantiene la estabilidad y precisión del control, sino que además reduce de manera significativa el costo computacional, permitiendo una ejecución más cercana a tiempo real.

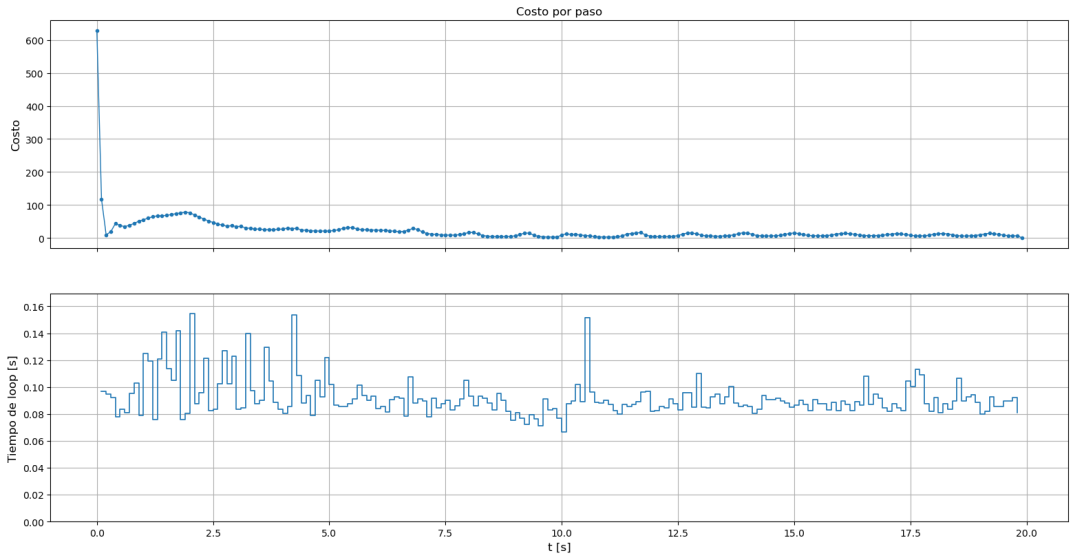


Figura 5.28: Evolución de la función de costo J (arriba) y tiempos de resolución del lazo MPC (abajo) durante la simulación con modelo PINN.

5.9.4. Análisis del desempeño de ambas implementaciones

El análisis comparativo entre ambas implementaciones del controlador MPC permite evaluar los efectos de utilizar diferentes modelos dinámicos internos, uno analítico basado en la librería `Pinocchio` y otro aprendido mediante una red PINN, sobre el desempeño general del sistema y la viabilidad de ejecución en tiempo real.

En primer lugar, los resultados muestran que el modelo de `Pinocchio`, aunque garantiza una descripción exacta de la dinámica del manipulador, implica un costo computacional considerable. El tiempo medio de resolución por iteración fue de aproximadamente 144 ms, con numerosos pasos superando el período de muestreo de $dt = 0{,}1$ s.

La sobrecarga se debe principalmente a la integración numérica del modelo de ecuaciones diferenciales en cada paso del horizonte predictivo y al cálculo de gradientes dentro del optimizador.

En contraste, el modelo dinámico aprendido mediante la PINN permitió reducir significativamente los tiempos de cómputo. El tiempo promedio de resolución del optimizador descendió a 92 ms. Esto representa una mejora superior al 50 % respecto al caso con modelo analítico. La reducción se debe a que la red neuronal reemplaza la integración explícita del modelo físico por una inferencia directa, manteniendo la diferenciabilidad necesaria

para el cálculo de gradientes del costo sin necesidad de resolver ecuaciones dinámicas en línea.

En términos de desempeño del control, ambos modelos lograron un seguimiento adecuado de las trayectorias de referencia, aunque con diferencias sutiles en la suavidad y estabilidad del movimiento. El modelo de `Pinocchio` ofreció una respuesta más precisa en las posiciones articulares, a costa de torques más abruptos y un mayor nivel de ruido numérico en las velocidades. Por su parte, la PINN mostró un comportamiento más regular y continuo, especialmente en las derivadas $\dot{q}_i(t)$, evidenciando una dinámica más estable y una menor sensibilidad a las variaciones discretas del horizonte de predicción.

No se observaron inestabilidades numéricas ni saturaciones de los torques en ninguno de los dos enfoques, confirmando la correcta elección de los parámetros de penalización en la función de costo.

5.9.5. Limitaciones

Si bien los resultados obtenidos demuestran la viabilidad del enfoque propuesto y el correcto funcionamiento del controlador predictivo, durante el desarrollo y la ejecución de las simulaciones se identificaron una serie de limitaciones tanto numéricas como estructurales que condicionan la escalabilidad del sistema hacia una implementación en tiempo real.

En primer lugar, el costo computacional del optimizador continúa siendo el principal factor restrictivo. Aun en el caso más eficiente con modelo PINN, el tiempo promedio de resolución del MPC se mantuvo en torno a 92 ms, valor que, si bien es inferior al período de muestreo definido ($dt = 0{,}1$ s), deja un margen temporal reducido para la ejecución de los demás procesos, como la comunicación entre nodos, el cálculo de la cinemática directa e inversa y la publicación de los torques al simulador. No obstante, es importante destacar que parte de la reducción en el tiempo de cómputo respecto a implementaciones previas, particularmente la presentada en la Sección 3.4.2 correspondiente al caso del oscilador de Van der Pol, se debió a la migración completa del código a operaciones vectorizadas mediante tensores. Esta modificación permitió aprovechar de forma más eficiente `Tensor Flow` y minimizar el uso de bucles explícitos, disminuyendo así el cálculo en cada iteración del optimizador. Asimismo, se sustituyó el algoritmo de optimización empleado originalmente por uno de menor complejidad computacional, lo que posibilitó incrementar el número de iteraciones por ciclo sin superar el período de muestreo. De este modo, el controlador logró aumentar la cantidad de iteraciones en la minimización de la función de costo y aproximarse a un control óptimo con menor tiempo de procesamiento total.

En consecuencia, si bien la capacidad de operar de manera determinista en tiempo real aún depende de la carga del sistema operativo y de las condiciones de ejecución, la combinación de estas mejoras representa un avance significativo hacia la ejecución en línea del esquema MPC.

Por ultimo otro aspecto a considerar es la dependencia del desempeño respecto a los parámetros del controlador MPC (Q, R, N_2, N_u). La sintonización de estos pesos se realizó de forma empírica, buscando un equilibrio entre precisión y suavidad de control.

Capítulo 6

Conclusiones

En este capítulo se presentan las conclusiones más relevantes de la realización del proyecto. En la Sección 6.1 se presentan conclusiones generales que refieren más al manejo de un proyecto, formas de trabajo e incluso también con métodos para mitigar riesgos y gestión de proyecto.

Por otra parte en la sección 6.2 se presentan las conclusiones técnicas del proyecto. Aquí se reflexiona sobre los aspectos técnicos, se analizan los objetivos iniciales y en particular en la Sección 6.3 se proponen mejoras y sugerencias para futuras iteraciones en la temática.

6.1. Conclusiones Generales

Como conclusión principal, se destaca la adecuada gestión del proyecto. Esto implica que el equipo logró conducir un trabajo de larga duración siguiendo las metodologías de gestión introducidas en el curso inicial. Asimismo, se mantuvo una comunicación efectiva entre las diferentes partes involucradas y, en términos generales, se alcanzaron los objetivos establecidos. Respecto a la forma de trabajo, se empleo una metodología con retroalimentación constante entre los integrantes y el tutor, con reuniones cada dos semanas y hasta semanales en el tramo final del trabajo. Gracias a este seguimiento se lograron sortear a tiempo los distintos inconvenientes, planteando objetivos semanales y alternativas ante posibles altercados. Si bien el proyecto contó con un tiempo extra respecto a lo planificado originalmente debido a limitaciones con respecto a capacidad computacional que llevó a incorporar la herramienta ClusterUy, se considera que de no ser por esta limitación, el trabajo hubiese finalizado en el plazo original gracias a dicho seguimiento. Dicho problema de capacidad computacional surgió como consecuencia de haber subestimado la alta dimensionalidad del problema inicial, optando por reducir la complejidad del brazo. Por otra parte, el trabajo en equipo y la división de tareas resultó vital. Dado que el presente proyecto abarcaba diversas áreas de la ingeniería, la asignación de tareas según el conocimiento específico de cada integrante adquirido tanto por el perfil de carrera elegido como por experiencias previas, resultó ser una estrategia fundamental. Es importante aclarar que esta división del trabajo no implicó una segmentación que obstaculizara la interacción entre los integrantes, por el contrario, se generaron espacios de intercambios y discusión constructiva para el desarrollo del proyecto.

6.2. Conclusiones Técnicas

6.2.1. Conclusiones técnicas sobre las PINNs

En general, las PINNs resultaron ser una gran alternativa para modelar problemas físicos complejos. Una de las contribuciones más relevantes es su eficiencia computacional en comparación con los métodos iterativos tradicionales. Gracias a su naturaleza de entrenamiento *offline*, el esfuerzo computacional se concentra en la fase de aprendizaje, mientras que la etapa de inferencia o predicción resulta extremadamente rápida. Esta ventaja es especialmente notable en contextos donde las evaluaciones repetidas del modelo son frecuentes, como en control predictivo, optimización o simulaciones en tiempo real. En este sentido, tanto las PINNs puras como las PINCs —una extensión que incluye variables de control en su formulación— demuestran un alto potencial para reemplazar solvers iterativos, reduciendo significativamente los costos computacionales en la etapa de despliegue.

Además, las PINNs resultan especialmente prometedoras en dominios donde las ecuaciones físicas teóricas son incompletas o difíciles de formular, como ocurre en dinámica de fluidos, transferencia de calor en medios heterogéneos, materiales no lineales o procesos biológicos complejos. En estos casos, las PINNs permiten inferir la dinámica subyacente directamente a partir de datos experimentales, al tiempo que imponen consistencia física mediante la inclusión parcial o total de las leyes conocidas. De esta manera, actúan como un puente entre el modelado empírico y el modelado teórico, ofreciendo una alternativa híbrida de gran valor en contextos donde el conocimiento físico es limitado o incierto.

No obstante, el uso de PINNs conlleva desafíos significativos. Uno de los principales reside en la selección adecuada de los hiperparámetros de la red —como la arquitectura, las funciones de activación, las tasas de aprendizaje y los pesos relativos de los términos de la función de pérdida—, que influyen de manera crítica en la estabilidad y la precisión del entrenamiento. La falta de guías teóricas sólidas en este aspecto obliga a recurrir a estrategias empíricas o de búsqueda sistemática, lo que puede aumentar los tiempos de desarrollo y la variabilidad de los resultados.

Otro reto importante está relacionado con la **escalabilidad**. A medida que aumenta la dimensionalidad del problema o la complejidad del sistema físico, el número de puntos de colocación y la cantidad de derivadas necesarias crece rápidamente, lo que repercute directamente en el costo computacional del entrenamiento. Si bien las PINNs pueden ofrecer ventajas de inferencia en tiempo real, el entrenamiento de modelos de alta dimensión sigue siendo costoso y, en algunos casos, inabordable con los recursos de cómputo convencionales.

En síntesis, las PINNs constituyen una herramienta poderosa y versátil para el aprendizaje de sistemas gobernados por ecuaciones diferenciales, combinando lo mejor del aprendizaje automático y del modelado físico. Su capacidad para generalizar desde datos escasos y su eficiencia en la predicción las convierten en una alternativa atractiva frente a los métodos tradicionales, especialmente en escenarios de control y simulación en tiempo real.

6.2.2. Conclusiones técnicas sobre MPC y ROS

Desde el punto de vista técnico, la implementación del controlador predictivo (MPC) dentro del entorno ROS 2 permitió validar el esquema propuesto, pero también evidenció diversas dificultades prácticas vinculadas a la optimización y la integración entre nodos.

En primer lugar, el tiempo de resolución del problema de optimización se presentó como una de las principales limitaciones del sistema. Aún con las mejoras introducidas, como la vectorización mediante tensores y el uso del modelo PINN para reemplazar la integración dinámica explícita, los tiempos de cómputo continúan siendo elevados para una ejecución determinista en tiempo real. En particular, el optimizador RMSprop mostró una convergencia estable pero relativamente lenta, lo que restringe el número de iteraciones posibles dentro del período de muestreo del lazo de control.

Por otro lado, la configuración del MPC requirió un ajuste manual de los parámetros, entre ellos los horizontes de predicción (N_2, N_u) y las matrices de ponderación Q y R. Esto dificultó el desempeño y exigió mayores pruebas empíricos hasta alcanzar resultados aceptables, estables y precisos del seguimiento de la trayectoria.

Finalmente, las pruebas comparativas entre el modelo analítico de `Pinocchio` y el modelo aprendido mediante PINN demostraron una mejora sustancial en el costo computacional del lazo de control. El uso de la PINN entrenada prediciendo el modelo dinamico del brazo permitió reducir el tiempo promedio por iteración, obteniendo un rendimiento similar.

En síntesis, la integración del MPC dentro de ROS 2 fue correctamente implementada reduciendo el costo de cómputo aunque en un robot en el cual se limitaron los grados de libertad.

6.3. Trabajo futuro

6.3.1. Trabajo futuro sobre PINNs

El modelo desarrollado constituye una primera aproximación al uso de redes neuronales informadas por física (PINNs) en el contexto de sistemas robóticos. A futuro, existen diversas direcciones que podrían explorarse con el objetivo de evaluar mejoras en capacidad de representación, estabilidad numérica y generalización. A continuación se presentan futuras alternativas a explorar:

Arquitecturas En este trabajo se evaluaron distintas configuraciones de red y, finalmente, se adoptó una arquitectura *fully connected* (MLP). La elección se fundamenta en que este tipo de arquitectura constituye la opción más simple e inmediata de implementar, al no requerir estructuras adicionales ni codificaciones especiales en las entradas. Además, las redes totalmente conectadas presentan buena capacidad de aproximación universal y, en numerosos trabajos previos con PINNs, han demostrado ofrecer resultados satisfactorios en problemas de dinámica robótica y modelado físico. Existen otros caminos a explorar en este ámbito, algunos de los cuales fueron abordados durante la fase de investigación. En particular, resulta intuitivo pensar que, en lugar de una única red encargada de aprender simultáneamente todas las relaciones dinámicas del sistema, puede ser más eficiente y estable dividir el aprendizaje en subredes especializadas, cada una dedicada a un componente físico distinto de la ecuación de movimiento. En [13] se explora la idea de una arquitectura modular que consiste de subredes que se encargan de representar diferentes componentes de la dinámica del manipulador, combinando las salidas para obtener la representación completa de un sistema.

Muestreo temporal adaptativo En una de las publicaciones más influyentes en este proyecto se presentan las PINNs [24], y en este documento ya se introduce la noción de muestreo adaptativo de puntos de colocación, conocida como *Residual-based Adaptive*

Refinement (RAR). Este método propone evaluar el residuo físico de la ecuación diferencial durante el entrenamiento y agregar nuevos puntos en aquellas regiones del dominio donde dicho residuo es elevado. De esta forma, el modelo concentra su capacidad de aprendizaje en las zonas donde la red viola más la física, mejorando la precisión sin necesidad de incrementar drásticamente la cantidad total de puntos de entrenamiento. En el contexto de un manipulador robótico, una estrategia de este tipo podría emplearse para reforzar el aprendizaje en regiones del espacio articular donde la dinámica resulta más compleja o donde se observan mayores errores en el residuo dinámico.

Ponderación de pérdidas Es sabido y esta ampliamente documentado que uno de los principales problemas de las PINNs es el entrenamiento multi-objetivo (por ejemplo, [34]) que se da al involucrar funciones de costo compuestas en la optimización. Sobre esto hay muchas técnicas presentadas, algunas se intentaron aplicar sin éxito durante el proyecto, lo que dejo un precedente para profundizar en estas ideas.

Optimización de hiperparametros En las PINNs, la elección de los hiperparámetros tiene un impacto considerable sobre los resultados obtenidos. A diferencia de las redes usuales, donde estos parámetros solo afectan la convergencia o el overfitting, en una PINN también determinan cuán bien la red logra respetar las leyes físicas. Es por esto que una posible línea a explorar a futuro es la automatización de la búsqueda de de hiperparametros para no depender de heurística/intuición.

Brazo físico Una línea de trabajo futura muy interesante es la experimentación directa con el brazo real, en lugar de basarse únicamente en simulaciones. Esto permitiría registrar mediciones reales de posiciones, velocidades y torques, generando así un conjunto de datos que refleje las particularidades físicas del sistema: fricción, flexibilidad, holguras, retardos o imprecisiones de los sensores y actuadores. Con esta información podría entrenarse un modelo data-driven complementario o un esquema híbrido (Physics + Data), donde la PINN mantenga la estructura física general pero aprenda también a corregir los desvíos entre el modelo ideal y el comportamiento real del robot.

Entrenamiento en lazo cerrado Otra línea de trabajo interesante sería implementar esquemas de entrenamiento en línea, donde la red se entrene en interacción directa con una simulación o con el robot físico. Esto podría llevar a enfoques similares al aprendizaje por refuerzo, donde el modelo aprende a partir de la retroalimentación continua del entorno. En este caso, la PINN no solo ajustaría sus parámetros para cumplir la física, sino también para mejorar su desempeño en una tarea concreta.

6.3.2. Trabajo futuro sobre MPC

Por otro lado, respecto del controlador MPC y su integración en ROS 2, el siguiente paso consiste en trasladar la arquitectura desarrollada desde el entorno de simulación hacia el brazo real *OpenManipulator-X*. Esto permitirá validar el desempeño del controlador frente a condiciones físicas reales, tales como fricción, flexibilidad estructural, latencias de comunicación y ruido sensorial. Para ello será necesario optimizar los tiempos de cómputo del lazo de control, de modo que el cálculo del torque óptimo pueda realizarse dentro del periodo de muestreo requerido por el sistema físico.

Una posible línea de mejora consiste en la implementación parcial del solucionador del MPC en bajo nivel. De esta manera, se podría reducir el tiempo de resolución del problema de optimización en cada iteración.

Asimismo, se propone mejorar el nodo `trajectory_node` para habilitar la generación de referencias en tiempo real. A diferencia del esquema actual, donde las trayectorias se definen de forma previa, un planificador online permitiría modificar dinámicamente los puntos de referencia en función de las condiciones del entorno o de la detección de nuevos objetivos.

Finalmente, se plantea incorporar un sistema visual mediante la instalación de una cámara en el efector final del manipulador. El procesamiento de imágenes en tiempo real, combinado con la planificación adaptativa y el control predictivo, permitiría la ejecución de tareas autónomas más complejas, como el seguimiento visual de objetos o el posicionamiento preciso frente a un marcador.

Estas líneas de trabajo futuro apuntan a consolidar un sistema completamente operativo sobre hardware real, capaz de percibir, planificar y actuar de manera autónoma dentro de su entorno físico.

Esta página ha sido intencionalmente dejada en blanco.

Referencias

[1] Pytorch: Defining new autograd functions. https://docs.pytorch.org/tutorials/beginner/examples_autograd/polynomial_custom_function.html, 2025. Accessed: 2025-11-10.

[2] Eric Aislan Antonelo, Eduardo Camponogara, Laio Oriel Seman, Jean Panaioti Jordanou, Eduardo Rehbein de Souza, and Jomi Fred Hübner. Physics-informed neural nets for control of dynamical systems. *Neurocomputing*, 579:127419, 2024.

[3] Jonathan Bohren and Dave Coleman. gazebo_ros2_control documentation index. https://github.com/ros-controls/gazebo_ros2_control/blob/master/doc/index.rst, 2025. Accessed: 2025-09-29.

[4] E. F. Camacho and C. Bordons. *Model Predictive Control*. Springer, 2nd edition, 2007.

[5] Christian A Díaz Cuadro, Mauricio C Vanzulli, and Pedro A Galione. Exploring the capability of pinns for solving material identification problems. *Mecánica Computacional*, 40(32):1183–1194, 2023.

[6] Roy Featherstone. *Rigid Body Dynamics Algorithms*. Springer, New York, 2008.

[7] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[8] Hassan K. Khalil. *Nonlinear Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1996.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Hüseyin Kutluca. Robot operating system 2 (ros 2) architecture. https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776, 2020.

[11] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[12] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

[13] Jingyue Liu, Pablo Borja, and Cosimo Della Santina. Physics-informed neural networks to model and control robots: a theoretical and experimental investigation, 2023.

[14] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. The robot operating system 2 design. https://design.ros2.org/, 2022.

[15] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[16] Jan M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002.

[17] David Q. Mayne, James B. Rawlings, Christopher V. Rao, and Peter O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, 2000.

[18] M.D. McKay, R.J. Beckman, and W.J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[19] Sergio Nesmachnow and Santiago Iturriaga. Cluster-uy: Collaborative scientific high performance computing in uruguay. In Moisés Torres and Jaime Klapp, editors, *Supercomputing*, pages 188–202, Cham, 2019. Springer International Publishing.

[20] J. E. Normey-Rico and E. F. Camacho. *Control of Dead-Time Processes*. Springer London, 2007.

[21] Open Robotics. Ros middleware interface (rmw). `https://design.ros2.org/articles/ros_middleware_interface.html`, 2017. Accedido: septiembre 2025.

[22] Open Robotics. Ros 2 concepts. `https://docs.ros.org/en/foxy/Concepts.html`, 2020. Accedido: septiembre 2025.

[23] Open Robotics. Ros 2 rolling ridley — official documentation. `https://docs.ros.org/en/rolling/Releases.html`, 2025.

[24] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[25] James B. Rawlings and David Q. Mayne. *Model Predictive Control: Theory and Design*. Nob Hill Publishing, Madison, WI, 2009.

[26] James B. Rawlings, David Q. Mayne, and Moritz Diehl. *Model Predictive Control: Theory, Computation, and Design*. Nob Hill Publishing, 2nd edition, 2017.

[27] ROBOTIS. *OpenMANIPULATOR-X e-Manual*, 2020.

[28] ROBOTIS. Dynamixel xm430-w350 specifications. `https://emanual.robotis.com/docs/en/dxl/x/xm430-w350/`, 2025. Accessed: 2025-09-29.

[29] ROBOTIS. Openmanipulator-x specification. `https://emanual.robotis.com/docs/en/platform/openmanipulator_x/specification/`, 2025. Accessed: 2025-09-29.

[30] ROS Wiki URDF. Urdf: Unified robot description format. `http://wiki.ros.org/urdf`, 2025.

[31] Addison Sears-Collins. Ros2 architecture overview. `https://automaticaddison.com/ros-2-architecture-overview/`, 2023.

[32] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, Hoboken, New Jersey, 2006.

[33] University of Freiburg. do-mpc: Theory of model predictive control. `https://www.do-mpc.com/en/latest/theory_mpc.html`, 2025. Accessed: October 2025.

[34] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

[35] S. Xu et al. On the preprocessing of physics-informed neural networks. *Journal of Computational Physics*, 2025. arXiv preprint arXiv:2403.19923.

Esta página ha sido intencionalmente dejada en blanco.

Índice de tablas

Esta página ha sido intencionalmente dejada en blanco.

Índice de figuras

Esta es la última página.
Compilado el jueves 13 noviembre, 2025.
http://iie.fing.edu.uy/