**Universidad Internacional de la Rioja (UNIR)**

## Escuela Superior de Ingeniería y Tecnología

**Máster Universitario en Ingeniería Matemática y Computación**

# RLIC-FMU: Control inteligente mediante aprendizaje por refuerzo profundo en modelos de sistemas físicos en Modelica

Con el apoyo de:

XRG Simulation GmbH

**Trabajo Fin de Máster**

**presentado por:** Pedro Padilla Quesada

**Dirigido por:** Daniel de la Rosa Gómez

Ciudad:Reading, Reino Unido

Fecha: 15 de septiembre de 2025

*A la memoria de mi madre.*

# Índice de Contenidos

# Índice de Ilustraciones

# Índice de Tablas

# Listings

# Agradecimientos

Agradezco sinceramente a *Dr.-Ing. Robert Flesch* y *Dr. Johannes Brunnemann*, de *XRG Simulation GmbH*, por la idea propuesta para este trabajo y por el apoyo y soporte continuo durante su desarrollo.

# Resumen

Este trabajo presenta el desarrollo de `RLIC-FMU`, una herramienta software en Python diseñada para entrenar y evaluar controladores inteligentes mediante aprendizaje por refuerzo profundo (DRL) sobre modelos de sistemas físicos exportados como FMU bajo el estándar FMI. El objetivo es integrar técnicas modernas de control basado en aprendizaje en entornos de simulación multidominio, cubriendo un vacío existente en el ecosistema Modelica. La metodología incluye el diseño de una arquitectura modular basada en `TF-Agents` y `PyFMI`, la implementación de entornos personalizados y el entrenamiento de agentes DRL (DQN y DDPG) sobre casos de estudio de complejidad creciente. Se ha demostrado la viabilidad del enfoque con resultados competitivos frente a controladores clásicos y la exportación de políticas entrenadas en TensorFlow Lite para su uso directo en OpenModelica mediante `SMArtInt`. La herramienta proporciona una base sólida para futuras investigaciones, incluyendo algoritmos más eficientes y aplicaciones industriales de control avanzado.

**Palabras clave:** Aprendizaje por refuerzo profundo, Modelica, FMI, control inteligente, simulación multidominio.

# Abstract

This work presents the development of `RLIC-FMU`, a Python software tool designed to train and evaluate intelligent controllers using deep reinforcement learning (DRL) on physical system models exported as FMUs under the FMI standard. The goal is to integrate modern learning-based control techniques into multi-domain simulation environments, addressing a gap in the Modelica ecosystem. The methodology includes a modular architecture built on `TF-Agents` and `PyFMI`, the implementation of custom environments, and the training of DRL agents (DQN and DDPG) on progressively complex case studies. The feasibility of the approach has been demonstrated with competitive results compared to classical controllers and by exporting trained policies in TensorFlow Lite for direct use in OpenModelica through `SMArtInt`. The tool provides a solid foundation for future research, including more sample-efficient algorithms and industrial applications of advanced control.

**Keywords:** Deep reinforcement learning, Modelica, FMI, intelligent control, multi-domain simulation.

# Capítulo 1

# Introducción

Muchas de las grandes innovaciones en ingeniería y tecnología no serían posibles sin herramientas de modelado y simulación, que permiten analizar y optimizar el comportamiento de sistemas complejos de forma segura y económica. Gracias a ellas, es posible experimentar virtualmente con sistemas como aviones o plantas nucleares, sin necesidad de construir prototipos físicos.

El modelado y la simulación de sistemas es una de las áreas más relevantes de la ingeniería matemática, ya que permite diseñar, analizar y controlar sistemas físicos en diversas industrias, aprovechando su carácter multidominio. Con el surgimiento del paradigma de modelado físico y el desarrollo del lenguaje Modelica, esta tarea se ha simplificado notablemente. Modelica permite describir sistemas mediante ecuaciones en su forma natural, sin necesidad de especificar la causalidad computacional, como sí ocurre en los enfoques basados en diagramas de bloques.

En la práctica industrial, muchos sistemas presentan tal nivel de complejidad que las técnicas de control convencional resultan insuficientes. Para hacer frente a estos desafíos, ha surgido el campo del control inteligente, que combina los fundamentos del control clásico con técnicas de inteligencia artificial. Este enfoque ha sido aplicado con éxito en ámbitos como la robótica y el control automático, empleando métodos como la lógica difusa, las redes neuronales o los algoritmos genéticos, que permiten desarrollar sistemas más adaptativos, robustos y capaces de operar en entornos inciertos o no lineales.

El uso de redes neuronales profundas ha sido fundamental en el auge actual de la inteligencia artificial, especialmente en el campo del aprendizaje automático. Aunque sus bases teóricas se remontan a mediados del siglo XX, su aplicación práctica se ha visto

impulsada en las últimas décadas gracias al aumento de la capacidad de cómputo, la mejora de algoritmos y la disponibilidad de software de código abierto, que ha democratizado su uso incluso entre quienes no tienen una formación matemática avanzada. Estas redes son también el pilar de las técnicas modernas de aprendizaje por refuerzo profundo, que permiten a los sistemas aprender a tomar decisiones óptimas a través de la experiencia, sin necesidad de contar con un modelo completo del entorno. Esta combinación de aprendizaje por refuerzo y redes neuronales ha abierto nuevas posibilidades en el desarrollo de controladores inteligentes, especialmente en entornos complejos e inciertos.

Este trabajo se enmarca en esa dirección y persigue integrar el aprendizaje por refuerzo en entornos de modelado y simulación de sistemas físicos, como paso hacia el desarrollo de controladores más inteligentes, adaptativos y autónomos.

### 1.0.1. Planteamiento del problema y finalidad del trabajo

Aunque las técnicas de control convencional siguen siendo mayoritarias en la simulación de sistemas dinámicos, sustentadas por una teoría sólida y ampliamente desarrollada, existen numerosas situaciones en las que estas metodologías resultan insuficientes. En particular, sistemas con comportamientos complejos o sujetos a altos niveles de incertidumbre demandan soluciones más flexibles y adaptativas. En este contexto, el control inteligente, que integra métodos tradicionales con herramientas de inteligencia artificial como la lógica difusa, las redes neuronales o los algoritmos genéticos, ha demostrado un gran potencial en diversas aplicaciones, gracias a su capacidad de adaptación y de aprendizaje a partir de experiencias previas (Linkens y Nyongesa, 1996).

Sin embargo, los principales entornos de modelado y simulación basados en el lenguaje Modelica, como OpenModelica o Dymola, no ofrecen soporte nativo para el desarrollo y validación de controladores inteligentes. Aunque existen alternativas para conectar modelos desarrollados en Modelica con frameworks de aprendizaje automático, estos procesos suelen ser complejos, poco sistemáticos y requieren una considerable inversión de tiempo y esfuerzo manual.

En este contexto, y fruto de una colaboración con la empresa alemana XRG Simulation GmbH[1], surge la propuesta de desarrollar una herramienta en Python que permita entrenar

[1]Empresa de ingeniería con sede en Hamburgo, miembro de la *Modelica Association* y desarrolladora de la biblioteca `SMArtInt` para integrar modelos de IA en simulaciones Modelica. Más información en https://xrg-simulation.de.

controladores inteligentes mediante aprendizaje por refuerzo, aplicados a modelos físicos construidos en Modelica. Esta herramienta establecerá una comunicación directa con la biblioteca TensorFlow Agents, facilitando así la integración entre el entorno de simulación y las técnicas modernas de aprendizaje automático.

Para abordar este desafío y cumplir con los objetivos del trabajo, se adoptará un enfoque multidisciplinar, que integrará conocimientos adquiridos a lo largo del programa de Máster en Ingeniería Matemática y Computación. En particular, se requerirá la aplicación de técnicas avanzadas de programación (como la orientación a objetos), conceptos de modelado y simulación de sistemas físicos, fundamentos de aprendizaje automático, y nociones de procesos estocásticos para la representación de ciertas variables dinámicas del sistema.

### 1.0.2. Justificación de su novedad

La integración del aprendizaje por refuerzo en entornos de modelado físico representa una línea de investigación y desarrollo aún poco explorada, a pesar del enorme potencial que ofrece. Aunque las técnicas de inteligencia artificial han revolucionado diversos campos de la ingeniería y la computación, su aplicación directa sobre sistemas modelados en lenguajes como Modelica sigue siendo limitada, en gran parte debido a la falta de herramientas que permitan una interoperabilidad sencilla, eficiente y estructurada entre estos dos mundos.

Los entornos de simulación basados en Modelica, como OpenModelica o Dymola, han sido diseñados principalmente para análisis deterministas, bajo esquemas de control clásico o mediante estudios de sensibilidad y optimización. En cambio, el aprendizaje por refuerzo se basa en la interacción iterativa con el entorno, lo cual exige una arquitectura flexible, con comunicación bidireccional en tiempo de simulación. Esto implica no solo la ejecución de simulaciones repetidas, sino también la modificación dinámica de entradas en función de las decisiones aprendidas por el agente inteligente. La fusión de Modelica con los algoritmos de aprendizaje profundo por refuerzo permitiría automatizar el desarrollo de controladores inteligentes adaptativos para sistemas complejos, algo que actualmente solo es posible mediante soluciones artesanales, poco accesibles o comerciales, que además no siempre ofrecen soporte para el lenguaje Modelica ni son de código abierto.

La novedad de este trabajo radica precisamente en el puente que establece entre el modelado físico y el aprendizaje automático por refuerzo, mediante el desarrollo de una

herramienta en Python que permite conectar modelos en Modelica con la biblioteca TensorFlow Agents. Esta integración no solo automatiza procesos que antes requerían intervención manual, sino que abre la puerta a nuevos métodos de diseño y validación de controladores inteligentes, entrenados directamente sobre modelos físicos sin necesidad de aproximaciones intermedias o simplificaciones excesivas.

Además, este enfoque responde a una necesidad real en la industria: el control de sistemas dinámicos complejos, con componentes multidominio, y bajo condiciones inciertas, donde las soluciones tradicionales ya no resultan eficaces. La posibilidad de entrenar controladores adaptativos directamente sobre modelos de planta desarrollados en Modelica, con una arquitectura abierta y extensible, constituye una innovación con gran potencial de impacto tanto en el ámbito académico como industrial. Esta capacidad no solo reducirá tiempos de desarrollo y validación, sino que también abrirá nuevas oportunidades para la innovación en el diseño de sistemas de control adaptativos y autónomos.

### 1.0.3. Contribución general del trabajo

Este Trabajo Fin de Máster aporta las siguientes contribuciones generales:

- Propuesta y desarrollo de la herramienta software abierta `RLIC-FMU` (*Reinforcement Learning for Intelligent Control of Functional Mock-up Units*), que conecta el modelado físico basado en Modelica con técnicas modernas de aprendizaje por refuerzo profundo, facilitando el entrenamiento de controladores inteligentes directamente sobre modelos de simulación de alta fidelidad.
- Integración de conceptos de ingeniería matemática, aprendizaje automático y simulación multidominio para demostrar el potencial del aprendizaje por refuerzo en el control avanzado de sistemas complejos.
- Validación experimental mediante casos de estudio representativos que muestran la viabilidad y ventajas del enfoque propuesto.
- Identificación de retos y líneas de investigación futura para optimizar y escalar la metodología, sentando las bases para aplicaciones industriales.

Estas contribuciones se desarrollan y justifican a lo largo del documento, y están directamente vinculadas con los objetivos planteados en el Capítulo 3 y con el análisis de su grado de consecución recogido en el Capítulo 5.

### 1.0.4. Estructura del trabajo

En el Capítulo 2 se presentan los fundamentos teóricos necesarios para contextualizar el problema, abarcando el modelado y la simulación de sistemas físicos, el aprendizaje por refuerzo profundo y el estado del arte (técnicas, bibliotecas y trabajos relacionados).

El Capítulo 3 expone el objetivo general y los objetivos específicos del trabajo, e incluye la *metodología* seguida para alcanzarlos (fases de revisión, desarrollo software, casos de estudio, entrenamiento y validación).

El Capítulo 4 constituye el núcleo del proyecto: se describe la herramienta `RLIC-FMU`, los modelos implementados y su configuración, el proceso de entrenamiento y evaluación de controladores, y se presenta la integración y prueba de políticas en OpenModelica mediante SMArtInt como demostración de despliegue.

Por último, el Capítulo 5 recoge las conclusiones principales, el *grado de consecución* de los objetivos y las líneas de trabajo futuro.

# Capítulo 2

# Contexto y Estado del Arte

## 2.1. Fundamentos teóricos

### 2.1.1. Modelado y simulación de sistemas físicos

Un sistema es un conjunto de entidades que actúan para conseguir un objetivo. En un sistema físico, estas entidades son objetos físicos que interactúan entre sí y con el entorno de acuerdo a las leyes de la Física. A la hora de analizar y predecir el comportamiento de estos sistemas físicos, se pueden encontrar problemas, por tanto, en cada uno de los dominios de la Física. Por ejemplo, problemas en el dominio mecánico, en el eléctrico, el electromecánico, fluidodinámica o térmico, sin pretender ser exhaustivo.

El modelado y la simulación son herramientas esenciales para el estudio y el análisis de sistemas físicos. Un modelo es simplemente una representación del sistema que pretende estudiar. Existen dos enfoque básicos para el modelado de sistemas. El primero es el empleo de modelos físicos, que son representaciones tangibles del sistema con el que realizar experimentos físicos reales. El segundo es el uso de modelos matemáticos, que representan el sistema mediante ecuaciones. Este trabajo se refiere exclusivamente al modelado matemático, y, por simplificación, se usará simplemente el término modelado para referirse a él.

Existen diferentes métodos para modelar y simular sistemas físicos. Este trabajo se centrará exclusivamente en la simulación de tiempo continuo. Los modelos de tiempo continuo se caracterizan por el número infinito de veces que las variables del modelo pueden cambiar. Por ejemplo, el valor de la caída de tensión en el condensador de un circuito eléctrico o el nivel de líquido en un depósito son variables continuas.

Dado que es imposible calcular el valor de estas variables en infinitos instantes de tiempo, la simulación de modelos de tiempo continuo se realiza mediante una discretización temporal, es decir, se calcula su valor sólo en determinados instantes de tiempo.

Urquía y Martín (2018) presentan una formalización matemática para el modelado y simulación de los sistemas físicos. Los sistemas dinámicos pueden modelarse a través de sistemas de ecuaciones diferenciales ordinarias (sistemas ODE, del inglés *ordinary differential equation*). Estos sistemas pueden expresarse como sistemas ODE explícitos si las variables derivadas aparecen despejadas:

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}), \tag{2.1}$$

o como sistemas ODE implícitos si están expresados de la forma:

$$\mathbf{F}(t, \mathbf{x}, \dot{\mathbf{x}}) = 0, \tag{2.2}$$

Las variables que aparecen derivadas en el modelo, $\dot{\mathbf{x}}$, se conocen como *variables de estado*. Estas variables determinan el estado del sistema en un instante de tiempo, es decir, describen cada uno de los elementos, atributos y actividades en dicho instante. Este concepto será clave para controlar el sistema mediante el aprendizaje por refuerzo, como se explicará más adelante.

Por lo general, un sistema ODE no es suficiente para modelizar la mayoría de sistemas físicos. En la mayoría de modelos se necesitan no sólo ecuaciones diferenciales ordinarias respecto al tiempo, sino también ecuaciones que describen ligaduras algebraicas entre las variables. Este tipo de modelos se denominan sistemas de ecuaciones algebraico diferenciales (DAE, del inglés *differential-algebraic system of equations*).

Las ligaduras algebraicas pueden o no aparecer explícitamente. Cuando aparecen explícitamente, el sistema DAE se puede modelar como:

$$\begin{aligned} \mathbf{F}(t, \mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}) &= 0 \\ \mathbf{G}(t, \mathbf{x}, \mathbf{y}) &= 0 \end{aligned} \tag{2.3}$$

La simulación es el proceso de resolver numéricamente el modelo en una computadora u otras herramientas (Li et al., 2020). Este proceso se realiza mediante el algoritmo mostrado en la Figura 2.1. Primeramente se asigna el valor inicial de las variables de estado. Se

dice que un sistema tiene un determinado número de grados de libertad igual al número de variables de estado. Esto es debido a que sus valores iniciales pueden ser elegidos inicialmente.

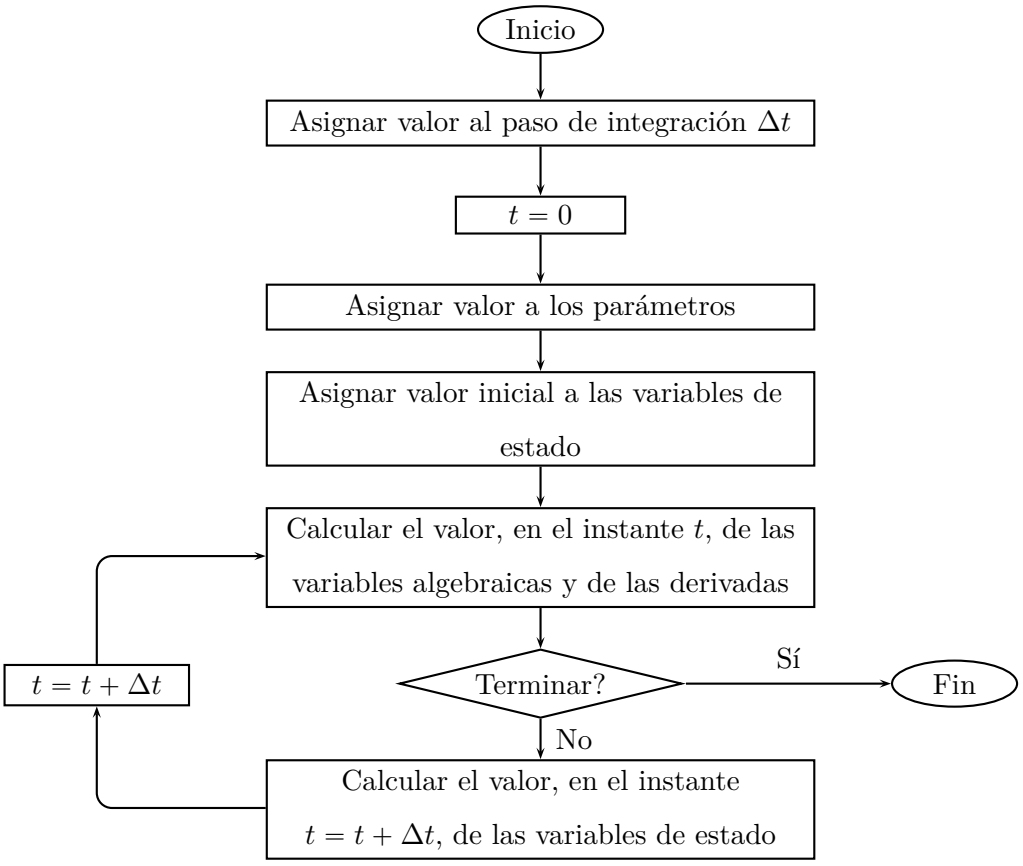


Figura 2.1: Algoritmo para la simulación de modelos matemáticos de tiempo continuo (Urquía y Martín, 2018).

Posteriormente se calcula el valor las variables algebraicas y las derivadas según la *causalidad computacional* asignada por un algoritmo de partición. Es decir, se ordenan las ecuaciones con las variables expresadas de manera explícita de manera que el sistema de ecuaciones pueda ser resuelto de manera secuencial. Es posible que aparezca en este paso un *lazo algebraico*, es decir, que no todas las variables puedan ser expresadas de manera explícita a partir de variables calculadas anteriormente. Este lazo algebraico implica la resolución de un sistema de ecuaciones, que, independientemente de si es o no lineal, se suele resolver mediante métodos numéricos.

El último paso consiste en calcular las variables de estado en el instante siguiente

mediante la integración numérica de sus derivadas. Por ejemplo, la función de paso del método explícito de Euler para la ecuación diferencial ordinaria $\dot{x} = f(t, x)$ es la siguiente,

$$x_{i+1} = x_i + \Delta \cdot f(x_i) \tag{2.4}$$

Sin embargo, los entornos para el modelado en Modelica incorporan métodos numéricos mucho más potentes, siendo DASSL un solucionador estándar para la resolución de sistemas DAE.

Los entornos *software* para el modelado y simulación de sistemas pueden agruparse en las dos siguientes categorías:

1. Entornos de simulación que permiten el modelado basado en diagramas de bloques. El caso más característico en este paradigma es el de Simulink, software de código propietario desarrollado por Mathworks.

2. Lenguajes de modelado orientado a objetos. Aunque existen diversos entornos y lenguajes, el caso más paradigmático es el del lenguaje Modelica.

### 2.1.2. El paradigma del modelado físico y el lenguaje Modelica

Modelica es un lenguaje de modelado multidominio, declarativo y orientado a objetos para el modelado orientado a componentes de sistemas complejos, por ejemplo, sistemas que contienen subcomponentes mecánicos, eléctricos, electrónicos, hidráulicos, térmicos, de control, de energía eléctrica u orientados a procesos. Aunque se puede modelar a través de instrucciones de código de programación, existen entornos gráficos como Dymola, software propietario, y OpenModelica, de código abierto, que permiten modelar sistemas dinámicos para su simulación y optimización de una manera sencilla.

El lenguaje de modelado Modelica es el fruto de un esfuerzo internacional de estandarización iniciado en 1996 para diseñar un lenguaje para el modelado físico orientado a objetos. Modelica se fundamenta en dos conceptos clave. Como se ha comentado, se trata de un modelado orientado a objetos y, por tanto, jerárquico, basado en clases y en su instanciación. Gracias a la herencia de clases, los componentes son fácilmente adaptables. Este enfoque de diseño facilita la reutilización del conocimiento de modelado y el intercambio de modelos y librerías. Por otro lado, se trata de un modelado no causal, en el que es posible introducir ecuaciones constitutivas y de balance en su forma natural como un sistema de ecuaciones diferenciales algebraicas. Modelica puede generar un código

de simulación eficiente a partir de estas ecuaciones. El modelado no causal supone una ventaja frente a los sistemas gráficos causales, ya que la descomposición de un sistema en estructuras de diagramas de bloques con interacciones causales supone un gran esfuerzo de análisis y es proclive al error.

Un modelo de Modelica consiste en un modelo compuesto que especifica la tipología del sistema modelado en términos de sus componentes y conexiones entre estos. Para el desarrollo de estos modelos se suele usar un editor gráfico en el que se puede dibujar un diagrama del modelo. La sintaxis en Modelica normalmente se encuentra oculta y los componentes estándar suelen estar disponibles en librerías de modelo. Una característica importante para la reutilización de modelos en Modelica es la capacidad para desarrollar modelos parciales. Estos modelos parciales son clases incompletas, es decir, que las ecuaciones constitutivas todavía no han sido añadidas a la clase. Esta característica permite el diseño de interfaces y la herencia entre clases. Es decir, a partir de estas clases parciales se pueden diseñar clases de componentes más específicas.

En la Figura 2.2 se muestra como ejemplo un modelo de un sistema de accionamiento de un motor, en el que intervienen los dominios eléctrico, mecánico y de control. Como se puede observar, se acoplan componentes de librerías preexistentes mediante conexiones que representan las mismas interfaces físicas de los sistemas, no relaciones causales. En la Figura 2.3 se muestra el modelo equivalente desarrollado en Simulink, en el cual el modelador del sistema debe establecer la causalidad computacional por sí mismo, ya que este tipo de modelado es causal. Por otro lado, este tipo de modelado no está orientado a objetos, por lo que la reutilización y el intercambio de los modelos es más limitada.

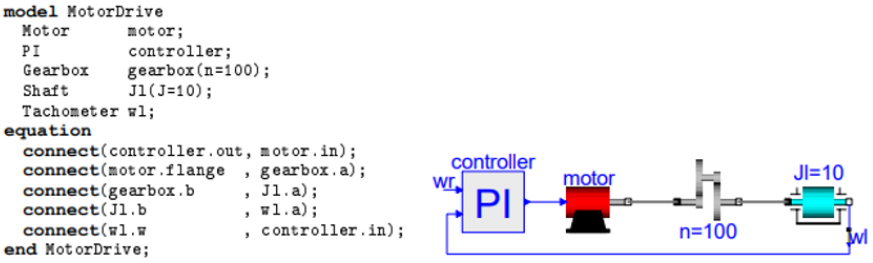


Figura 2.2: Modelo en Modelica de un sistema de accionamiento de motor, tanto en formato de código como de representación gráfica (Elmqvist et al., 1998).

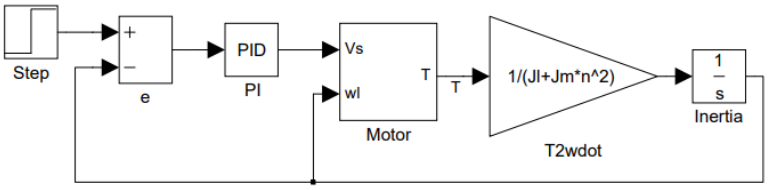


Figura 2.3: Modelo de diagrama de bloques de un accionamiento de motor desarrollado en Simulink (Elmqvist et al., 1998).

### 2.1.3. Intercambio de modelos y co-simulación

En el campo del modelado y la simulación, equipos multidisciplinares trabajan conjuntamente en proyectos de envergadura, que en multitud de ocasiones involucran el empleo de diferentes herramientas de análisis, en muchos casos incompatibles entre sí. Para la colaboración de estos equipos surgió el estándar FMI (del inglés, *Functional Mock-up Interface*), proyecto mantenido por la organización sin ánimo de lucro Modelica Association.

FMI es un estándar libre e independiente de la herramienta que permite el intercambio y la co-simulación de modelos de sistemas dinámicos en un formato estándar, específicamente en un archivo de extensión `.fmu` denominado FMU (del ingles, *Functional Mock-up Unit*). Estos archivos contienen en todo caso un *modelo de simulación*.

En la actualidad existen más de 200 herramientas software adheridas a este estándar, entre las cuales no sólo están entornos dedicados al modelado en lenguaje Modelica como OpenModelica o Dymola, también son compatibles con FMI herramientas como MATLAB Simulink, ANSYS CFX o Simcenter de Siemens, por nombrar algunos casos relevantes. En cuanto a bibliotecas Python de código abierto para inspeccionar, compilar y simular archivos FMU que se adhieren a este estándar, se encuentran algunas como FMPy, de Dassault Systèmes, o PyFMI, de Modelon, siendo esta última la empleada en este proyecto.

El estándar FMI soporta dos tipos de FMU, *co-simulación* (CS, del inglés *co-simulation*) e *intercambio de modelo* (ME, del inglés *model exchange*). Cuando una herramienta de simulación exporta un FMU de tipo CS, esta incorpora el solucionador numérico (*solver*) en el propio FMU (Figura 2.4). Por tanto, la herramienta que importa el FMU, puede emplearlo para obtener directamente una salida en el siguiente paso de integración después de haber proporcionado unas entradas. En cambio, en la simulación de un FMU de tipo ME, el solucionador es proporcionado por la herramienta importadora y el FMU ME tan sólo representa un sistema dinámico en ecuaciones diferenciales. En cualquier caso, el

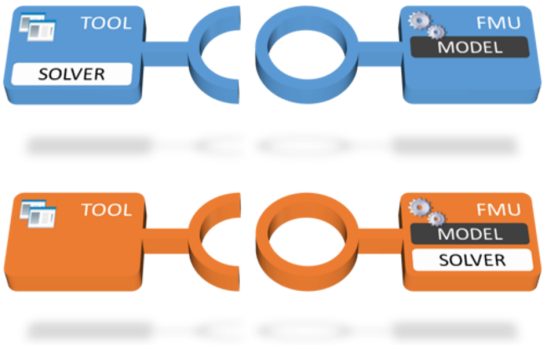


Figura 2.4: Existen dos tipos de *Functional Mock-up Units* (FMU): de co-simulación (CS) e intercambio de modelo (ME). Ambos encapsulan un modelo dinámico, pero únicamente el FMU de co-simulación (abajo, en naranja) incluye su propio solucionador numérico, mientras que el tipo ME (arriba, en azul) delega la integración en la herramienta de simulación que lo importa (Modelon, 2024).

FMU proporciona funciones para establecer el estado y las entradas, y para calcular las derivadas del estado. En este trabajo, sólo se implementará la carga de modelos FMU del tipo CS.

### 2.1.4. Control inteligente

La teoría de control convencional no cubre todas las necesidades para el desarrollo de controladores de sistemas dinámicos complejos, principalmente debido a la incertidumbre del entorno. Es necesaria una toma de decisiones similar a la humana, que implique el uso de razonamiento heurístico y aprendizaje a partir de experiencias pasadas.

Linkens y Nyongesa (1996) señalaron que, para desarrollar sistemas de control más flexibles, era preciso incorporar a la teoría de control convencional otros elementos tales como la lógica, el razonamiento y la heurística.

Es por estas limitaciones que, en la década de 1990, surge el campo del control inteligente, cuando diversos autores establecen su marco conceptual, aunque conceptos como redes neuronales, el modelo CMAC para aprendizaje adaptativo o la lógica difusa son anteriores. Según Werbos (1992), el control inteligente consiste en el uso de sistemas de control de propósito general que aprenden con el tiempo a alcanzar objetivos (u optimizar) en entornos complejos, ruidosos y no lineales cuya dinámica debe aprenderse en última instancia en tiempo real.

Este enfoque trata de abordar un problema de control a través de formas de representación y toma de decisiones inspiradas en sistemas humanos, animales o biológicos, y con frecuencia construidos de manera heurística, lo que resulta en un controlador no lineal, posiblemente adaptativo (Passino, 1993).

Diversos autores, como Linkens y Nyongesa (1996), agrupan las técnicas de control inteligente en tres grandes áreas: lógica difusa (*fuzzy logic*), neuro-control (técnicas basadas en redes neuronales) y métodos heurísticos de optimización tales como los algoritmos genéticos. También existe el control experto, es decir, el uso de sistemas expertos de inteligencia artificial clásicos para tareas de control. Sin embargo, estas técnicas no son prácticas debido al irrealizable número de reglas que requerirían.

**Neuro-control**

El neuro-control considera a un sistema de control como una correspondencia entre el estado de la planta y los comandos actuadores. Mediante el aprendizaje, este mapa se modifica para mejorar el objetivo de actuación del sistema de control. La principal ventaja del aprendizaje de las redes neuronales es su habilidad para ajustarse a sistemas dinámicos no lineales y mal modelados.

Ya en los años noventa comenzaron a desarrollarse aplicaciones pioneras de neuro-control en sistemas reales. Por ejemplo, KrishnaKumar et al. (1995) propusieron un controlador neuronal adaptativo para el control de actitud de la Estación Espacial Freedom, demostrando capacidades de control no lineal y adaptación frente a variaciones de inercia. En el ámbito biomédico, Chang et al. (1997) aplicaron redes neuronales para controlar el ángulo de la rodilla mediante estimulación eléctrica funcional, combinando un controlador neuronal con realimentación PID para mejorar el seguimiento de trayectorias. Asimismo, Cisneros y Leal (2001) estudiaron distintos esquemas de neuro-control, como modelos inversos directos y controladores basados en CMAC, para estabilizar manipuladores no lineales como el péndulo invertido y su versión doble. Estos trabajos ilustran cómo el neuro-control se exploró tempranamente en sistemas aeroespaciales, biomédicos y robóticos, sentando las bases de su desarrollo posterior.

Los métodos modernos de aprendizaje profundo por refuerzo, que emplean redes neuronales de manera efectiva, son mucho más recientes que el desarrollo de estas clasificaciones. Sin embargo, ya en 1992 se desarrolló el algoritmo TD-Gammon (Tesauro, 1994), que entrenaba una red neuronal artificial mediante una forma muy básica de aprendizaje por

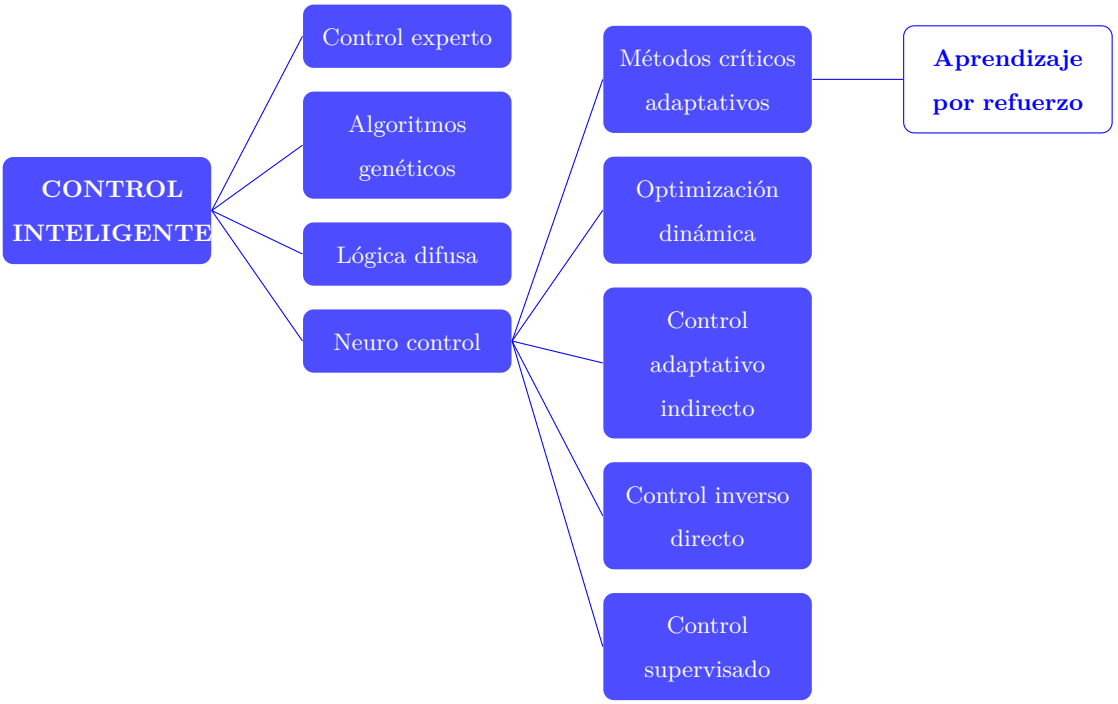


Figura 2.5: Clasificación de las diferentes técnicas de control inteligente y encuadre del aprendizaje por refuerzo en el campo del neuro-control.

refuerzo denominada aprendizaje por diferencia temporal. Es por eso que, ya en la década de 1990, Werbos (1992) identificaba al aprendizaje por refuerzo como una de las cinco categorías de neuro-control, como se ha representado en la Figura 2.5.

Gracias a los avances en las tecnologías de la computación, el control inteligente ha experimentado un gran crecimiento en la ultimas décadas, especialmente cualquier técnica basada en redes neuronales, al igual que ha ocurrido en cualquier otra área del aprendizaje automático.

El advenimiento de las técnicas modernas de aprendizaje por refuerzo no implica que otras técnicas más básicas de neuro-control hayan caído en desuso. Todavía hoy se dan numerosos casos de aplicación de neuro-control, sobre todo de manera híbrida con control convencional. Tal es el caso del trabajo de Sierra-García y Santos (2021), en el que añadieron al control convencional del ángulo de ataque de una turbina eólica flotante un estimador basado en redes neuronales para estimar la velocidad del viento efectiva en la turbina y predecir su valor futuro, las cuales están sujetas a condiciones ambientales extremas.

Desde sus inicios siempre ha existido cierta controversia en cuanto a las capacidades del

control inteligente para realizar tareas de control convencional, ya que el control inteligente no posee un marco teórico tan bien desarrollado como el control convencional.

Los controladores inteligentes basados en aprendizaje automático presentan un buen comportamiento incluso si el sistema es no lineal y desconocido. Su principal ventaja es que no necesitan de un conocimiento del modelo matemático de la planta, ya que el controlador actúa como una caja negra que aproxima el modelo basándose en los datos recogidos durante el entrenamiento. En cambio, es difícil garantizar el rendimiento y una convergencia rápida (Cano Lopes et al., 2018).

En cualquier caso, el control inteligente no viene, de momento, a reemplazar al control clásico. Existen casos concretos en los que la aplicación de técnicas de control inteligente, en sí mismas o acompañando a elementos de control clásico, pueden ser beneficiosas. Generalmente, se trata de situaciones de gran incertidumbre, con perturbaciones o modelos sujetos a una gran incertidumbre.

### 2.1.5. Aprendizaje por refuerzo

El *aprendizaje automático* es una de las ramas de la inteligencia artificial que se centra en desarrollar modelos y algoritmos que permiten a una computadora aprender a partir de datos y mejorar a través de experiencias pasadas sin ser explícitamente programada para cada tarea. El aprendizaje por refuerzo constituye un área separada de otros tipos de aprendizajes, como el supervisado y el no supervisado, al incluir características de ambos (Figura 2.6). Cuando cualquiera de estas técnicas emplea redes neuronales, se le añade la etiqueta de *profundo*, o más frecuentemente, su correspondiente término inglés, *deep*.

El aprendizaje por refuerzo (RL, del término inglés *Reinforcement Learning*), una de las áreas de investigación en inteligencia artificial más activas, es un enfoque computacional del aprendizaje en el cual un agente trata de maximizar la recompensa que recibe de la interacción con un entorno incierto. La obra de Sutton y Barto (2018) se ha convertido en un referente en esta área del aprendizaje automático, de la cual se extraerán algunas ideas clave en este apartado necesarias para entender los algoritmos del estado del arte que se emplearán en este trabajo.

Los problemas de aprendizaje por refuerzo se idealizan en una formalización matemática clásica de toma de decisiones secuencial, llamada proceso de decisión de Markov (MDP, del inglés, *Markov Decision Problem*). En este marco matemático, el sujeto que aprende y toma decisiones se denomina el *agente*. El agente interacciona con su *entorno*, de manera

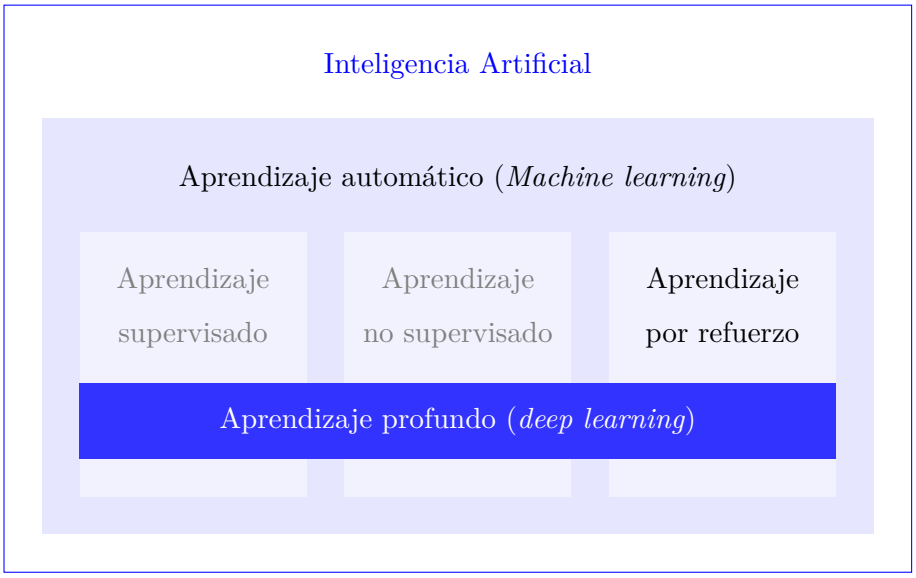


Figura 2.6: El aprendizaje por refuerzo constituye un área en sí misma dentro del aprendizaje automático.

que, al seleccionar una acción, este recibe no sólo una representación del nuevo estado, sino también una recompensa. Esta interacción se representa en la Figura 2.7.

El objetivo del agente es maximizar la recompensa total que recibe en el largo plazo.

En cada instante de tiempo $t$, el agente recibe una representación del estado del entorno, $S_t$, en base a la cual selecciona una acción $A_t$. En el siguiente instante de tiempo, el agente recibe una recompensa numérica, $R_{t+1}$ y se encuentra en el nuevo estado, $S_{t+1}$.

Además de la existencia de un agente y de un entorno, los tres elementos comunes a cualquier algoritmo de aprendizaje por refuerzo son los siguientes:

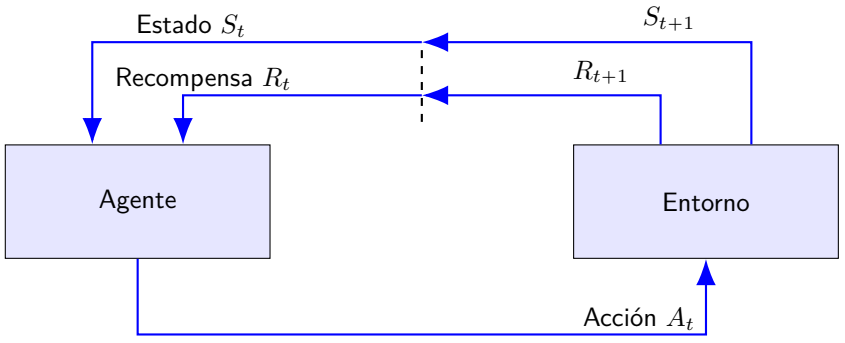


Figura 2.7: Interacción entre agente y entorno en un proceso de decisión de Markov. Adaptado de Sutton y Barto (2018).

1. Una *política*, que define la manera de comportarse del agente en cualquier instante, al determinar su acción en base a las observaciones que percibe del entorno. En general, una política será estocástica, es decir, especifica probabilidades para cada acción. Formalmente, es una correspondencia de un estado a una probabilidad $\pi(a \mid s)$ de seleccionar cada acción posible.

2. Una *señal de recompensa*, que define la meta de cualquier problema de aprendizaje por refuerzo.

3. Una *función valor*, que determina la meta a largo plazo. Es una predicción de la recompensa dado un estado.

Un estado $s$ es una descripción completa del sistema. En cambio, una observación $o$ es una descripción parcial del estado, es decir, omite información. En este trabajo, por lo general, hablaremos de estado, que será la información proporcionada por las variables de estado. El agente de aprendizaje por refuerzo tendrá acceso a la descripción completa del sistema dinámico, que viene determinado por el conjunto de sus variables de estado.

La recompensa es una señal que recibe el agente del entorno en cada intervalo de tiempo, después de cada acción. Esta es simplemente un número, $R_t \in \mathbb{R}$. La meta del agente es maximizar la recompensa que recibe, no inmediata sino acumulada a largo plazo

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.5}$$

donde $\gamma$ es un parámetro, $0 \leqslant \gamma \leqslant 1$, conocido como tasa de descuento, el cual determina el valor presente de las recompensas futuras.

La *función valor* de un estado $s$ bajo un política $\pi$ es el retorno esperado de decidir una acción según $\pi$ estando en $s$. Se define como

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[ G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \text{ para todo } s \in \mathcal{S} \tag{2.6}$$

La *función valor de acción* es usada por la mayoría de algoritmos de aprendizaje por refuerzo.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi \left[ G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right], \text{ para todo } s \in \mathcal{S} \tag{2.7}$$

Resolver un problema de aprendizaje por refuerzo implica encontrar una política que maximice la recompensa a largo plazo. Una *política óptima* es aquella que es mejor o igual a todas las demás para todos los estados. Así, una política $\pi$ es mejor que otra $\pi'$ ($\pi \geqslant \pi'$) si y sólo si $v_\pi(s) \geqslant v_{\pi'}(s)$ para todo $s \in \mathcal{S}$.

La *función valor de estado óptima*, $v_*$, se define como

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ para todo } s \in \mathcal{S} \tag{2.8}$$

De la misma manera, la *función valor de acción óptima*, $q_*$, se define como

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \text{ para todo } s \in \mathcal{S} \text{ y } a \in \mathcal{A}(s) \tag{2.9}$$

Se puede expresar $q_*$ en función de $v_*$ de la siguiente manera:

$$q_*(s,a) = \mathbb{E}_\pi \left[ R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a \right] \tag{2.10}$$

**Ecuaciones de Bellman**

Ambas funciones se pueden expresar mediante recursividad, propiedad fundamental para los algoritmos de aprendizaje por refuerzo a la hora de estimar tanto $v_\pi$ como $q_\pi$. Así, descomponiendo el retorno en el instante actual (Ecuación 2.5) en la recompensa inmediata más el retorno en el instante siguiente descontado al presente se obtiene la *ecuación de Bellman para* $v_\pi$:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[ G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[ R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[ r + \gamma \mathbb{E}_\pi \left[ G_{t+1} \mid S_{t+1} = s' \right] \right] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[ r + \gamma v_\pi(s') \right], \text{ para todo } s \in \mathcal{S}, \end{aligned}$$

que expresa el valor de un estado en función del valor de estados sucesivos, y que viene dada por las probabilidades tanto de la política $\pi(a \mid s)$ (si es estocástica) como de las transiciones entre estados $p(s', r \mid s, a)$ (si el entorno es estocástico).

De manera análoga, se puede encontrar la *ecuación de Bellman para* $q_\pi$:

$$q_\pi(s,a) \doteq \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s',a')\right], \text{ para todo } s \in \mathcal{S} \tag{2.11}$$

A través de las ecuaciones anteriores, se puede llegar a las ecuaciones de optimalidad de Bellman. La ecuación de optimalidad de Bellman para $v_*$ expresa que el valor de un estado bajo una política óptima es igual al retorno esperado de la mejor acción en ese estado. A continuación se muestran dos formas equivalentes de esta ecuación:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \qquad (2.12) \\ &= \max_a \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma v_*(s')\right] \qquad (2.13) \end{aligned}$$

De manera análoga, la ecuación de optimalidad de Bellman para la función de acción-valor $q_*$ puede expresarse de las dos formas siguientes:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \qquad (2.14) \\ &= \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right] \qquad (2.15) \end{aligned}$$

Estas expresiones para $q_*$ son especialmente relevantes porque constituyen la base teórica de los algoritmos de control en aprendizaje por refuerzo. En particular, el algoritmo *Q-learning* se fundamenta en la ecuación de optimalidad de Bellman para $q_*$, aproximándola de manera iterativa mediante actualizaciones de valores de acción estimados.

**Algoritmo *Q-learning* (Aprendizaje-Q)**

Uno de los grandes avances en el aprendizaje por refuerzo fue el desarrollo de un algoritmo de control de TD fuera de política (*off-policy*) conocido como *Q-learning*, presentado originalmente por Watkins (1989).

El agente actualiza en cada paso del episodio la función valor de acción estimada $Q$, tomando como objetivo la recompensa recibida en la transición al siguiente estado $S_{t+1}$ más el valor máximo de $Q$, descontado al instante actual, para todas las acciones posibles en el siguiente estado, como se muestra a continuación:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right], \tag{2.16}$$

donde $\alpha$ representa la tasa de aprendizaje.

La política objetivo de Q-learning es codiciosa (*greedy*) con respecto a las estimaciones actuales del valor de acción. La actualización en el aprendizaje de Q sólo mejora sobre la acción codiciosa, ignorando los resultados de las acciones exploratorias.

Posteriormente, Watkins y Dayan (1992) demostraron rigurosamente que, bajo ciertas condiciones, la función valor de acción aprendida $Q$ converge a la función valor de acción óptima $q_*$. Esta actualización se basa en la ecuación de optimalidad de Bellman para los valores de acción.

Cabe destacar que Q-learning es un método tabular, ya que representa el valor $Q$ de cada acción en cada estado mediante tablas, lo que limita su aplicación a espacios de acción discretos.

## 2.2. Estado del Arte

La revisión del Estado del Arte comenzará con un resumen de los principales algoritmos modernos de aprendizaje por refuerzo, mostrando su desarrollo histórico y las ventajas que cada algoritmo ha ido aportando. También se realizará un recorrido por las principales bibliotecas existentes relacionadas con el aprendizaje profundo, que de no existir complicarían enormemente el desarrollo de una aplicación como la aquí presentada. A continuación se mostrarán algunos casos de éxito recientes en la aplicación del RL a problemas de control reales en diferentes sistemas. Por último, se revisarán algunos casos de aplicaciones con objetivos similares a los que aquí se pretenden alcanzar, integrando el aprendizaje por refuerzo en entornos de simulación multidominio, tales como la aplicación ModelicaGym, que se tomó como punto de partida en el desarrollo de este software.

### 2.2.1. Algoritmos modernos de aprendizaje por refuerzo profundo (Deep RL)

En la mayoría de tareas, el espacio de estados es combinatorio y enorme. Por tanto, debido a límite de recursos de tiempo y memoria, encontrar una política óptima o una función valor óptima mediante métodos tabulares como *Q-learning* sería imposible (Sutton

y Barto, 2018). En la práctica, los modernos algoritmos de aprendizaje por refuerzo son los llamados métodos de solución aproximada, que se mostrarán en este apartado.

El problema de los métodos tabulares no sólo es una cuestión de tiempo y memoria, sino también que existen muchos estados nunca vistos. Por tanto, para tomar decisiones en dichos estados, el agente necesita de la *generalización* a partir de situaciones similares. La generalización no es posible con una tabla sino gracias a la *aproximación de funciones*, la cual, con sólo unos ejemplos tomados de la función deseada, construye una aproximación de la función completa. La aproximación de funciones es, por tanto, un tipo de *aprendizaje supervisado*. Cualquier método empleado en este campo, como es el caso de las redes neuronales artificiales o del ajuste de curvas estadístico, podría ser usado como método de aproximación de función.

**El algoritmo Deep Q-Network (DQN)**

Probablemente el avance más significativo en este campo en los últimos años fue el desarrollo del algoritmo DQN (*Deep Q-network*) por Mnih et al. (2015), quienes consiguieron por primera vez estimar la función acción-valor óptima (Ecuación 2.14) empleando redes neuronales convolucionales profundas como aproximadores de función.

Como demostraron sus autores, el agente artificial DQN alcanzó un rendimiento comparable al de un jugador profesional y superó a cualquier algoritmo previo al interactuar con videojuegos clásicos de Atari 2600. Todo ello sin conocer el funcionamiento interno de los juegos (*modelo*) y utilizando únicamente como entradas los píxeles sin procesar (*observaciones*) y el resultado del juego (*recompensa*). Además, demostró una notable robustez al emplear los mismos hiperparámetros en todos los juegos.

Antes de la introducción de DQN, se pensaba que el aprendizaje de funciones valor mediante aproximadores no lineales era difícil e inestable (Lillicrap et al., 2016). DQN consiguió hacerlo de manera estable y robusta gracias a dos innovaciones clave: (i) el entrenamiento *fuera de política* utilizando muestras almacenadas en una memoria de trayectorias o experiencias (*experience replay buffer*), lo que minimiza las correlaciones entre muestras, y (ii) el uso de una red *target* para proporcionar objetivos consistentes en las actualizaciones (sólo periódicamente) de los valores de acción $Q$ por diferencias temporales, mitigando así la inestabilidad y reduciendo las correlaciones con el objetivo. Estas dos ideas se han consolidado como elementos fundamentales en la mayoría de algoritmos modernos de aprendizaje por refuerzo.

Por otro lado, aunque DQN resolvió de manera efectiva problemas con espacios de observación multidimensionales, únicamente puede manejar espacios de acción discretos y de baja dimensionalidad. En consecuencia, su aplicabilidad resulta limitada en problemas de control de sistemas dinámicos con espacios de acción continuos, como los que se abordan en este trabajo.

**Clasificación de los algoritmos RL modernos**

El primer nivel en la clasificación de los algoritmos de aprendizaje por refuerzo modernos viene determinado por el acceso por parte del agente a un modelo del entorno. Un modelo del entorno es simplemente una función que predice las transiciones de estados y las recompensas. Por tanto, se tienen algoritmos *sin modelo* (conocidos como *model-free*) y *basados en modelos* (conocidos como *model-based*).

En los primeros, el agente carece de un modelo y tiene que aprender a partir de la experiencia. El aprendizaje de modelos es fundamentalmente difícil, por lo que incluso el esfuerzo intenso (estar dispuesto a dedicar mucho tiempo y realizar cálculos) puede no dar frutos. En los segundos, el agente puede planificar con anticipación, viendo qué sucedería para conjuntos de posibles acciones, y explícitamente decidir entre estas. Sin embargo, no se dispone normalmente de un modelo real del entorno. Los métodos basados en modelos se basan en la planificación como su componente principal, mientras que los métodos sin modelo se basan principalmente en el aprendizaje. Los métodos *model-free* son más populares y han sido desarrollados y probados más ampliamente que los *model-based* (OpenAI, 2018).

Es preciso aclarar, en el contexto de este trabajo, que, aunque se cuenta con un modelo en lenguage Modelica que permite simular la respuesta del sistema a una acción de control, el agente no tendrá acceso a este modelo, el modelo será como una caja negra para este, de la cual aprenderá por un mecanismo de prueba y error. El uso de un modelo en este caso surge de la necesidad de realizar experimentos simulados, ya que no se cuenta con un sistema físico, sino tan sólo un modelo matemático. Por tanto, todos los algoritmos que se emplearán en este trabajo, estarán dentro de la clasificación de algoritmos *sin modelo*.

En OpenAI (2018) se presenta una clasificación adecuada para la taxonomía de los algoritmos modernos de aprendizaje por refuerzo. En la Figura 2.8 se muestran los principales métodos *model-free*, omitiéndose los basados en modelos por no ser relevantes en este trabajo. Los algoritmos sin modelo se dividen en dos grandes familias según su estrategia

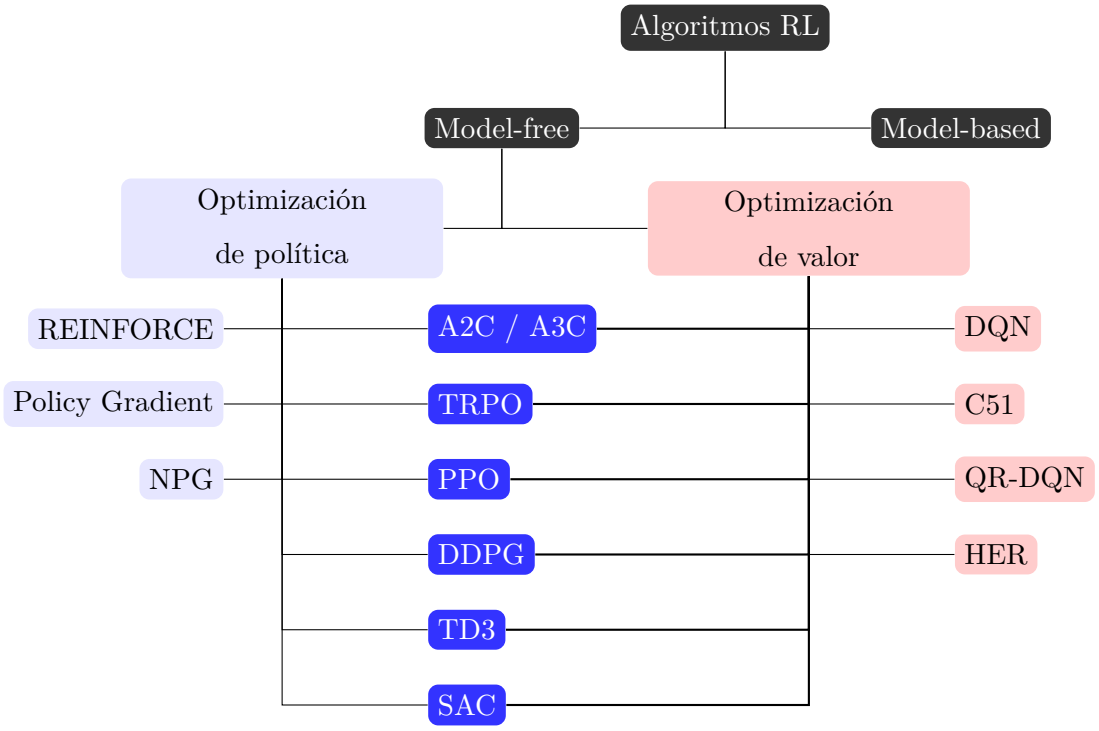


Figura 2.8: Taxonomía de los algoritmos RL *sin modelo*, adaptado de OpenAI (2018).

de aprendizaje: por un lado, los métodos *basados en el valor*, derivados del Q-learning, y por otro, los métodos de *optimización directa de política*.

Los métodos *basados en el valor* (o métodos de *acción-valor*) emplean durante el aprendizaje un aproximador $Q_\theta(s, a)$ para estimar la función acción-valor óptima $q_*(s, a)$. Suelen realizar la optimización fuera de política (*off-policy*), lo que significa que cada actualización puede usar datos recolectados en cualquier punto del entrenamiento, independientemente de cómo el agente eligió explorar el entorno en ese momento. La política se deriva directamente de esta función, seleccionando siempre las acciones con mayor valor estimado:

$$a(s) = \arg\max_a Q_\theta(s, a). \tag{2.17}$$

En otras palabras, primero se aprende el valor de cada acción en cada estado y, a partir de ahí, se seleccionan las acciones que maximizan dicho valor. Ejemplos clásicos de este enfoque son Q-learning y sus extensiones como DQN, C51 o QR-DQN.

Por otro lado, los métodos de *optimización de política* parametrizan explícitamente una política $\pi_\theta(a \mid s)$, que se entrena ajustando directamente los parámetros $\theta$ para maximizar el retorno esperado. Estos algoritmos suelen trabajar *on-policy*, ya que cada actualización se basa en experiencias generadas con la versión más reciente de la política. Este enfo-

que es más flexible al permitir políticas estocásticas y puede manejar espacios de acción continuos de manera natural. Métodos como REINFORCE o Policy Gradient clásico son representativos de esta categoría.

Ambos enfoques presentan ventajas y limitaciones complementarias: los métodos basados en el valor son eficientes en el uso de datos, mientras que los basados en política son más adecuados para entornos continuos y de alta dimensionalidad. Para combinar estas virtudes, surgen los algoritmos *actor-crítico*, que integran:

- un *actor*, encargado de parametrizar y optimizar la política (heredando las ventajas de los métodos de política directa),
- y un *crítico*, que estima funciones de valor (como en los métodos basados en valor) para guiar la actualización del actor.

Esta arquitectura híbrida, que se detalla en el siguiente apartado, ha demostrado ser especialmente potente y es la base de la mayoría de algoritmos de aprendizaje por refuerzo modernos (A2C/A3C, DDPG, TD3, SAC, entre otros).

**Algoritmos actor-crítico**

Una desventaja de los métodos exclusivamente de actor es que los estimadores de gradiente pueden tener una gran varianza. Por otro lado, los métodos de crítico, basados únicamente en la aproximación de funciones valor, pueden carecer de garantías sólidas sobre la optimalidad de la política resultante.

Los métodos actor-crítico combinan las ventajas de ambos enfoques: el *actor* representa la política y se actualiza en la dirección de un gradiente estimado, mientras que el *crítico* utiliza una arquitectura de aproximación (habitualmente basada en TD learning) y simulación para aprender una función valor, proporcionando así la señal que guía la actualización de la política. Como señalan Konda y Tsitsiklis (1999), siempre que se formulen sobre gradientes, estos algoritmos pueden analizarse como métodos de gradiente estocástico en los parámetros de la política, con propiedades de convergencia bajo ciertas condiciones. Además, ofrecen la promesa de una convergencia más rápida que los métodos exclusivamente de actor, gracias a la reducción de la varianza.

Una expresión clave para el gradiente del rendimiento promedio $J(\theta)$ de una política parametrizada es:

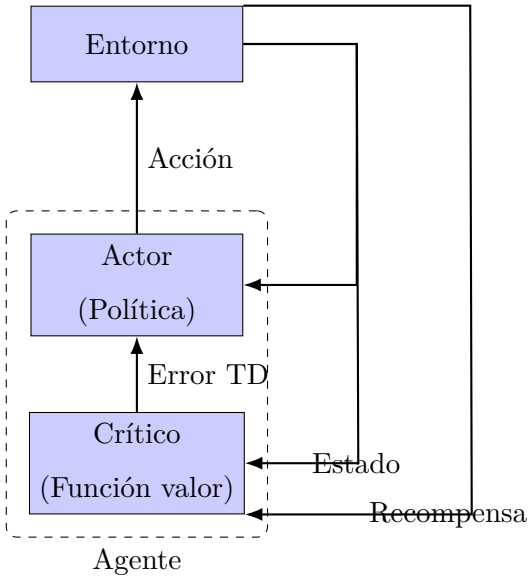


Figura 2.9: Arquitectura actor-crítico (basada en Diederichs, 2019).

$$\nabla_\theta J(\theta) \;=\; \sum_{x\in\mathcal{S}} \sum_{u\in\mathcal{A}} \eta_\theta(x,u)\, q_\theta(x,u)\, \psi_\theta(x,u), \tag{2.18}$$

donde $\eta_\theta(x,u)$ es la distribución estacionaria de pares estado-acción bajo la política $\pi_\theta$, $q_\theta(x,u)$ es la función acción-valor diferencial, y $\psi_\theta(x,u) = \nabla_\theta \log \pi_\theta(x,u)$ es el gradiente de la política respecto a los parámetros.

Este resultado establece la base teórica de los algoritmos actor-crítico: el crítico aproxima $q_\theta(x,u)$ o su proyección en un subespacio adecuado, y el actor ajusta sus parámetros en la dirección sugerida por la Ecuación 2.18.

En este trabajo se aplicará específicamente el algoritmo DDPG, que pertenece a esta familia. Aunque no es el más reciente ni el más eficiente, muestra una buena capacidad de aprendizaje en problemas de control continuo, como se evidenciará más adelante.

### Deep Deterministic Policy Gradient (DDPG)

El algoritmo DDPG (*Deep Deterministic Policy Gradient*) es un método *actor-crítico*, libre de modelo y fuera de política, diseñado para aprender políticas en espacios de acción continuos multidimensionales (Lillicrap et al., 2016). Se fundamenta en los algoritmos de gradiente de política determinista (*Deterministic Policy Gradient*, DPG) desarrollados por Silver et al. (2014), quienes demostraron que, a diferencia del gradiente de política estocástico, el gradiente determinista sólo requiere integrar sobre el espacio de estados, lo

que lo hace más eficiente en espacios de acción de gran dimensión.

La idea central de DDPG es mantener dos redes neuronales: un *actor* $\mu(s|\theta^\mu)$ que representa la política determinista, y un *crítico* $Q(s,a|\theta^Q)$ que estima la función acción-valor asociada. El actor se actualiza siguiendo el teorema del gradiente de política determinista:

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_{s\sim\rho^\beta}\Big[\nabla_a Q(s,a|\theta^Q)\big|_{a=\mu(s)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)\Big], \tag{2.19}$$

donde $\rho^\beta$ denota la distribución de estados inducida por una política de comportamiento $\beta$, lo que permite el aprendizaje *off-policy*.

Inspirado en el éxito de DQN, DDPG incorpora dos mecanismos clave para estabilizar el aprendizaje con aproximadores no lineales: (i) una memoria de experiencias (*replay buffer*) para reducir correlaciones entre muestras, y (ii) redes objetivo (*target networks*) que actualizan sus parámetros lentamente, proporcionando objetivos más estables en las actualizaciones de diferencias temporales (Lillicrap et al., 2016).

Gracias a estas características, DDPG ha demostrado un rendimiento competitivo en tareas complejas de control continuo, desde manipuladores robóticos hasta locomoción, incluso aprendiendo directamente a partir de observaciones en píxeles.

### 2.2.2. Bibliotecas de aprendizaje por refuerzo

Programar un algoritmo de aprendizaje por refuerzo moderno, es decir, basado en redes neuronales, desde cero, en cualquier lenguaje de programación y sin usar ninguna biblioteca de aprendizaje profundo sería un tarea titánica.

Los métodos RL toman un gran impulso gracias a las mejoras en computación y al desarrollo de bibliotecas de aprendizaje profundo (*deep learning*), es decir, aprendizaje automático basado en redes neuronales. Existen varias bibliotecas de aprendizaje automático profundo. Sin embargo, son dos las que dominan: TensorFlow y PyTorch. Este trabajo eludirá cualquier comparativa entre estas y se centrará en el empleo de TensorFlow, ya que su compatibilidad es un requisito como se especificará en el Capítulo 3.

TensorFlow es un sistema de aprendizaje automático basado en redes neuronales desarrollado por Google, inicialmente como software propietario para uso interno, pero liberado como software de código abierto en 2015. Es relativamente factible desarrollar un algoritmo RL moderno como puede ser DQN empleando únicamente un sistema de aprendizaje automático profundo. Sin embargo, a medida que la complejidad del algoritmo se incrementa, como es el caso de los algoritmos actor-crítico, esta tarea se hace más y más complicada.

Es por ello que han surgido numerosas bibliotecas que facilitan la tarea de implementar un algoritmo RL. A través de estas bibliotecas el usuario dispone de una serie de algoritmos ya implementados (juntos con sus respectivas redes neuronales). El usuario sólo se debe preocupar de ajustar los hiperparámetros, tales como la estructura de las redes neuronales, la tasa de aprendizaje de estas, su función de pérdida, o la tasa de descuento, que suelen ser comunes a todo algoritmo RL.

Del mismo modo que existen diferentes bibliotecas de aprendizaje profundo, existen bibliotecas para el aprendizaje por refuerzo, de desarrollo muy reciente. Como los algoritmos de aprendizaje por refuerzo modernos aproximan las funciones de valor mediante redes neuronales, estas bibliotecas se construyen sobre las bibliotecas de *deep learning* mencionadas anteriormente. En nuestro caso, debido al requisito de compatibilidad con TensorFlow, en este trabajo se empleará únicamente TensorFlow Agents (TF-Agents).

Es importante señalar que también existen entornos de simulación que incorporan herramientas para el aprendizaje por refuerzo. Es el caso de Matlab/Simulink que en su Reinforcement Learning Toolbox incorporan los métodos DQN, PPO, SAC y DDPG.

**TensorFlow *Agents*.** TF-Agents es una biblioteca de aprendizaje por refuerzo desarrollada por Google Research sobre TensorFlow, diseñada para ofrecer una API coherente y extensible para construir, entrenar y evaluar agentes RL. A diferencia de otras bibliotecas populares (como `Stable Baselines3` o `SKRL`), que se apoyan directamente en la API de `OpenAI Gym` —un marco abierto propuesto por OpenAI para estandarizar la interfaz de entornos RL— y `Gymnasium` —su sucesor mantenido por la comunidad, compatible con versiones anteriores y con mejoras en estabilidad— para la definición de entornos, TF-Agents implementa su propia abstracción de entornos, basada en las clases `PyEnvironment` (entorno en Python puro) y `TFPyEnvironment` (entorno integrado con TensorFlow). Esta independencia de `Gym` permite definir entornos de forma más explícita, flexible y nativa para TensorFlow, sin depender de versiones específicas de `Gym`/`Gymnasium` ni de sus cambios de API. Este aspecto es especialmente relevante para el presente trabajo, dado que, como se verá, aplicaciones similares a la aquí desarrollada dependen fuertemente de la API de `Gym`/`Gymnasium`.

TF-Agents adopta una arquitectura modular, en la que se distinguen claramente los componentes clave de cualquier experimento RL: entornos (`environments`), políticas (`policies`), agentes (`agents`), generadores de experiencia (`drivers`), búferes de repetición (`replay`

Tabla 2.1: Algoritmos RL implementados por TF-Agents.

| | DQN | DDPG | C51 | TD3 | PPO | REINFORCE | SAC |
|---|---|---|---|---|---|---|---|
| Discreto | ✓ | | ✓ | | ✓ | ✓ | |
| Continuo | | ✓ | | ✓ | ✓ | | ✓ |

`buffers`) y métricas de evaluación. Esta separación de responsabilidades, junto con utilidades avanzadas como *wrappers*, *observers* y *metrics*, permite construir flujos de entrenamiento reproducibles, fácilmente extensibles y compatibles con despliegue en producción.

Hasta la fecha, TF-Agents implementa algoritmos representativos de todas las familias modernas de RL, incluyendo métodos basados en valor (DQN, C51), optimización de políticas (PPO, REINFORCE) y arquitecturas actor-crítico fuera de política (DDPG, SAC, TD3), abarcando tanto espacios de acción discretos como continuos (Tabla 2.1). Esta cobertura resulta especialmente adecuada para el control de sistemas dinámicos modelados en Modelica, donde predominan espacios de acción continuos y multidimensionales. Es posible usar métodos de acción discreta, como DQN (véase Apartado 2.2.1), con problemas continuos a través de la discretización de los espacios de acción continuos; sin embargo, el aprendizaje sólo es eficiente en modelos con una única acción de control.

Gracias a su estrecha integración con TensorFlow, los modelos de política entrenados pueden exportarse de manera nativa al formato TensorFlow Lite, lo que facilita su despliegue en aplicaciones embebidas o en entornos de simulación compatibles.

### 2.2.3. Aplicaciones del RL al control de sistemas dinámicos

El aprendizaje por refuerzo (RL) ha sido aplicado con éxito al control de sistemas dinámicos en diversos campos, tales como la robótica, turbinas eólicas individuales y parques eólicos.

#### Robótica

El RL profundo permite desarrollar controladores autónomos sin necesidad de modelado explícito. Haarnoja et al. (2018) aplicaron con éxito el algoritmo SAC para entrenar en el mundo real un robot cuadrúpedo Minitaur, sin entrenamiento previo en simulación. Cano Lopes et al. (2018) demostraron el uso de PPO para controlar un cuadrirrotor con buena capacidad de generalización, usando los simuladores V-REF y Vortex.

**Control de turbinas eólicas**

El RL se ha aplicado al control del ángulo de *pitch* y del generador en turbinas eólicas, sistemas que presentan gran complejidad dinámica. Sierra-García y Santos (2020) diseñaron un controlador de *pitch* basado en RL que superó el rendimiento de un controlador PID en simulaciones. Fernandez-Gauna et al. (2022) aplicaron un algoritmo actor-crítico sobre OpenFAST para optimizar el control conjunto del *pitch* y del par del generador, logrando una mejora del 22 % en el error medio de potencia. Por su parte, Tomin et al. (2019) utilizaron el algoritmo TRPO para desarrollar un controlador MIMO aplicado a una turbina de referencia de 5 MW.

**Control de parques eólicos**

El control cooperativo de turbinas en parques eólicos ha demostrado beneficios significativos frente a estrategias *greedy* convencionales. Dong et al. (2021) propusieron un enfoque basado en DDPG con regularización de recompensa, validado con el simulador SOWFA, logrando hasta un 15 % más de generación. Dong y Zhao (2022) propusieron el algoritmo CER-RL para mejorar la eficiencia de aprendizaje. Xie et al. (2022) introdujeron DN-DDPG, con doble red neuronal para manejar señales de control incompatibles, validado en WFSim con mejoras frente a DDPG y estrategias convencionales.

### 2.2.4. Entrenamiento en entornos simulados

La mayoría de los estudios anteriores entrenaron los agentes en simuladores específicos del dominio (como OpenFAST o V-REF), que permiten simulaciones precisas pero carecen de flexibilidad para representar sistemas complejos multidominio. Este trabajo no busca sustituir dichas herramientas, sino ofrecer una alternativa basada en entornos como Modelica, que permiten un modelado físico orientado a objetos con componentes de múltiples dominios y compatible con el estándar FMI.

La propuesta de este trabajo consiste en una interfaz entre TensorFlow y Modelica, que permitirá aplicar técnicas de RL directamente sobre modelos físicos complejos, con mayor generalidad y escalabilidad. Esto abre la posibilidad de entrenar controladores inteligentes en sistemas Modelica y, potencialmente, integrarlos con subsistemas desarrollados en herramientas externas como Simulink u OpenFAST.

### 2.2.5. Casos de integración de agentes de aprendizaje por refuerzo con entornos de simulación multidominio

El único caso de extensión de un entorno de simulación para capacidades de aprendizaje por refuerzo, a conocimiento del autor, se ha dado en el software propietario MATLAB, con su respectiva caja de herramientas. En el ámbito del lenguaje Modelica, ni Dymola ni OpenModelica cuentan con ninguna característica similar. Tal vez por ello, algunos investigadores han desarrollado software de código abierto basado en Python que permite de alguna manera entrenar modelos desarrollados en lenguaje Modelica en entornos de aprendizaje por refuerzo para realizar tareas de control. Este es el caso de las bibliotecas ModelicaGym, DymolaGym y FMUGym, todos proyectos de código abierto con repositorios públicos.

#### MATLAB Reinforcement Learning Toolbox™

La *Reinforcement Learning Toolbox™* de MATLAB es una herramienta comercial que proporciona una interfaz gráfica, funciones y bloques de Simulink para entrenar políticas mediante algoritmos de aprendizaje por refuerzo como DQN, PPO, SAC y DDPG. Esta caja de herramientas permite diseñar, entrenar y desplegar controladores inteligentes y estrategias de toma de decisiones en aplicaciones complejas, incluyendo robótica, sistemas autónomos y problemas de optimización de recursos.

#### ModelicaGym

ModelicaGym (Lukianykhin y Bogodorova, 2019) fue la primera biblioteca de código abierto que formalizó una interfaz estandarizada entre modelos de Modelica exportados como FMU y el entorno OpenAI Gym (Figura 2.10). Desarrollada en Python, ModelicaGym permite que modelos dinámicos complejos se integren de forma modular con agentes de RL, reduciendo la necesidad de programar lógica específica para cada entorno. Su arquitectura define una jerarquía clara de clases (`ModelicaBaseEnv`, `ModelicaCSEnv`, etc.) que implementan la API estándar de Gym y encapsulan la interacción con la FMU mediante la biblioteca `PyFMI`. Esto facilita crear nuevos entornos especificando únicamente parámetros del modelo, variables de entrada/salida y políticas de recompensa, sin modificar el núcleo de la herramienta.

La principal innovación de ModelicaGym radica en ofrecer un conector genérico entre

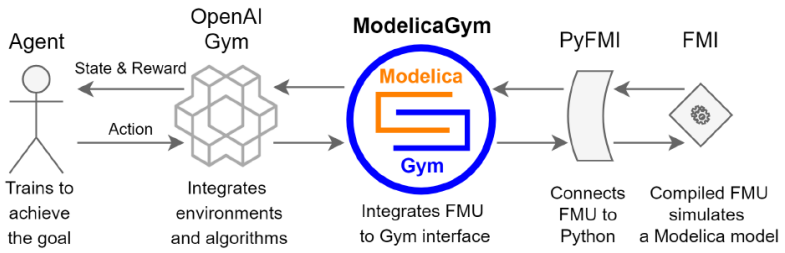


Figura 2.10: Diagrama de alto nivel del flujo de trabajo con la herramienta ModelicaGym (Lukianykhin y Bogodorova, 2019).

Gym y FMI, cubriendo un vacío señalado en trabajos previos en los que la integración RL–Modelica se realizaba de forma ad hoc y limitada a modelos concretos. Para validar la propuesta, los autores implementaron un caso de estudio basado en el sistema clásico carro–péndulo invertido, exportado desde JModelica.org y entrenado mediante Q-learning. El artículo incluye experimentos que analizan el impacto de parámetros físicos (relación de masas, magnitud de la fuerza) y de configuración (paso de simulación, recompensas) sobre el aprendizaje, destacando el valor de la herramienta como banco de pruebas reproducible.

A pesar de su potencial, ModelicaGym presenta varias limitaciones: solo soporta FMU en modo de co-simulación y depende de librerías que ya no se mantienen activamente (JModelica.org, OpenAI Gym), lo que ha dificultado su adopción y evolución. No obstante, este proyecto sentó las bases conceptuales para bibliotecas posteriores como DymolaGym y FMUGym, y ha servido de inspiración para trabajos como el presente.

### DymolaGym

DymolaGym (Pigott et al., 2022) es otro esfuerzo en la integración de Modelica con entornos de aprendizaje por refuerzo, similar a ModelicaGym, pero con algunas diferencias clave. DymolaGym también conecta con OpenAI Gym, pero, al contrario que en el caso de ModelicaGym, en este trabajo no se usaron ficheros FMU para modelar el comportamiento del entorno. En su lugar, se empleó un método para su uso con la interfaz de programación avanzada (API, del inglés *Advanced Programming Interface*) basada en Python de Dymola, junto con modelos de sistemas de potencia construidos mediante la biblioteca OpenIPSL de Modelica. Al simular los modelos con la API del compilador en lugar de FMUs, se garantiza que el modelo sea tan preciso como sea posible (se mantienen las relaciones acausales) y se aumenta el rendimiento.

**FMUGym**

FMUGym (Wrede et al., 2024) es una biblioteca de Python, de código abierto y con licencia MIT, desarrollada por el *Fraunhofer Institute for Integrated Circuits* (Fraunhofer IIS, Alemania). Este proyecto introduce mejoras significativas respecto a ModelicaGym, especialmente en términos de flexibilidad, compatibilidad con bibliotecas modernas de aprendizaje por refuerzo y capacidad para manejar incertidumbres durante el entrenamiento.

FMUGym está diseñado para integrarse directamente con bibliotecas actuales como Stable-Baselines3 (SB3) y SKRL, facilitando el uso de algoritmos avanzados como SAC y HER. Además, está orientado explícitamente a la evaluación de controladores bajo condiciones inciertas, permitiendo la inyección de perturbaciones y variaciones de parámetros en las dinámicas del sistema, lo que fomenta el desarrollo de políticas de control robustas.

A diferencia de ModelicaGym, la interfaz con el estándar FMI se realiza mediante la biblioteca FMPy (Figura 2.11). Otro cambio significativo es la migración de OpenAI Gym (ya discontinuado) a Gymnasium. El proyecto se publicó en 2024 en el congreso EG-ICE, acompañado de casos de estudio aplicados a sistemas no lineales y climatización (HVAC), y mantiene un repositorio público y activo en GitHub[1].

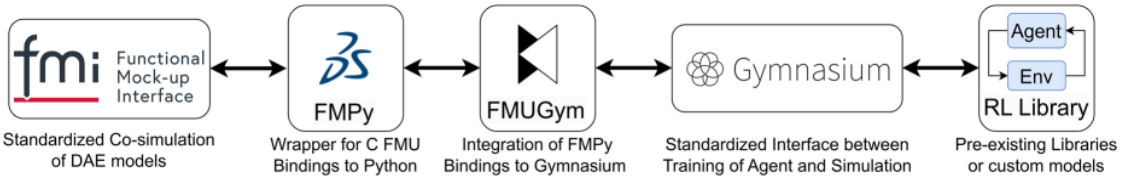


Figura 2.11: Diagrama de alto nivel del flujo de trabajo con la herramienta FMUGym (Wrede et al., 2024).

Como se observa en la Tabla 2.2, los esfuerzos previos para la integración RL–Modelica/FMI han sido valiosos, pero presentan carencias significativas. ModelicaGym y DymolaGym introdujeron por primera vez conectores genéricos entre Modelica y entornos Gym, pero ambos proyectos se encuentran descontinuados, dependen de bibliotecas ya obsoletas (como OpenAI Gym) y ofrecen una integración limitada con ecosistemas modernos de aprendizaje profundo. FMUGym, en cambio, es el único proyecto activo y en evolución, con soporte actualizado para Gymnasium y compatibilidad con bibliotecas de RL basadas

[1] https://github.com/Fraunhofer-IIS/fmugym

en PyTorch como Stable-Baselines3 (SB3) y SKRL, lo que lo hace atractivo para experimentación rápida. Sin embargo, ninguno de estos marcos aborda de forma simultánea un flujo estandarizado con TF-Agents y PyFMI ni un *pipeline* completo de despliegue sobre Modelica (TFLite/SMArtInt).

Tabla 2.2: Comparativa de marcos RL+Modelica/FMI en la literatura.

| **Criterio** | **ModelicaGym** (Lukianykhin y Bogodorova, 2019) | **DymolaGym** (Pigott et al., 2022) | **FMUGym** (Wrede et al., 2024) |
|---|---|---|---|
| Soporte FMU | ✓ (CS) | ✗ (API Dymola; sin FMU) | ✓ |
| Librerías RL soportadas | Genérico vía Gym (ej.: Q-learning) | Genérico vía Gym | SB3, SKRL (PyTorch) |
| Compatibilidad Gym/Gymnasium | OpenAI Gym (discontinuado) | OpenAI Gym (discontinuado) | Gymnasium |
| Integración con TF-Agents | ✗ | ✗ | ✗ |
| Backend FMI en Python | PyFMI | – | FMPy |
| Pipeline TFLite/`SMArtInt` | ✗ | ✗ | ✗ |
| Estado del proyecto | Sin mantenimiento | Sin mantenimiento | Activo (2024) |

Este panorama evidencia una oportunidad clara para el desarrollo de un marco abierto que cubra estas limitaciones, ofreciendo integración fluida con tecnologías modernas de RL, una base sólida para experimentación científica y una orientación práctica hacia la transferencia de controladores inteligentes a entornos de simulación y producción. En el Cap. 4 se presentará `RLIC-FMU` y se repondrá la comparativa incluyendo la propuesta de este trabajo.

# Capítulo 3

# Objetivos

Para la elaboración de los objetivos de este Trabajo Fin de Máster se trabajó estrechamente con la compañía alemana XRG Simulation GmbH, miembro de *Modelica Association*. Esta compañía propuso la temática de este trabajo, colaboró en la definición del objetivo general y de los específicos, y realizó un seguimiento periódico del desarrollo.

## 3.1. Objetivo general

El objetivo principal de este Trabajo Fin de Máster es diseñar, desarrollar y demostrar un módulo software que permita entrenar y validar controladores inteligentes mediante técnicas de aprendizaje por refuerzo (*Reinforcement Learning*, RL), aplicados a modelos de sistemas físicos desarrollados en Modelica. La herramienta deberá ser capaz de integrarse en entornos de modelado y simulación multidominio, con el propósito de facilitar el diseño y despliegue de soluciones de control automático y generación de trayectorias óptimas en aplicaciones industriales complejas.

## 3.2. Objetivos específicos

Para la consecución del objetivo general, se plantean los siguientes objetivos específicos, definidos en colaboración con XRG Simulation GmbH:

1. Investigar los fundamentos teóricos del aprendizaje por refuerzo y analizar los algoritmos más relevantes del estado del arte en el contexto del control de sistemas físicos.

2. Analizar y aplicar la biblioteca *Agents* de TensorFlow, explorando sus funcionalidades a través de ejemplos prácticos y tutoriales. Esta elección se fundamenta en su compatibilidad con SMArtInt (*Simple Modelica Artificial Intelligence Interface Library*), una herramienta desarrollada por XRG que permite integrar modelos de inteligencia artificial entrenados en TensorFlow dentro de entornos de simulación basados en Modelica, con soporte tanto para Dymola como para OpenModelica.

3. Revisar el estado del arte de entornos ya existentes que combinen aprendizaje por refuerzo y Modelica (por ejemplo, ModelicaGym, desarrollado por Lukianykhin y Bogodorova (2019)), evaluando las soluciones disponibles y seleccionando la más adecuada como punto de partida para el desarrollo.

4. Desarrollar un entorno en Python que permita el entrenamiento de agentes de aprendizaje por refuerzo sobre modelos de sistemas físicos exportados como FMU desde Modelica.

5. Implementar casos de prueba de control, como el péndulo invertido sobre carro (Figura 4.5), incluyendo la exportación del modelo TensorFlow del agente RL para demostrar su uso directo en una simulación en Modelica a traves de SMArtInt.

6. Diseñar y evaluar controles de ejemplo para modelos de sistemas más complejos, proporcionados por XRG Simulation GmbH, que sirvan como pruebas de validación y benchmarking del enfoque propuesto.

7. Identificar posibles limitaciones del método (por ejemplo, en términos de velocidad de entrenamiento o escalabilidad) y proponer estrategias de mejora que puedan ser implementadas en fases posteriores.

## 3.3. Metodología del trabajo

La metodología que se seguirá en este Trabajo Fin de Máster se estructura en varias fases interrelacionadas, diseñadas para alcanzar los objetivos planteados en colaboración con XRG Simulation GmbH:

1. **Revisión bibliográfica.** Se llevará a cabo un análisis exhaustivo de la literatura sobre técnicas de aprendizaje por refuerzo, con especial énfasis en algoritmos de interés para el control de sistemas físicos. Asimismo, se revisarán casos de éxito

de aplicación de control inteligente en la industria, identificando ventajas frente a métodos convencionales.

2. **Estudio de entornos existentes.** Se analizarán soluciones previas que integran aprendizaje por refuerzo y Modelica, como ModelicaGym, evaluando su aplicabilidad y seleccionando los enfoques más prometedores como punto de partida.

3. **Desarrollo software.** Se implementará un módulo en Python para entrenar y validar agentes de RL aplicados a modelos exportados como FMU desde Modelica. El desarrollo se llevará a cabo siguiendo una orientación a objetos, bajo un sistema de control de versiones Git (repositorio corporativo de XRG en GitLab). Las principales librerías que se utilizarán serán TensorFlow (y su módulo *Agents*) y herramientas auxiliares del ecosistema científico de Python (NumPy, Pandas, Matplotlib).

4. **Casos de estudio.** Se definirán y resolverán casos de control representativos. En primer lugar, se trabajará con un modelo de péndulo invertido sobre carro como ejemplo base. Asimismo, se ilustrará el desarrollo de un modelo en OpenModelica con fines didácticos, consistente en un depósito de líquido sometido a un control multivariable de nivel y temperatura. Posteriormente, se considerarán modelos más complejos proporcionados por XRG para realizar un benchmarking y validar la escalabilidad del enfoque.

5. **Entrenamiento y ajuste de hiperparámetros.** Una fase clave del trabajo consistirá en el entrenamiento de los agentes de RL, lo que implicará la definición y ajuste de hiperparámetros como tasas de aprendizaje, factores de descuento o tamaños de lote. Se explorarán diferentes configuraciones con el fin de optimizar la estabilidad y la velocidad de convergencia de los modelos. Asimismo, se diseñarán funciones de recompensa adecuadas para cada caso de estudio, teniendo en cuenta los objetivos de control específicos (por ejemplo, minimización del error en nivel y temperatura en el depósito, o estabilización del péndulo invertido). El proceso incluirá un análisis iterativo de los resultados, ajustando las estrategias de entrenamiento y las métricas de rendimiento hasta alcanzar un comportamiento satisfactorio de los agentes.

6. **Integración en entornos Modelica.** Se investigará la exportación de agentes entrenados en TensorFlow Lite y su integración en Dymola mediante la librería SMArtInt, desarrollada por XRG, con el fin de demostrar la interoperabilidad de la

solución propuesta.

7. **Revisión y validación continua.** A lo largo del proyecto se mantendrá una comunicación periódica con XRG Simulation GmbH (reuniones semanales y coordinación por correo electrónico), lo que permitirá alinear el desarrollo con las necesidades de la empresa y validar los avances de manera iterativa.

# Capítulo 4

# Desarrollo del trabajo

## 4.1. Desarrollo de la herramienta software `RLIC-FMU`

`RLIC-FMU` (*Reinforcement Learning for Intelligent Control of Functional Mock-up Units*) es un *framework* en Python que permite entrenar y evaluar agentes de aprendizaje por refuerzo sobre modelos físicos desarrollados en Modelica y exportados como FMU. La herramienta integra de forma nativa la librería `TF-Agents` y emplea `PyFMI`, desarrollada por Modelon, para la ejecución paso a paso de FMU. Además, proporciona utilidades para crear entornos de RL estandarizados, configurar algoritmos de control inteligente y automatizar el ciclo de entrenamiento y evaluación.

El software ha sido diseñado para facilitar la experimentación y comparación de estrategias de control, abstraer los detalles técnicos de TF-Agents y gestionar automáticamente la comunicación con las FMU mediante PyFMI.

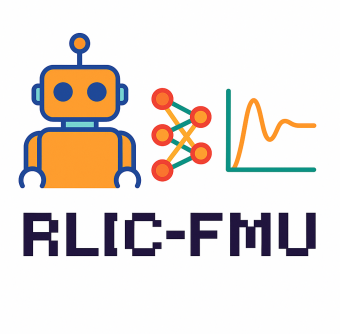


Figura 4.1: Logotipo de `RLIC-FMU`.

### 4.1.1. Identificación de requisitos

Los requisitos del desarrollo de la herramienta `RLIC-FMU` fueron identificados conjuntamente con la empresa XRG Simulation GmbH, considerando las necesidades de integrar algoritmos de aprendizaje por refuerzo con modelos dinámicos exportados como FMU. Los requerimientos clave son los siguientes:

1. Compatibilidad con cualquier modelo dinámico exportado en el estándar FMI (Modelica/FMU).
2. Capacidad de crear y entrenar agentes de aprendizaje por refuerzo mediante TensorFlow y `TF-Agents`.
3. Flexibilidad del entorno para su uso con distintos modelos de sistemas.
4. Documentación y ejemplos de uso claros que faciliten su adopción y extensión.

Estos requisitos dieron lugar a una arquitectura modular interna de `RLIC-FMU`, en la que se diferencian claramente las capas de agentes, núcleo de entrenamiento y entornos de simulación. Antes de describir en detalle esta arquitectura, la Figura 4.2 presenta una visión de alto nivel del flujo de trabajo de la herramienta en su ecosistema: su integración con `TF-Agents` para el aprendizaje por refuerzo, `PyFMI` para la ejecución de FMU, y modelos desarrollados en Modelica.

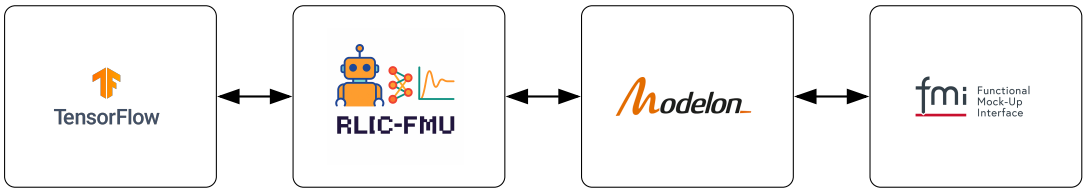


Figura 4.2: Diagrama de alto nivel del flujo de trabajo con la herramienta RLIC-FMU.

### 4.1.2. Arquitectura de alto nivel

La herramienta `RLIC-FMU` se estructura en tres bloques principales: Agents, Core y Environments, cuya interacción se resume en la Figura 4.3. Esta organización refleja la separación entre (i) los algoritmos de aprendizaje por refuerzo, (ii) la lógica central de entrenamiento y evaluación, y (iii) los entornos personalizados basados en modelos físicos exportados como FMU.

- **Agents**: capa que encapsula los algoritmos de RL proporcionados por TF-Agents. Incluye envoltorios ligeros para DQN, C51, DDPG, SAC y PPO, instanciados mediante `AgentSelector`. Todos comparten una API coherente que facilita cambiar de algoritmo sin modificar el resto del *pipeline*.

- **Core**: núcleo del framework, implementado en la clase `ExperimentRL`. Esta clase:

  - Instancia el agente y los entornos (entrenamiento y evaluación) a partir de `AgentSelector` y `EnvironmentFactory`.
  - Configura los *drivers* de recogida de experiencia (instancia clases como `DynamicStepDriver` o `DynamicEpisodeDriver`), el *replay buffer* y las métricas de evaluación.
  - Implementa el bucle de entrenamiento (`train`), con evaluación periódica, selección del mejor agente y representación gráfica en tiempo real.
  - Ofrece utilidades para ejecutar episodios de forma interactiva (métodos como `render_random_collection` o `render_evaluation`) y exportar políticas entrenadas mediante `PolicySaver`.

  Este bloque abstrae los detalles de TF-Agents, de modo que el usuario puede lanzar experimentos sin programar manualmente los bucles de entrenamiento o evaluación.

- **Environments**: implementa entornos compatibles con RL sobre modelos Modelica exportados como FMU. Se basan en la clase abstracta `ModelicaBaseEnvironment`, que gestiona la interacción paso a paso con la FMU mediante `FMUdriver`. Las clases `EnvironmentFactory` y `EnvironmentConfig` centralizan la configuración y aplicación de *wrappers* (p. ej., `TimeLimit`, `RunStats`). Los entornos concretos se definen en `custom_envs` (`CartPole`, `Watertank`, `Superheater`), especificando observaciones, acciones y recompensas.

El flujo sigue el ciclo estándar de RL: los Agents generan acciones que se aplican en los Environments; estos devuelven observaciones y recompensas, mientras el Core coordina el entrenamiento, almacenamiento de experiencias y evaluación. Esta modularidad permite añadir nuevos modelos o algoritmos manteniendo interfaces homogéneas y un *pipeline* de entrenamiento reutilizable.

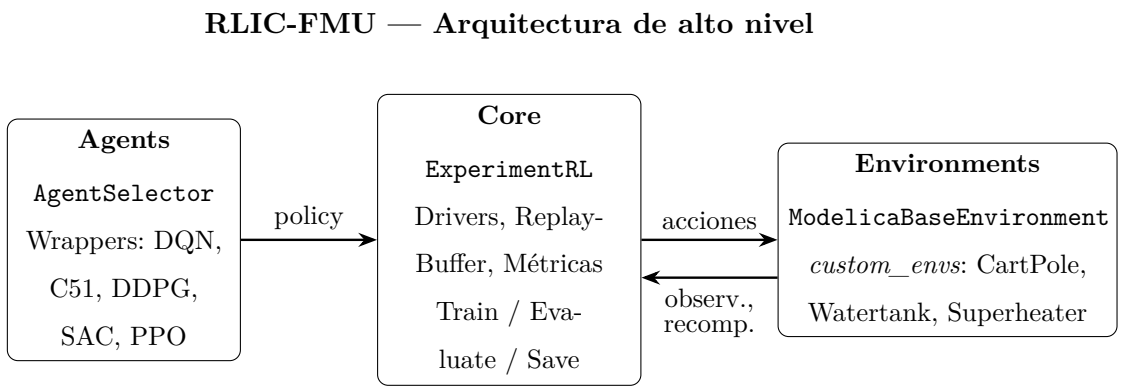


Figura 4.3: Arquitectura de alto nivel de `rlic_fmu`: el Core (`ExperimentRL`) orquesta el entrenamiento entre Agents y Environments.

### 4.1.3. Entorno personalizado (`rlic_fmu/environments`)

Un aspecto fundamental del software desarrollado es la definición de un entorno personalizado que permite aplicar algoritmos de RL sobre modelos físicos implementados en Modelica. En este contexto, se ha seguido la arquitectura de TF-Agents, que al igual que OpenAI Gym, proporciona una abstracción estandarizada para la interacción entre un agente de RL y un entorno (*environment*).

La clase `ModelicaBaseEnvironment` constituye el núcleo de esta implementación. Esta clase hereda directamente de la abstracción `py_environment.PyEnvironment` de TF-Agents, lo que garantiza la compatibilidad con cualquier agente de RL. Además, proporciona una capa intermedia de generalización que facilita la creación de entornos personalizados sin necesidad de reimplementar la lógica de interacción básica (métodos `step` y `reset`).

Las subclases específicas de `ModelicaBaseEnvironment`, contenidas en el módulo custom_envs, son las que se emplean para cada modelo concreto. En estas subclases se definen los elementos clave del entorno:

- Variables de observación y de acción.
- Especificaciones de entrada y salida (`_action_spec`, `_observation_spec`).
- Señales de referencia o *set-points*.
- Función de recompensa asociada al desempeño del agente.

De esta manera, el modelador solo necesita declarar las variables y parámetros rele-

vantes del modelo para contar con un entorno de RL plenamente operativo, sin tener que ocuparse de la infraestructura subyacente. La clase `FMUdriver` juega un papel esencial en este proceso, ya que se encarga de gestionar la comunicación con los modelos exportados como FMU, permitiendo al entorno ejecutar simulaciones, recopilar observaciones y aplicar acciones en cada paso de interacción.

La clase `ModelicaBaseEnvironment` implementa la lógica central de interacción reescribiendo los métodos protegidos `_reset()` y `_step()`, responsables de gestionar el ciclo de episodios y pasos en TF-Agents. En nuestra implementación, `_reset()` reinicia la FMU (`reset`, `setup_experiment`, `initialize`), fija los parámetros iniciales y la ventana temporal de simulación, recupera el estado inicial (incluyendo observaciones "custom"), y devuelve `ts.restart(...)`. Por su parte, `_step(action)` valida y aplica la acción a la FMU, actualiza los *set-points*, ejecuta la simulación en la ventana $[t_{\text{start}}, t_{\text{stop}}]$, apila observaciones y recompensas, normaliza con los límites de `_observation_spec`, comprueba el fin de episodio (p. ej., si alguna observación sale de rango) y retorna `ts.transition(...)` o `ts.termination(...)` según corresponda; además, avanza la ventana temporal incrementando $\tau$ para el siguiente paso.

La configuración de los entornos se centraliza en la clase `EnvironmentConfig`, que define parámetros como límites de acciones, normalización de observaciones o duración de episodios. A partir de esta configuración, la clase `EnvironmentFactory` no solo automatiza la creación de instancias de entornos personalizados, sino que también aplica una serie de *wrappers* que extienden sus funcionalidades (por ejemplo, `TimeLimit` para limitar el número de pasos o `RunStats` para recopilar estadísticas de ejecución). Un aspecto especialmente relevante es el uso de `TFPyEnvironment`, que permite convertir los entornos implementados en Python puro en entornos compatibles con TensorFlow, integrándolos directamente con los algoritmos de TF-Agents.

En resumen, la jerarquía basada en `ModelicaBaseEnvironment` y sus subclases en `custom_envs` ofrece una interfaz clara y flexible que enlaza los modelos físicos desarrollados en Modelica con los algoritmos de RL, facilitando la experimentación y la comparación entre diferentes configuraciones de control. Un aspecto destacado es la relación de *composición* entre `ModelicaBaseEnvironment` y `FMUdriver`: cada entorno posee y controla su propia instancia de `FMUdriver`, responsable de gestionar el ciclo de vida de la FMU asociada. La organización de estas clases se ilustra en el diagrama UML de la Figura 4.4.

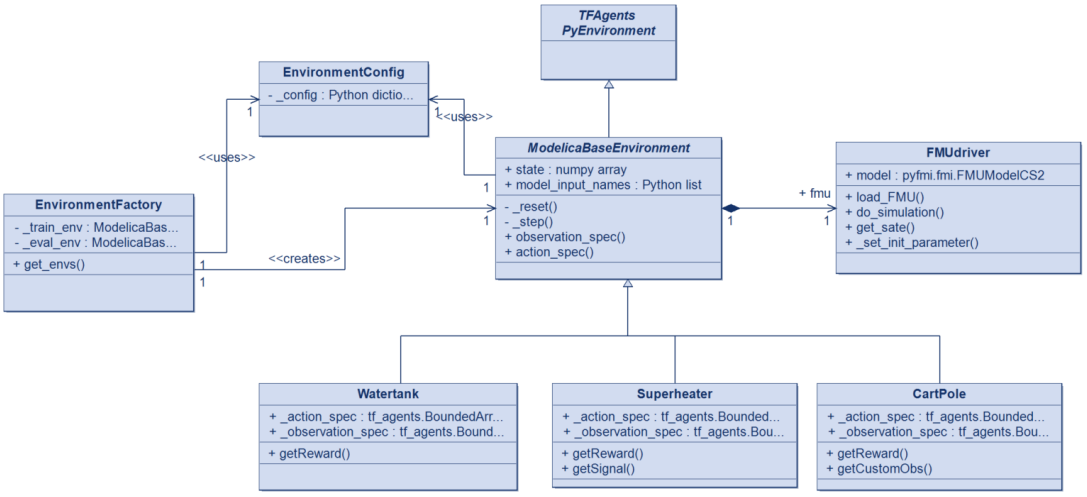


Figura 4.4: Diagrama UML de las clases principales que implementan los entornos personalizados, con `ModelicaBaseEnvironment` como clase base.

### 4.1.4. Agentes RL personalizados (`rlic_fmu/agents`)

El paquete `agents` proporciona envoltorios ligeros sobre los agentes de TF-Agents con el objetivo de simplificar su instanciación con hiperparámetros razonables por defecto y una interfaz homogénea para su uso con los entornos descritos en la Sección 4.1.3.

**Alcance en este trabajo.** Aunque el paquete incluye envoltorios para varios algoritmos, en este trabajo únicamente se han probado y validado DQN (para espacios de acción discretos) y DDPG (para acción continua). El resto de clases sirven como guía y punto de partida para trabajos futuros.

**Envoltorios disponibles.** Actualmente se incluyen los siguientes envoltorios (wrappers) que instancian internamente a los agentes equivalentes de TF-Agents:

- **DQN**: agente *Deep Q-Network* para acción discreta.
- **C51**: variante distribuida de DQN (*Categorical DQN*).
- **DDPG**: *Deep Deterministic Policy Gradient* para acción continua.
- **SAC**: *Soft Actor-Critic* para acción continua fuera de política.
- **PPO**: *Proximal Policy Optimization*, algoritmo basado en políticas con redes actor-crítico.

Cada envoltorio expone una configuración mínima (tamaños de red, tasas de aprendizaje, *target update* o *polyak*, tamaño del *replay buffer*, etc.) con valores por defecto razonables y compatibles con los entornos del paquete.

**API común de los agentes (por convención).** Actualmente no se define una interfaz explícita. En su lugar, los envoltorios exponen la API nativa de TF-Agents (p. ej., `agent.train()`, `agent.policy`, `agent.collect_policy`, utilidades de guardado/carga), lo que permite cambiar de algoritmo sin modificar el resto del *pipeline* (fábricas, bucles de entrenamiento, *drivers* de experiencia).

**`AgentSelector` (Factory).** El paquete `agents` incluye la clase `AgentSelector`, que a partir de una configuración con el campo `tf_agent_name` devuelve una instancia del agente correspondiente (`SAC`, `DQN`, `C51`, `DDPG` o `PPO`). De este modo: (i) se desacopla la elección del algoritmo del código de entrenamiento; (ii) se centraliza la configuración; y (iii) se mantiene una API estable aun si cambian detalles internos de cada envoltorio. Véase `RL_Agent.py`.

**Resumen de uso recomendado.** Para problemas con acción *discreta*, emplear `DQN` (o `C51` si se requiere una aproximación distribuida del retorno). Para acción *continua*, emplear `DDPG` (o `SAC` cuando se busque mayor robustez fuera de política). En este trabajo, los experimentos y resultados reportados se han realizado exclusivamente con `DQN` y `DDPG`.

### 4.1.5. Comparativa de `RLIC-FMU` con marcos RL–Modelica/FMI existentes

La Tabla 4.1 resume las principales características de los marcos existentes que combinan aprendizaje por refuerzo (RL) con modelos físicos exportados como FMU, previamente analizados en el Capítulo 2, e incorpora ahora la propuesta de este trabajo. Esta comparación evidencia las contribuciones clave de `RLIC-FMU`, como su integración nativa con TF-Agents, el uso explícito de PyFMI como backend FMI, y el soporte de un flujo completo de despliegue sobre Modelica mediante TFLite y SMArtInt.

Cabe destacar que proyectos como FMUGym presentan ventajas notables en otros ámbitos, como su enfoque explícito en control bajo incertidumbre y su compatibilidad con bibliotecas modernas de RL basadas en PyTorch (`Stable Baselines3`, `SKRL`), así como un desarrollo activo en los últimos años. La diferencia de objetivos explica esta diversidad

de funcionalidades: mientras FMUGym busca flexibilidad y robustez en entornos inciertos, `RLIC-FMU` responde a unos requisitos de diseño distintos, orientados a la integración profunda con TensorFlow y TF-Agents y a la exportación de modelos entrenados para su uso directo en entornos Modelica mediante SMArtInt. Esta complementariedad resalta el valor de ambas iniciativas en contextos de aplicación diferenciados.

Tabla 4.1: Comparativa de marcos RL+Modelica/FMI, incluyendo `RLIC-FMU`.

| **Criterio** | **ModelicaGym** (Lukianykhin y Bogodorova, 2019) | **DymolaGym** (Pigott et al., 2022) | **FMUGym** (Wrede et al., 2024) | **RLIC-FMU** (este trabajo) |
|---|---|---|---|---|
| Soporte FMU | ✓ (CS) | ✗ (API Dymola; sin FMU) | ✓ | ✓ (CS) |
| Librerías RL soportadas | Genérico vía Gym (ej.: Q-learning) | Genérico vía Gym | SB3, SKRL (PyTorch) | TF-Agents (TensorFlow) |
| Compatibilidad Gym/Gymnasium | OpenAI Gym (discontinuado) | OpenAI Gym (discontinuado) | Gymnasium | Wrapper propio; interoperable |
| Integración con TF-Agents | ✗ | ✗ | ✗ | ✓ |
| Backend FMI en Python | PyFMI | – | FMPy | PyFMI |
| Pipeline TFLite/SMArtInt | ✗ | ✗ | ✗ | ✓ |
| Estado del proyecto | Sin mantenimiento | Sin mantenimiento | Activo (2024) | Activo (2025) |

## 4.2. Modelos de sistemas físicos

En este trabajo se emplean tres modelos desarrollados en lenguaje Modelica, siguiendo el paradigma de *modelado físico* y la *orientación a objetos*, previamente introducidos en el Apartado 2.1.2. Estos modelos han sido exportados como FMU y se utilizarán como banco de pruebas para entrenar controladores inteligentes mediante técnicas de aprendizaje por refuerzo.

Los modelos se presentan en orden creciente de complejidad:

- El primero es el célebre carro con péndulo invertido, ampliamente utilizado como *benchmark* en la validación de algoritmos de control y que constituye un caso de referencia clásico.
- El segundo corresponde a un tanque de líquido con control multivariable de nivel

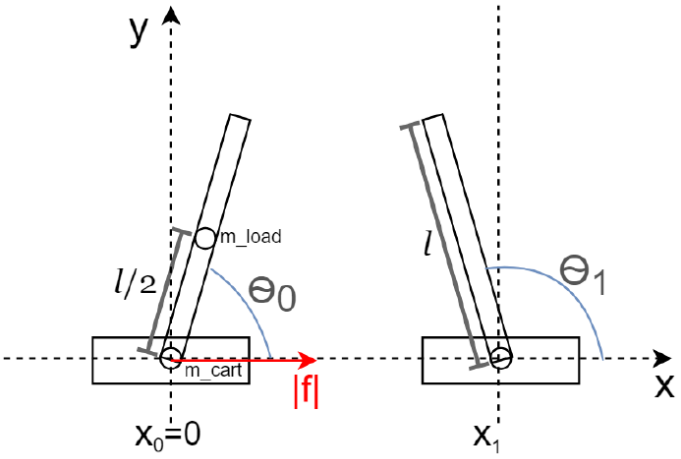


Figura 4.5: Sistema de carro con péndulo invertido (Lukianykhin y Bogodorova, 2019).

y temperatura, que introduce un mayor grado de acoplamiento entre entradas y salidas.

- Finalmente, se considera una línea de sobrecalentamiento de vapor (*superheater line*), un caso de control industrial realista proporcionado por XRG Simulation GmbH y desarrollado en Dymola.

En las subsecciones siguientes se ofrece una breve descripción de cada uno de estos modelos, destacando sus características principales y los retos de control que plantean.

### 4.2.1. Sistema de carro con péndulo invertido

La tarea del carro y el péndulo invertido (*cart-pole*) constituye uno de los problemas de referencia más utilizados tanto en la teoría de control como en el aprendizaje por refuerzo (Lundberg y Barton, 2010). Su interés radica en que, siendo un sistema mecánico relativamente sencillo, reproduce de manera clara los retos asociados al control de sistemas inestables, como es el caso del lanzamiento de cohetes o la estabilización de robots bípedos.

El modelo consiste en un carro de masa $M$ que se desplaza sin fricción sobre un raíl horizontal, sobre el cual se articula un péndulo rígido de masa $m$ y longitud $l$. El carro está sometido a una fuerza externa $F$ que constituye la variable de control, mientras que la dinámica del sistema queda determinada por el desplazamiento horizontal del carro $x$ y el ángulo del péndulo $\theta$ con respecto a la vertical. El diagrama esquemático de este sistema físico se representa en la Figura 4.5.

El modelo matemático que describe la evolución temporal del sistema puede obtenerse

mediante la segunda ley de Newton o mediante las ecuaciones de Lagrange. En forma no lineal, las ecuaciones diferenciales resultantes son:

$$\begin{aligned} (M+m)\ddot{x} - ml\ddot{\theta}\cos\theta + ml\dot{\theta}^2\sin\theta &= F \\ l\ddot{\theta} - g\sin\theta &= \ddot{x}\cos\theta \end{aligned} \tag{4.1}$$

donde $g$ representa la aceleración de la gravedad. Este sistema de ecuaciones acopladas refleja la fuerte interacción entre el movimiento del carro y la dinámica del péndulo.

La dificultad principal del problema reside en mantener el péndulo en posición vertical inestable ($\theta \approx 0$). Para el diseño de controladores, es habitual linealizar las ecuaciones en torno a este punto de equilibrio, obteniendo un modelo lineal que conserva la inestabilidad del sistema pero permite aplicar técnicas de control clásicas (por ejemplo, colocación de polos o LQR). No obstante, la versión no lineal resulta especialmente interesante en el contexto de aprendizaje por refuerzo, ya que obliga a los algoritmos a explorar estrategias más generales de control, incluyendo la fase de *swing-up* (llevar el péndulo desde la posición estable hacia abajo hasta la vertical inestable).

El sistema *cart-pole* ha sido ampliamente adoptado como *benchmark* en la literatura, tanto en métodos de control óptimo y robusto, como en el campo del aprendizaje por refuerzo profundo, donde se emplea en entornos clásicos como `CartPole-v1` de OpenAI Gym. Su sencillez conceptual, junto con la riqueza de los comportamientos dinámicos que exhibe, lo convierten en un caso ideal para validar algoritmos de control adaptativo e inteligencia artificial.

En la Figura 4.6 se muestra el modelo desarrollado en lenguaje Modelica. Este modelo se utilizará como primera prueba de concepto para la conexión entre simulación física (FMU exportada desde Modelica) y algoritmos de aprendizaje por refuerzo implementados en TensorFlow Agents. Como se puede observar, el modelo cuenta con una única entrada que representa la fuerza aplicada sobre el carro, y con cuatro salidas: la posición del carro, su velocidad, el ángulo $\varphi$ del péndulo respecto a la horizontal y su velocidad angular $\omega$. La entrada se considerará como la acción en el contexto de RL, mientras que las cuatro variables de salida constituirán el vector de observaciones definido en el entorno de aprendizaje.

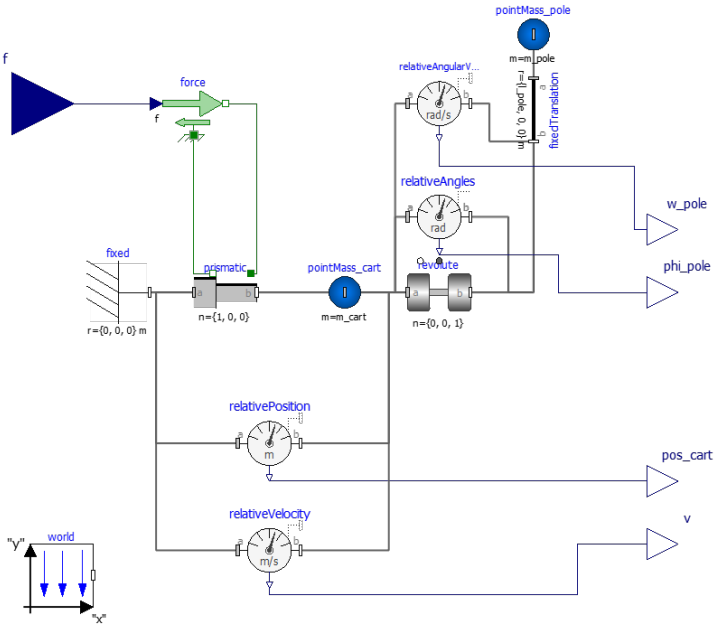


Figura 4.6: Diagrama del modelo del sistema de carro con péndulo invertido, desarrollado en OpenModelica a partir de componentes del paquete `Mechanics.Multibody` de la *Modelica Standard Library (MSL)*.

### 4.2.2. Depósito de líquido

El modelo del depósito de líquido con control de nivel y temperatura introduce un mayor grado de dificultad al incorporar dos acciones de control simultáneas. Se trata, por tanto, de un problema multivariable en el que existen dos lazos de control que influyen el uno sobre el otro. Además, este sistema se empleará como ejemplo ilustrativo del proceso de modelado de un sistema físico en lenguaje Modelica.

El sistema, cuyo esquema se muestra en la Figura 4.7, consiste en un depósito en el que el fluido entra por la parte superior mediante una bomba y sale por un orificio en la base debido a la acción de la gravedad. La bomba está gobernada por un controlador de nivel (LC, del inglés *level controller*), que lee el nivel $h$ mediante un sensor y ajusta el voltaje $u_h$ de la bomba para alcanzar el valor de consigna $h_{ref}$. Por otro lado, un elemento calefactor situado en el depósito permite regular la temperatura del fluido hasta el valor de referencia $T_{ref}$. La temperatura es controlada mediante un controlador de temperatura (TC, del inglés *temperature controller*), que envía un voltaje $u_T$ al calefactor en función de la medición de $T$. Ambos controladores son de tipo proporcional-integral (PI).

El flujo de salida $F_{out}$ se obtiene en función del nivel del líquido $h$ como:

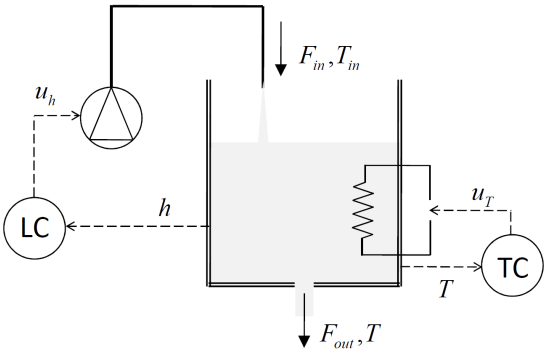


Figura 4.7: Depósito de líquido en el que dos variables, nivel $h$ y temperatura $T$, son controladas mediante controladores PI (Urquía y Martín, 2018).

$$F_{out} = a \cdot \rho \cdot \sqrt{2 \cdot g \cdot h} \tag{4.2}$$

El flujo de entrada $F_{in}$ está regulado por la bomba y depende de la señal de control $u_h$:

$$F_{in} = \text{máx}(0, k_f \cdot u_h), \tag{4.3}$$

donde $k_f$ es el factor de proporcionalidad entre el voltaje aplicado y el caudal generado. Obsérvese que la fuente únicamente puede inyectar fluido, pero no extraerlo.

El balance de masa en el depósito queda descrito por:

$$\frac{dm}{dt} = F_{in} - F_{out} \tag{4.4}$$

$$m = \rho \cdot A \cdot h \tag{4.5}$$

y la dinámica térmica por el balance de energía:

$$H = m \cdot C_p \cdot T \tag{4.6}$$

$$C_p = C_{p,0} + C_{p,1} \cdot T, \tag{4.7}$$

donde $C_p$ representa la capacidad calorífica específica del fluido.

En la simulación del sistema se definen además las señales de referencia de nivel y temperatura, que representan los valores de consigna $h_{ref}$ y $T_{ref}$ a los que deben responder

los controladores. Dichas señales se han configurado como funciones escalonadas en el tiempo a lo largo de un horizonte de 1000 segundos, de acuerdo con las expresiones:

$$h_{ref}(t) = \begin{cases} 5\,\text{m} & \text{si} \quad t < 300\,\text{s} \\ 3\,\text{m} & \text{si} \quad 300 \leqslant t < 600\,\text{s} \\ 7\,\text{m} & \text{si} \quad t \geqslant 600\,\text{s} \end{cases} \tag{4.8}$$

$$T_{ref}(t) = \begin{cases} 340\,\text{K} & \text{si} \quad t < 500\,\text{s} \\ 320\,\text{K} & \text{si} \quad t \geqslant 500\,\text{s} \end{cases} \tag{4.9}$$

Estos perfiles de referencia permiten analizar la capacidad del sistema de control para seguir consignas variables tanto en el nivel como en la temperatura del fluido.

En resumen, la dinámica del sistema queda representada mediante un sistema de ecuaciones diferenciales algebraicas (DAE) que combina el balance de masa, el balance de energía y las relaciones de flujo de entrada y salida. En conjunto, las ecuaciones son:

$$\frac{dm}{dt} = F_{in} - F_{out} \tag{4.10}$$

$$m = \rho \cdot A \cdot h \tag{4.11}$$

$$F_{out} = a \cdot \rho \cdot \sqrt{2 \cdot g \cdot h} \tag{4.12}$$

$$F_{in} = \text{máx}(0, k_f \cdot u_h) \tag{4.13}$$

$$H = m \cdot C_p \cdot T \tag{4.14}$$

$$C_p = C_{p,0} + C_{p,1} \cdot T \tag{4.15}$$

$$h_{ref} = f_h(t) \tag{4.16}$$

$$T_{ref} = f_T(t) \tag{4.17}$$

donde $f_h(t)$ y $f_T(t)$ representan las señales de referencia temporales. Este conjunto de expresiones describe completamente la evolución del nivel y la temperatura del fluido en el tanque, así como las consignas a seguir durante la simulación.

Los resultados de la simulación en OpenModelica, realizada a partir del código mostrado en el Listado A.1, se presentan en las Figuras 4.8–4.10, donde se observa cómo los controladores PI son capaces de alcanzar los valores deseados en cada uno de los intervalos definidos. Cabe destacar la interacción existente entre los lazos de control de temperatura y nivel: una variación brusca en el nivel del tanque introduce una perturbación térmica significativa que debe ser corregida para devolver la temperatura a su valor de consigna.

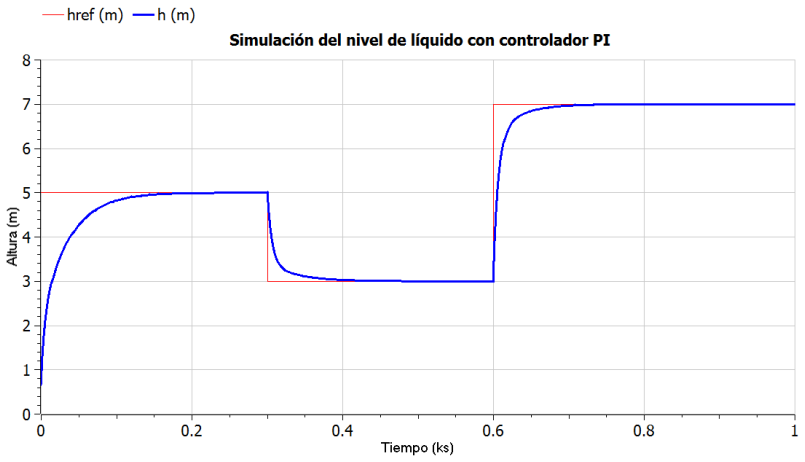


Figura 4.8: Evolución del nivel simulado en OpenModelica bajo consignas escalonadas.

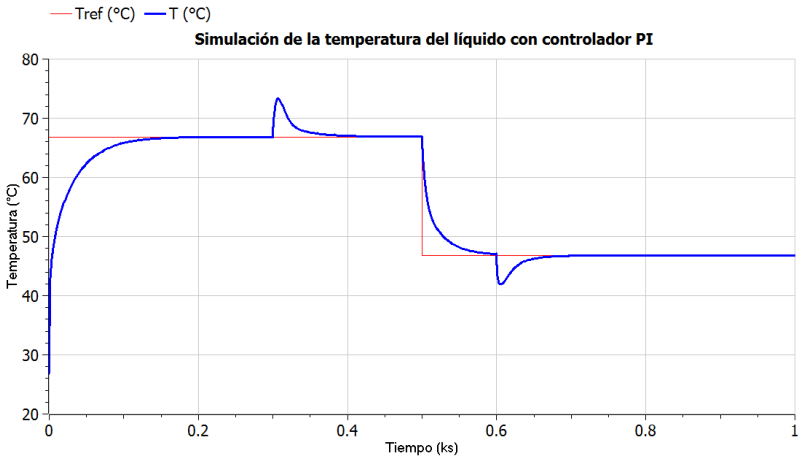


Figura 4.9: Evolución de la temperatura simulada en OpenModelica bajo consignas escalonadas.

Asimismo, resulta esencial observar el rango real de actuación de ambos controladores; si estos no están limitados, será necesario imponer restricciones a las acciones de control durante el entrenamiento del agente de aprendizaje por refuerzo, con el fin de obtener un comportamiento más realista y acorde con las capacidades físicas del sistema.

En este caso de estudio se plantea reemplazar los dos lazos de control clásicos por un único controlador inteligente entrenado mediante aprendizaje por refuerzo. De esta forma, el agente no solo debe aprender a mantener de forma conjunta el nivel y la temperatura en sus consignas, sino también gestionar la interacción entre ambas variables, característica fundamental de este problema multivariable.

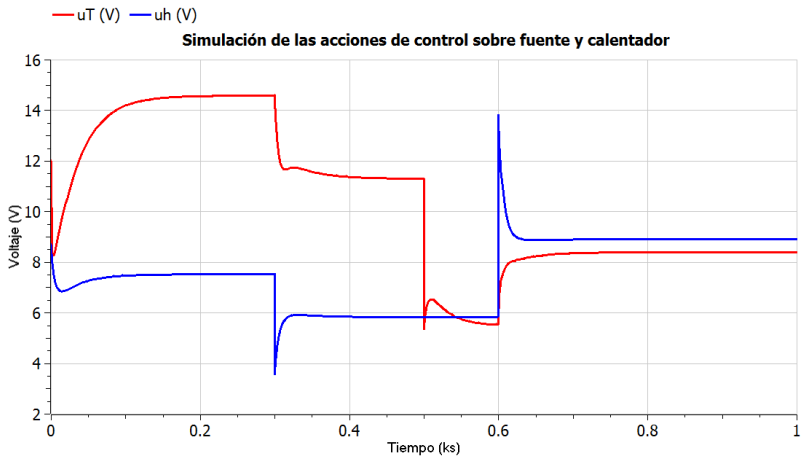


Figura 4.10: Señales de control $u_h$ y $u_T$ aplicadas por los controladores PI.

**Depósito de líquido con perturbaciones del entorno**

Con el objetivo de situar el problema de control en un escenario más realista y desafiante, se parte del modelo anterior y se incorporan varias fuentes de incertidumbre y perturbaciones exógenas. El propósito no es hacer un modelado exhaustivo del entorno, sino disponer de un banco de pruebas ilustrativo donde los controladores PI afronten condiciones adversas y se ponga de manifiesto la interacción entre lazos.

En particular, se han introducido las siguientes perturbaciones (implementadas en el fichero `Fluid_tank_PI_control_Noisy_env.mo`, ver Listado A.2):

1. **Temperatura ambiente estocástica.** La temperatura del entorno $T_{\text{amb}}$ se modela como una realización de un proceso de Wiener (movimiento browniano) sin deriva, que induce variaciones lentas pero persistentes en las pérdidas/gancias térmicas del depósito. La Figura 4.11 muestra una trayectoria típica.

2. **Pérdidas por evaporación.** Se añade un término de salida de masa $m_\ell$ dependiente del estado térmico y del nivel del fluido, con un factor de ganancia aleatorizado para representar incertidumbre de modelo. En la Figura 4.12 se comprueba que el caudal de evaporación $m_\ell$ se mantiene en todo momento muy por debajo del flujo de entrada $F_{in}$, asegurando que su modelado no sea exagerado en relación al balance de masa del sistema. También se comprueba en la gráfica que esta perturbación introduce ruido en la dinámica de nivel, observable en las oscilaciones del flujo de entrada $F_{in}$.

3. **Ruido de medida y actuadores reales.** Se agrega ruido blanco de baja potencia en los sensores de $h$ y $T$, y se consideran saturaciones y pequeños retardos en las

señales de control $u_h$ y $u_T$.

Las Figuras 4.13 y 4.14 ilustran el efecto conjunto de estas perturbaciones sobre el desempeño de los controladores PI. En el lazo de nivel, la aleatoriedad asociada a la evaporación y al ruido de medida dificulta el seguimiento preciso de la referencia, aumentando el error estacionario y el rizado de $h(t)$. En el lazo térmico se aprecia la *acoplamiento cruzado*: las oscilaciones de nivel alteran la inercia térmica $mC_p$ y, en consecuencia, el balance de energía, lo que obliga al controlador de temperatura a actuar con mayor agresividad para recuperar $T_{\text{ref}}(t)$ tras cambios en $h$.

En conjunto, el escenario perturbado incrementa la $IAE$ y la $TV$ de las señales de control respecto al caso nominal, y pone de relieve la sensibilidad del sistema a incertidumbres no modeladas. Este caso servirá más adelante para evaluar si un agente de aprendizaje por refuerzo, entrenado sobre este entorno ruidoso, es capaz de coordinar ambas acciones ($u_h$, $u_T$) con mayor robustez frente a perturbaciones y acoplamientos.

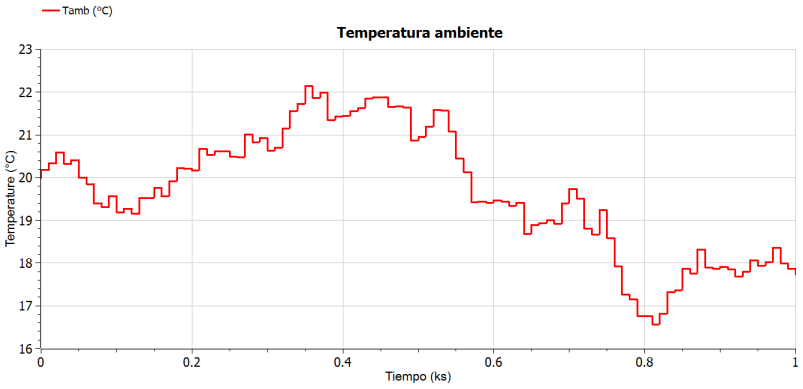


Figura 4.11: Realización de un proceso de Wiener utilizado para representar $T_{\text{amb}}(t)$ en OpenModelica.

### 4.2.3. Caso real: línea de sobrecalentamiento

El modelo considerado corresponde a una versión simplificada de una línea de sobrecalentamiento de vapor perteneciente a una planta real de generación eléctrica mediante turbina de vapor. Este modelo ha sido desarrollado en Dymola utilizando componentes avanzados de la biblioteca `ClaRa+`, ampliamente empleada para el modelado termofluidodinámico de sistemas de generación de energía. La línea está dividida en dos tramos principales, denominados `SH2` y `SH3`. Para el control de la temperatura en cada tramo se dispone de un pulverizador de agua, colocado antes de la entrada a cada uno de ellos. El

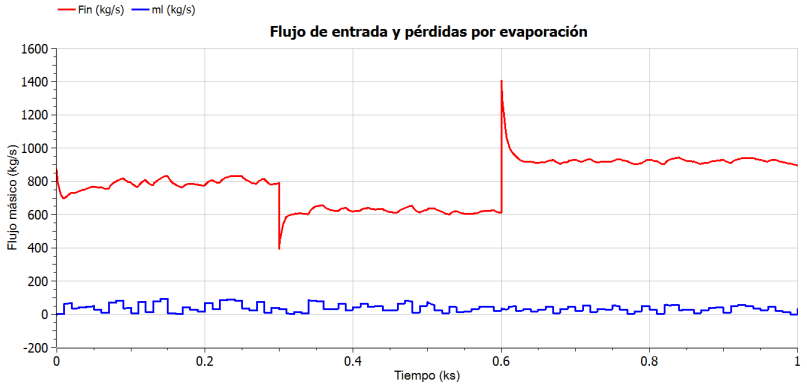


Figura 4.12: Comparación entre el flujo de entrada $F_{in}$ y las pérdidas por evaporación $m_\ell$ en el modelo perturbado.

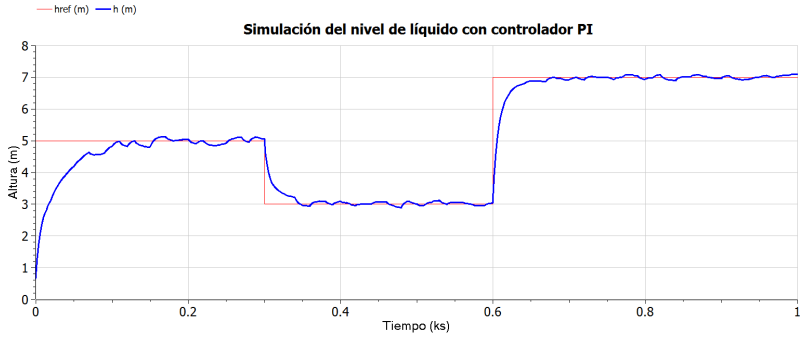


Figura 4.13: Seguimiento del nivel con PI bajo perturbaciones: aumento de rizado y error en régimen.

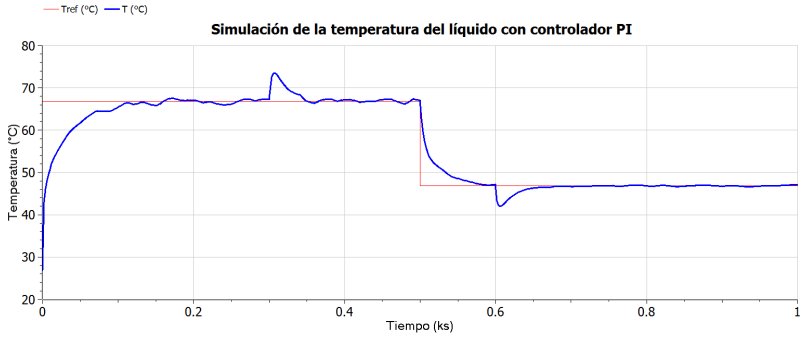


Figura 4.14: Seguimiento de temperatura con PI bajo perturbaciones: impacto del acoplamiento con el lazo de nivel.

sistema cuenta con cuatro sensores de temperatura estratégicamente ubicados: `T_SH2_in` y `T_SH2_out`, que miden la temperatura a la entrada y salida del tramo `SH2`, y `T_SH3_in` y `T_SH3_out`, que cumplen la misma función para el tramo `SH3`. Para representar la incertidumbre de medición, la señal de temperatura a la entrada del sistema ha sido perturbada con un ruido aleatorio de distribución uniforme. La disposición de estos componentes puede observarse en la Figura 4.15, que muestra el diagrama del modelo desarrollado en Dymola con componentes de la biblioteca `ClaRa+`.

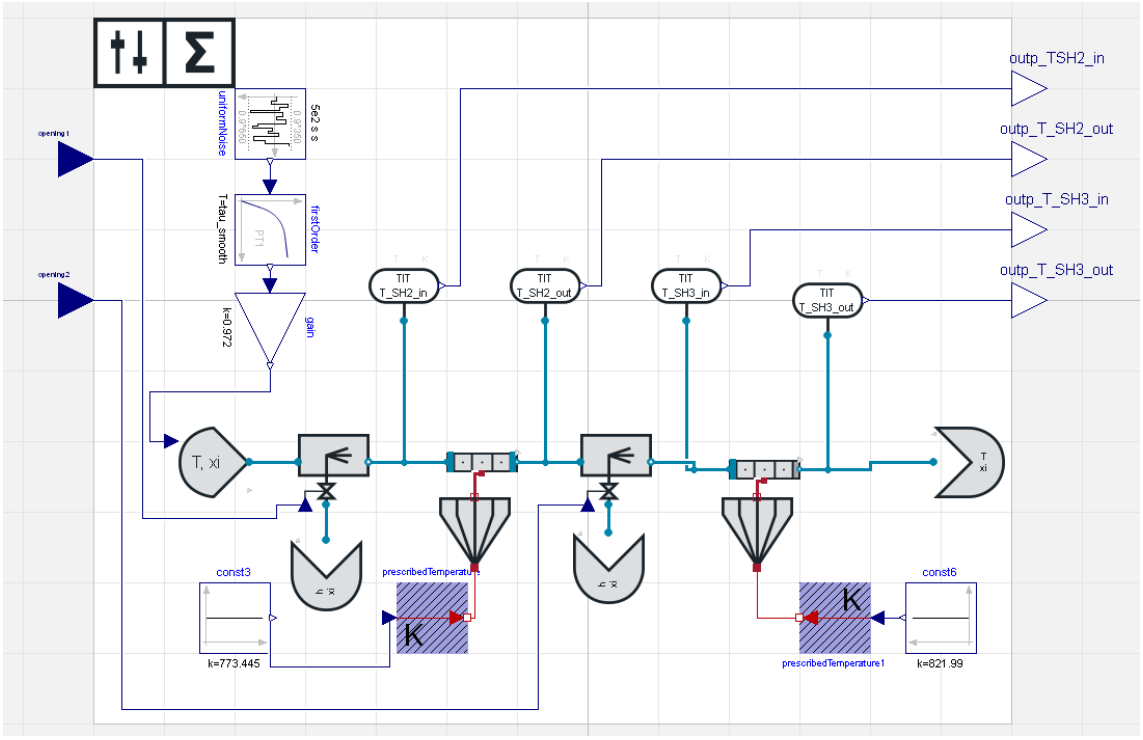


Figura 4.15: Diagrama del modelo de la línea de sobrecalentamiento, desarrollado en Dymola mediante componentes de la biblioteca `ClaRa+` (modelo proporcionado por XRG Simulation GmbH).

Cada pulverizador es accionado mediante una válvula de control gobernada por una cascada de controladores PI basada en las señales de los cuatro sensores de temperatura instalados en la línea. La Figura 4.16 muestra el esquema de control, en el que las dos señales de actuación (aperturas de las válvulas de los pulverizadores) se calculan para cumplir dos objetivos:

$$T_{\text{set}} = \texttt{outp\_T\_SH3\_out}, \tag{4.18}$$

$$\Delta T_{\text{set}} = \texttt{outp\_T\_SH2\_out} - \texttt{outp\_T\_SH3\_in}. \tag{4.19}$$

donde $T_{\text{set}} = 821{,}15$ K y $\Delta T_{\text{set}} = 24$ K son los valores de consigna para la temperatura de salida del tramo `SH3` y la diferencia de temperatura entre los tramos intermedios, respectivamente.

La problemática principal de este caso radica en las variaciones continuas de temperatura que se producen en distintos puntos de la línea. Si no se aplica ningún control de temperatura, aparecen oscilaciones pequeñas de `T_SH3_out` con respecto al valor de consigna de 821.15 K, siempre inferiores a 0.2 K. Aunque no son significativas en magnitud, estas fluctuaciones térmicas sostenidas pueden contribuir a tensiones y fatiga en los materiales de la tubería a largo plazo. Este riesgo se mitiga mediante los pulverizadores situados a la entrada de los tramos `SH2` y `SH3`, cuya función es refrigerar el vapor de manera controlada.

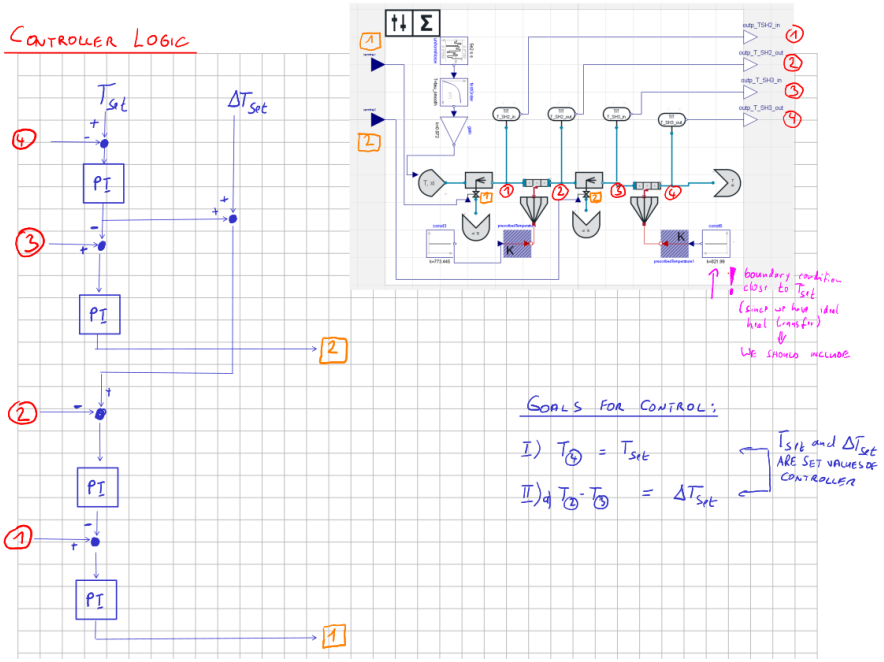


Figura 4.16: Esquema del sistema de control basado en una cascada de controladores PI para regular la temperatura en los tramos `SH2` y `SH3`.

Este modelo se ha incorporado al presente trabajo con el objetivo de evaluar controladores inteligentes en un escenario más complejo y realista, y comparar su desempeño con la estrategia de control clásico basada en controladores PI implementados en casca-

da. Además de ser representativo de una aplicación industrial, procede de un proyecto de investigación desarrollado por XRG Simulation GmbH, y su implementación en `ClaRa+` aporta un mayor nivel de detalle físico y precisión en el modelado del comportamiento dinámico del sistema.

## 4.3. Entrenamiento y evaluación de controladores inteligentes mediante `RLIC-FMU`

El desarrollo de este trabajo incluye el diseño, entrenamiento y validación de controladores basados en aprendizaje por refuerzo profundo (DRL) aplicados a modelos físicos de sistemas dinámicos exportados como FMU. Para ello se ha creado `RLIC-FMU`, una herramienta que integra entornos personalizados, agentes de aprendizaje y utilidades de entrenamiento, evaluación y despliegue. En esta sección se describen los principios de diseño de las funciones de recompensa, la configuración de los entornos de simulación y los resultados obtenidos en casos de estudio representativos que emplean los modelos de sistemas físicos presentados en el apartado anterior: un sistema carro–péndulo invertido, un depósito de líquido con control multivariable y una línea de sobrecalentamiento de vapor.

### 4.3.1. Diseño de la función de recompensa

En el presente trabajo se ha optado por emplear funciones de recompensa continuas basadas en el error de seguimiento respecto a las consignas del sistema. Este enfoque contrasta con muchos ejemplos clásicos de aprendizaje por refuerzo, como los entornos de carro con péndulo invertido de OpenAI Gym y los tutoriales de TF-Agents, así como la demostración de ModelicaGym (Lukianykhin y Bogodorova, 2019), donde se emplean recompensas discretas: el agente recibe una recompensa positiva constante ($+1$) por cada paso de simulación mientras las variables permanecen dentro de un rango aceptable, y una penalización elevada (p. ej., $-100$) cuando alguna condición de seguridad se incumple. Dicho esquema favorece episodios largos, pero no proporciona información fina sobre el desempeño en régimen estacionario ni sobre la calidad del control en presencia de oscilaciones alrededor del valor de referencia.

Para abordar este problema, todas las funciones de recompensa definidas en los entornos personalizados (`custom_envs.py`) se diseñaron de forma continua, utilizando expre-

siones de la forma:

$$r = \sum_i w_i \exp(-k_i|e_i|) + \text{términos de penalización/recompensa},$$

donde $e_i$ es el error absoluto de la variable $i$ respecto a su consigna, $w_i$ y $k_i$ son coeficientes de ponderación y sensibilidad respectivamente, y los términos adicionales penalizan cambios bruscos en las acciones de control o premian precisión extrema. Este tipo de función, suave y diferenciable, proporciona al agente información densa y gradualmente decreciente sobre su desempeño en cada paso, facilitando el ajuste fino de las variables controladas y evitando recompensas binarias poco informativas.

La Figura 4.17 ilustra el efecto de distintos valores de $k$: con valores moderados, la recompensa decrece suavemente y fomenta la exploración amplia, permitiendo al agente encontrar el *setpoint* incluso desde estados alejados; valores altos de $k$ favorecen una precisión extrema una vez alcanzada la referencia, pero dificultan el aprendizaje inicial al ofrecer gradientes muy bajos lejos del punto deseado. La elección de $k$ es, por tanto, un compromiso entre exploración y precisión, y constituye un parámetro clave en el diseño de recompensas continuas para entornos físicos.

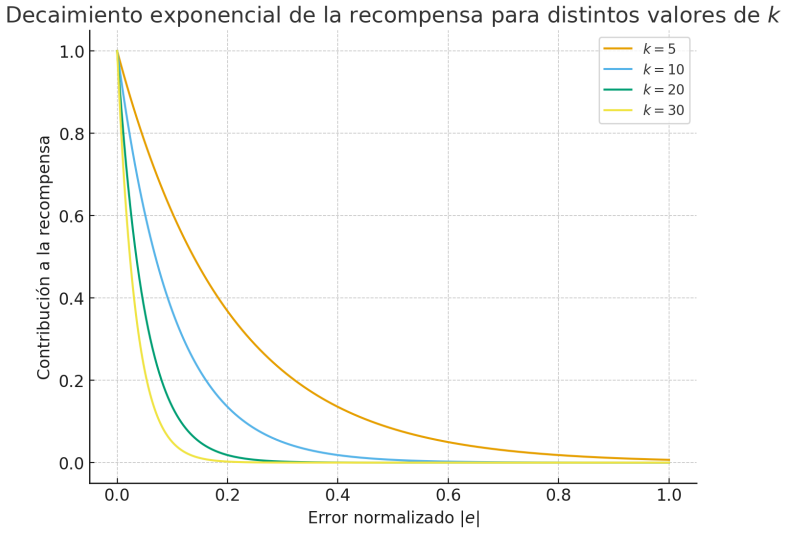


Figura 4.17: Efecto del parámetro $k$ en la función de recompensa exponencial: a mayor $k$, mayor sensibilidad cerca del *setpoint* y menor información lejos de él.

Por ejemplo, en el entorno `CartPole` se combinan dos exponenciales que penalizan los errores de ángulo y posición del carro, junto con términos cuadráticos que penalizan tanto la magnitud de la acción aplicada como su variación entre pasos consecutivos; en el `Watertank` se penalizan simultáneamente errores de nivel y temperatura, añadiendo

pequeñas recompensas por exactitud extrema; y en el `Superheater` se incluye una penalización adicional proporcional a la variación de los actuadores. Este planteamiento ha permitido entrenar políticas que no sólo maximizan la duración de los episodios, sino que también mantienen un comportamiento estable y preciso en régimen estacionario.

### 4.3.2. Caso de estudio: carro con péndulo invertido

**Entorno y límites.** Este primer caso de estudio se ha diseñado para ilustrar el flujo de trabajo de entrenamiento en `RLIC-FMU` y mostrar las métricas y gráficos que la herramienta genera por defecto. El entorno `CartPole`, definido en `custom_envs.py`, envuelve el modelo del sistema carro–péndulo invertido desarrollado en OpenModelica y presentado en el apartado 4.2.1, el cual se ha exportado como FMU para su ejecución paso a paso mediante `PyFMI`. El agente debe aprender a mantener el péndulo en posición vertical y, simultáneamente, a estabilizar el carro en la posición $x = 0$, lo que aumenta significativamente la dificultad de la tarea frente a versiones simplificadas del problema.

El espacio de observaciones está compuesto por cuatro variables: posición y velocidad del carro, ángulo $\phi$ del péndulo respecto a la vertical y su velocidad angular. Los límites de observación, establecidos en el `BoundedArraySpec`, son $[-2{,}4, 2{,}4]$ m para la posición, $[-5, 5]$ m/s para su velocidad, y $[78^\circ, 102^\circ]$ (equivalente a $\pm 12^\circ$ alrededor de la vertical) para el ángulo, valor tomado de implementaciones previas en OpenAI Gym y ModelicaGym, donde se considera que para ángulos mayores el péndulo no es recuperable. El espacio de acción es unidimensional y continuo, definido en el intervalo $[-100, 100]$ N. La clase también implementa una función de recompensa que combina seguimiento de referencias con penalizaciones cuadráticas sobre la magnitud y variación de la acción aplicada.

La configuración del entorno se gestiona mediante la clase `EnvironmentConfig`, como se ilustra en el script `cart_pole_dqn.py` (Listado B.1 en el Apéndice B). Este objeto permite especificar parámetros clave como el paso de integración (`ts`), el número máximo de pasos por episodio (`steps_max`), valores de penalización para episodios fallidos y opciones de visualización. En este caso, se determinó que, para la dinámica del sistema, resulta adecuado emplear un paso de integración de 0,01 s, indicado mediante el argumento `ts`. Se ha elegido un límite de 1000 pasos por episodio, de modo que cada episodio tiene una duración máxima de 10 s. El fragmento de código siguiente muestra cómo se inicializa la configuración en el script:

Listado 4.1: Configuración del entorno en `cart_pole_dqn.py`.

```
# configure environment
config = EnvironmentConfig(parameter_dict=None,
                           ts=0.01,               # simulation step size (s)
                           steps_max=1000,        # max steps per episode
                           display=True,
                           negative_reward=0,
                           log_level=logging.DEBUG,
                           discretize=True,       # discretize continuous
                               actions
                           num_actions=7,         # odd number so force
                               includes 0
                           fmu_silent_mode=True).get_config()
```

**Recompensa.** La recompensa del entorno `CartPole` se define como:

$$r_t = k\, e^{-a e_{\phi,t}} + k\, e^{-a e_{x,t}} - \lambda_u \left(\frac{u_t}{u_{\text{máx}}}\right)^2 - \lambda_{\Delta u} \left(\frac{u_t - u_{t-1}}{\Delta u_{\text{máx}}}\right)^2,$$

donde $e_{\phi,t}$ y $e_{x,t}$ representan los errores instantáneos de ángulo y posición, $u_t$ es la fuerza aplicada al carro y $u_{\text{máx}}$ el valor máximo permitido (100 N). El parámetro $k$ ajusta la recompensa máxima, mientras que $a$ controla la sensibilidad al error. Los coeficientes $\lambda_u$ y $\lambda_{\Delta u}$ ponderan las penalizaciones cuadráticas asociadas al uso de energía y a la variación de la acción, respectivamente.

En este caso se emplearon los valores $k = 100$, $a = 30$, $\lambda_u = 2{,}0$ y $\lambda_{\Delta u} = 2{,}0$. Este diseño genera una recompensa continua y suave, que incentiva mantener el equilibrio y penaliza soluciones con fuerzas extremas o cambios bruscos.

**Elección del agente (DQN).** Dado que se optó por emplear DQN por su sencillez y robustez en espacios de acción discretos, fue necesario discretizar la fuerza aplicada al carro, originalmente definida como una acción continua en el intervalo $[-100, 100]$ N. Esta opción se activa estableciendo `discretize=True` y `num_actions=7` en la configuración del entorno. Con estos parámetros, el `EnvironmentFactory` aplica el envoltorio `ActionDiscretizeWrapper` de TF-Agents, generando siete niveles de fuerza equiespaciados:

$$\{-100,\ -66{,}67,\ -33{,}33,\ 0,\ 33{,}33,\ 66{,}67,\ 100\}\ \text{N},$$

asegurando así que el conjunto de acciones incluya exactamente 0 N.

El agente DQN implementado en `dqn.py` utiliza una red neuronal aproximadora de la función $Q$ con cuatro capas densas (100, 100, 100 y 50 neuronas, activación `ReLU`) y una tasa de aprendizaje de $10^{-3}$. El bloque de código siguiente muestra cómo se define la configuración del agente:

Listado 4.2: Definición de la configuración del agente DQN.

```
# define agent and experiment
tf_agent_config = {
    'tf_agent_name': "DQN",
    'fc_layer_params': (100, 100, 100, 50)}
```

**Configuración del entrenamiento.** El experimento se configura mediante el script `cart_pole_dqn.py` y la clase `ExperimentRL`, que gestiona tanto el entrenamiento como la evaluación periódica del agente. En este caso, se definen 1000 episodios de entrenamiento y una evaluación cada 100 episodios (`eval_interval`), ejecutando un único episodio de prueba en cada evaluación (`num_eval_episodes=1`). Antes de comenzar el entrenamiento, se recolectan 10 000 pasos iniciales (`initial_collect_steps`) para llenar el *replay buffer*. El fragmento de código siguiente muestra cómo se inicializa el experimento y se ejecuta el proceso de entrenamiento:

Listado 4.3: Inicialización del experimento y parámetros de entrenamiento.

```
num_eval_episodes = 1

experimentDQN = ExperimentRL(
    tf_agent_config,
    model_name,
    config,
    num_eval_episodes,
    training_figure,
    training_ax,
    initial_collect_steps=10000,
    use_tf_functions=use_tf_functions
)

# train the agent
num_episodes = 1000
trained_agent, episodes_length, exec_time = experimentDQN.train(
    num_iterations=10000000,
    training_end="episodes",
```

```
    max_num_episodes=num_episodes,
    eval_interval=100,
)
```

**Resultados del entrenamiento.** Al ejecutar el método `train` de la clase `ExperimentRL`, `RLIC-FMU` abre automáticamente una ventana de monitorización en tiempo real con dos gráficos complementarios: el superior muestra el *retorno de evaluación* del episodio (etiquetado como "Average Return" porque, si `num_eval_episodes`>1, se traza la media), y el inferior la *duración media del episodio* en número de pasos ("Average Episode Length"). El eje X representa episodios o pasos de entrenamiento según el parámetro `training_end`: si `training_end="episodes"` (caso aquí), el eje X cuenta episodios y el fin del entrenamiento lo fija `max_num_episodes`; si `training_end="steps"`, el eje X cuenta pasos y el límite lo determina el argumento `num_iterations` del propio `train`. La frecuencia de actualización se controla con `eval_interval`: cada `eval_interval` episodios/pasos se ejecuta una evaluación de `num_eval_episodes` episodios y el resultado (media si procede) se añade a los gráficos. La Figura 4.18 ilustra estos trazados para el experimento `CartPole` con DQN.

La curva de aprendizaje muestra una fase inicial lenta, con retornos medios próximos a cero hasta aproximadamente el episodio 200, seguida de una mejora pronunciada. Hacia el episodio 300, el agente alcanza de forma consistente el máximo de 1000 pasos por episodio (límite configurado), lo que indica que ha resuelto la tarea. A partir de ese punto, el retorno medio fluctúa en el rango de $7 \times 10^4$ a $1{,}6 \times 10^5$, variación atribuible tanto a la función de recompensa dependiente del error como a la exploración $\epsilon$-greedy. Pese a estas oscilaciones, la duración de los episodios permanece saturada en el valor máximo, evidenciando estabilidad del comportamiento aprendido. *Nota:* el método `train` de `ExperimentRL` devuelve el mejor modelo observado durante el entrenamiento (según retorno), no necesariamente el último.

La Figura 4.19 muestra el resultado de la evaluación de la política entrenada mediante `RLIC-FMU`. Esta representación gráfica corresponde al formato estándar que genera la herramienta tanto en evaluaciones periódicas durante el entrenamiento como en evaluaciones finales mediante la clase `Deployment`, como es este caso. La ventana se compone de un número variable de gráficas determinado por la configuración del entorno definida en la clase correspondiente en el módulo `custom_obs`, en este caso la clase `CartPole`.

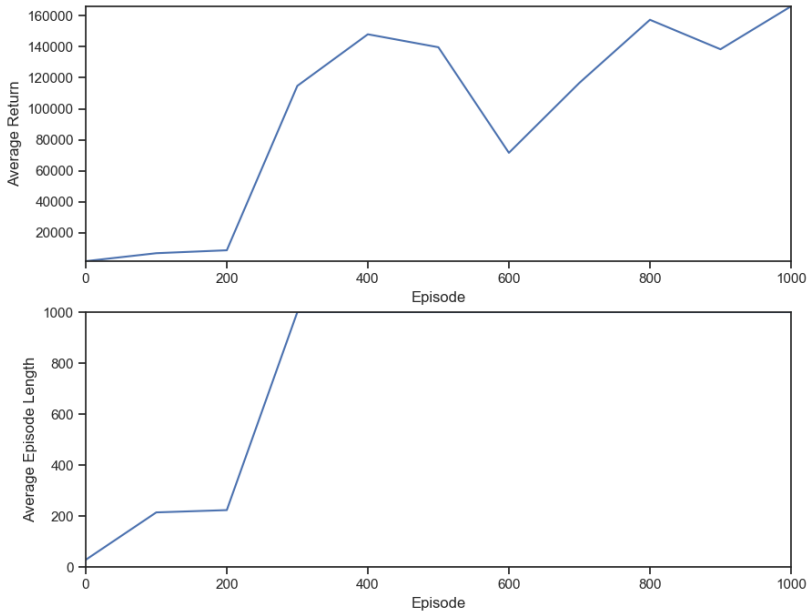


Figura 4.18: Curva de aprendizaje del agente DQN en `CartPole`: arriba, retorno medio por episodio; abajo, longitud media del episodio (número de pasos).

En la parte superior se incluye siempre una gráfica con las acciones del agente; en este caso, la única acción corresponde a la fuerza aplicada al carro (en N). A continuación, se representan las observaciones seleccionadas en `config['observations_to_render']`, que en este ejemplo es `[0, 2]`. Esto implica que se muestran únicamente las dos variables de mayor interés: la posición del carro (en metros respecto al origen) y el ángulo $\phi$ del péndulo con respecto a la horizontal (en radianes). Sobre estas gráficas se superponen los límites definidos en el atributo `self._observation_spec`; si estos límites se superan, el episodio finaliza. Es recomendable que dichos rangos no sean excesivamente amplios, ya que las redes neuronales tienen mayor sensibilidad a los cambios de observación en intervalos acotados. Cabe recordar que las observaciones se normalizan internamente en el rango $[0, 1]$. También se representa el valor de consigna (set-point) —en este caso constante en 90° o 1,57 rad—, fundamental para el cálculo de la recompensa. Finalmente, se incluye una gráfica con la evolución de la recompensa a lo largo del episodio, cuyo valor depende de la función definida en el método `get_reward`.

A partir de esta visualización se aprecia un seguimiento preciso tanto de `pos_cart` como de `phi_pole`. En la señal de control destaca que el actuador rara vez alcanza los

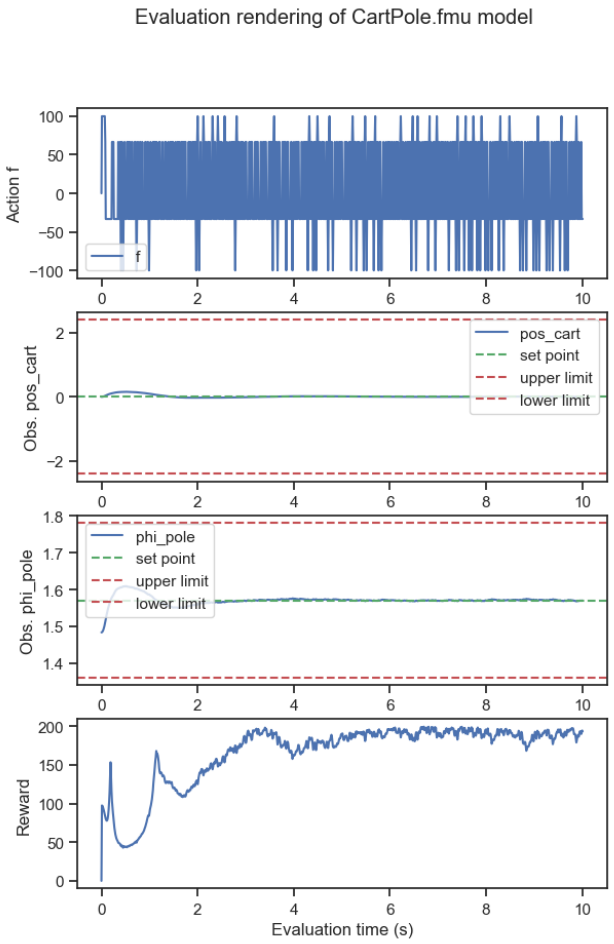


Figura 4.19: Evaluación del control inteligente sobre el sistema de carro con péndulo invertido.

límites $\pm 100\,\mathrm{N}$, lo que refleja el efecto del término de penalización por energía en la recompensa: desincentiva esfuerzos máximos sostenidos. No obstante, la acción todavía presenta una variabilidad elevada. Esta podría reducirse aumentando el peso $\lambda_{\Delta u}$ en la función de recompensa, atenuando los cambios rápidos de la fuerza. La contrapartida es que valores demasiado altos pueden ralentizar la respuesta y empeorar la precisión fina en régimen estacionario.

Para evaluar la robustez y la capacidad de generalización del controlador, se reevalúa el agente DQN ya entrenado bajo condiciones distintas a las usadas durante el entrenamiento. En concreto, mediante el parámetro `parameter_dict` de `EnvironmentConfig` se fijó `phi_start` $= 100/180 \cdot \pi$ rad respecto a la horizontal (100°, frente al valor por defecto de 80°) y `m_cart` $= 12\,\mathrm{kg}$ (entrenado con $10\,\mathrm{kg}$). Además, el horizonte del episodio

de evaluación se amplió a 50 s (frente a 10 s en entrenamiento). Como se aprecia en la Figura 4.20, el agente DQN mantiene un comportamiento estable bajo estas variaciones, evidenciando capacidad de generalización.

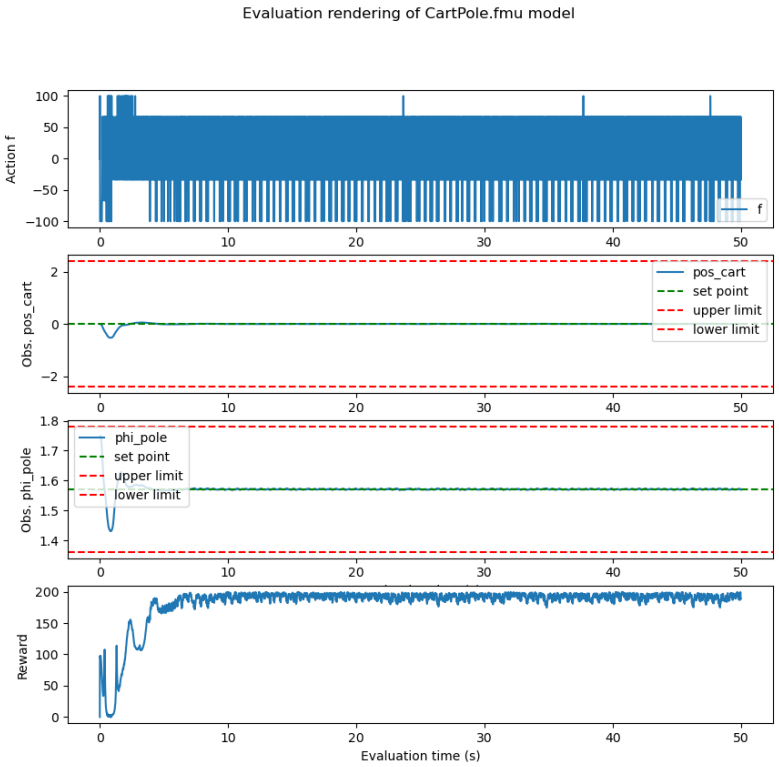


Figura 4.20: Evaluación de robustez del agente DQN en `CartPole` con condiciones distintas a las del entrenamiento: `phi_start` = 100° desde la horizontal (100/180 · $\pi$ rad), `m_cart` = 12 kg (entrenado con 10 kg) y horizonte de 50 s (entrenamiento a 10 s). El controlador mantiene la estabilidad y un rendimiento adecuado, mostrando capacidad de generalización.

### 4.3.3. Caso de estudio: depósito de líquido

**Entorno y límites.** El entorno `Watertank` modela un depósito con control simultáneo de nivel y temperatura. El vector de observaciones es $[h,\ h_{\text{ref}},\ T,\ T_{\text{ref}}]$, con cotas (en unidades SI) $h, h_{\text{ref}} \in [0{,}1,\ 10]$ y $T, T_{\text{ref}} \in [300,\ 370]$. El espacio de acción es bidimensional y continuo, $u = [u_h,\ u_T] \in [-20,\ 20]^2$, donde $u_h$ es la apertura de la válvula de entrada de agua y $u_T$ la potencia de calefacción. El entorno puede operar con consignas fijas o con referencias generadas dinámicamente por episodio mediante escalones aleatorios

(`random_setpoints`). Para enriquecer la experiencia de entrenamiento, en cada episodio las consignas de nivel y temperatura se extraen de los siguientes rangos:

$$h_{\text{ref}} \in [2{,}0, 8{,}0]\ \text{m}, \quad \Delta t_{\text{esc},h} \in [30, 60]\ \text{s}; \qquad T_{\text{ref}} \in [310, 350]\ \text{K}, \quad \Delta t_{\text{esc},T} \in [100, 200]\ \text{s}.$$

Este planteamiento expone al agente a una amplia variedad de situaciones y reduce el riesgo de sobreajuste a consignas constantes.

El entorno cuenta con dos FMU —una base y otra con perturbaciones— que corresponden a los modelos del depósito presentados en el apartado 4.2.2. En este estudio se ha entrenado el agente sobre el modelo base, reservando el modelo con perturbaciones para evaluar posteriormente la capacidad de generalización del controlador aprendido.

**Recompensa.** Las observaciones se normalizan linealmente usando los rangos definidos en la especificación `BoundedArraySpec` (p. ej., $\hat{h} = (h - h_{\text{mín}})/(h_{\text{máx}} - h_{\text{mín}})$ y análogamente para $T$). Definimos los errores normalizados $e^{(n)}_{h,t} = |\hat{h}_t - \hat{h}_{\text{ref},t}|$ y $e^{(n)}_{T,t} = |\hat{T}_t - \hat{T}_{\text{ref},t}|$, con $e^{(n)} \in [0, 1]$. La recompensa es:

$$r_t \;=\; k\, e^{-a\, e^{(n)}_{h,t}} \;+\; k\, e^{-a\, e^{(n)}_{T,t}} \;+\; b_h\, \mathbf{1}_{\{e^{(n)}_{h,t} < \varepsilon_h\}} \;+\; b_T\, \mathbf{1}_{\{e^{(n)}_{T,t} < \varepsilon_T\}},$$

donde $k$ fija la recompensa máxima por seguimiento y $a$ su sensibilidad al error. En los experimentos se usaron $k = 0{,}5$, $a = 30$, $b_h = b_T = 0{,}5$ y umbrales adimensionales $\varepsilon_h = \varepsilon_T = 10^{-3}$. Los términos exponenciales incentivan estados cercanos a la consigna, y las bonificaciones premian precisión muy alta en el régimen normalizado.

**Elección del agente (DDPG).** La razón principal para emplear DDPG es que el entorno exige *dos* acciones simultáneas, mientras que DQN sólo admite *una* acción (y, además, en espacios discretos). Por el contrario, DDPG está diseñado para espacios de acción *continuos y multidimensionales*, encajando de forma natural con $u = [u_h, u_T] \in [-20, 20]^2$. Para la exploración se utiliza ruido de Ornstein–Uhlenbeck ($\sigma = 30$, $\theta = 0{,}5$), que introduce correlación temporal apropiada para sistemas físicos continuos. El agente DDPG definido en `ddpg.py` emplea redes neuronales profundas con tres capas densas en el actor $(400, 300, 300)$ y dos capas $(400, 300)$ en el crítico, una tasa de descuento $\gamma = 0{,}995$ y un factor de escalado de recompensa de 10.

**Configuración del entrenamiento.** La configuración emplea un paso de integración $ts = 1{,}0\,\text{s}$ y un máximo de 2000 pasos por episodio (`steps_max`), de modo que cada

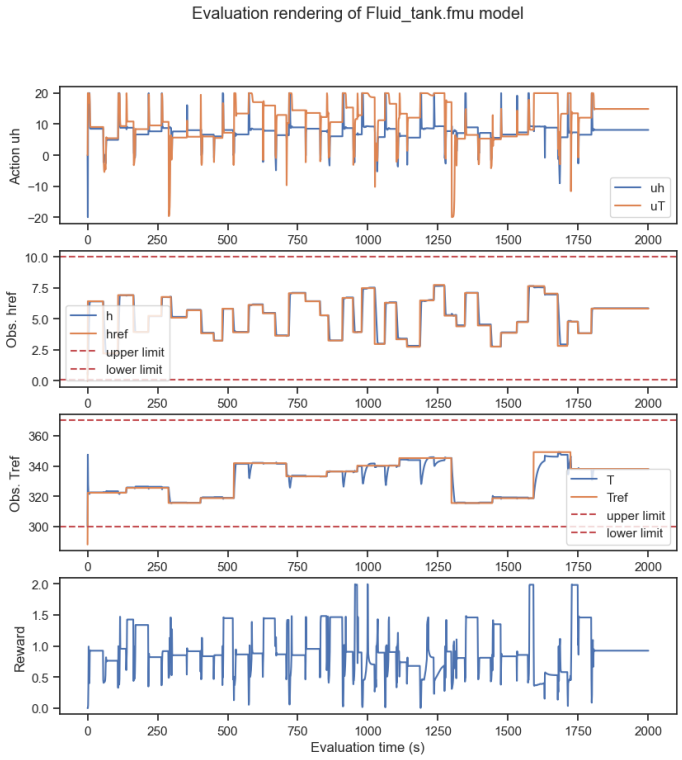


Figura 4.21: Última evaluación del agente DDPG durante el entrenamiento (mejor caso) con referencias aleatorias: acciones (arriba), seguimiento de $h$ frente a $h_{\text{ref}}$ y de $T$ frente a $T_{\text{ref}}$ (centro), y recompensa instantánea (abajo).

episodio cubre 2000 s de simulación. Para inicializar el *replay buffer* se recolectan 50,000 pasos aleatorios (`initial_collect_steps`). El entrenamiento se organiza en hasta 3000 episodios, con evaluaciones periódicas cada 300 episodios de entrenamiento. Todos los detalles se encuentran implementados en el script `water_tank.py`.

**Resultados del entrenamiento.** A lo largo del entrenamiento se realizaron evaluaciones periódicas (cada 300 episodios de entrenamiento) con consignas aleatorias para nivel y temperatura, a fin de monitorizar la evolución del desempeño bajo señales de referencia variadas. La Figura 4.21 muestra la *última evaluación* realizada durante el entrenamiento —la de mejor rendimiento—: el agente sigue con precisión $h$ y $T$ respecto a $h_{\text{ref}}$ y $T_{\text{ref}}$ a lo largo de 2000 s, mantiene las variables dentro de límites y alcanza recompensas altas con caídas transitorias en los cambios de consigna.

La Figura 4.22 muestra un arranque rápido: hacia el episodio $\sim 300$ la *duración media* ya alcanza el tope de 2000 pasos. Se observa una regresión temporal en torno a 800–1000

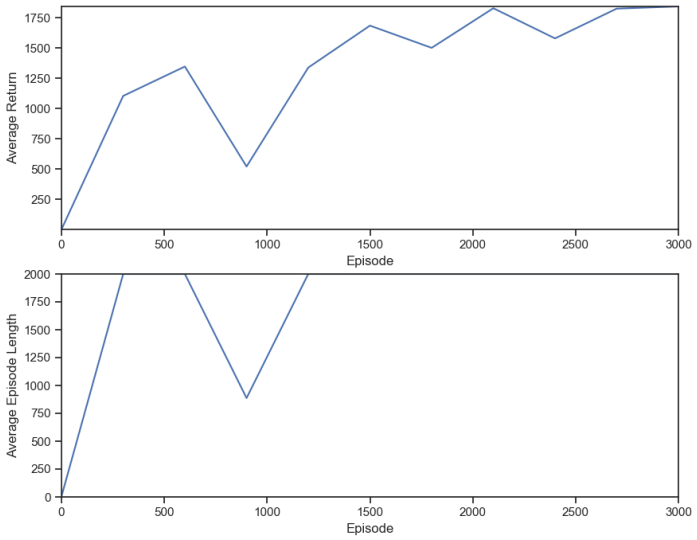


Figura 4.22: Con la configuración del agente DDPG, el entrenamiento muestra un aprendizaje sostenido en el entorno del depósito de líquido.

episodios (caída simultánea de retorno y duración hasta $\sim$ 900 pasos), seguida de una recuperación clara: desde $\sim$ 1200 episodios las evaluaciones vuelven a saturar el máximo de duración y el *retorno medio* mantiene una tendencia ascendente con oscilaciones moderadas. Este patrón sugiere estabilización progresiva del crítico y una política que, pese a la exploración, conserva episodios largos y un seguimiento más preciso.

Tras el entrenamiento, la política se exportó a TensorFlow y se evaluó mediante la clase `Deployment`. La Figura 4.23 muestra la respuesta en el entorno base (sin ruido) ante consignas escalonadas de nivel y temperatura: el agente sigue las referencias con tiempos de asentamiento cortos, mantiene las variables dentro de sus límites y sólo presenta un leve sesgo estacionario en el nivel durante algunos peldaños. Las acciones $u_h$ y $u_T$ permanecen acotadas y concentran sus picos en los cambios de consigna, mientras que la recompensa converge rápidamente a valores altos con caídas transitorias.

Para evaluar la capacidad de generalización, se ejecutó la misma política DDPG (exportada y cargada con `Deployment`) en el modelo del depósito con perturbaciones no vistas en entrenamiento (`Fluid_tank_Noisy_env.fmu`). La Figura 4.24 muestra que, pese al ruido de proceso/medida, el agente mantiene un seguimiento razonable de nivel y temperatura, con asentamientos algo más lentos; las acciones permanecen acotadas y la recompensa, aunque más oscilante, converge hacia valores altos en los tramos estacionarios.

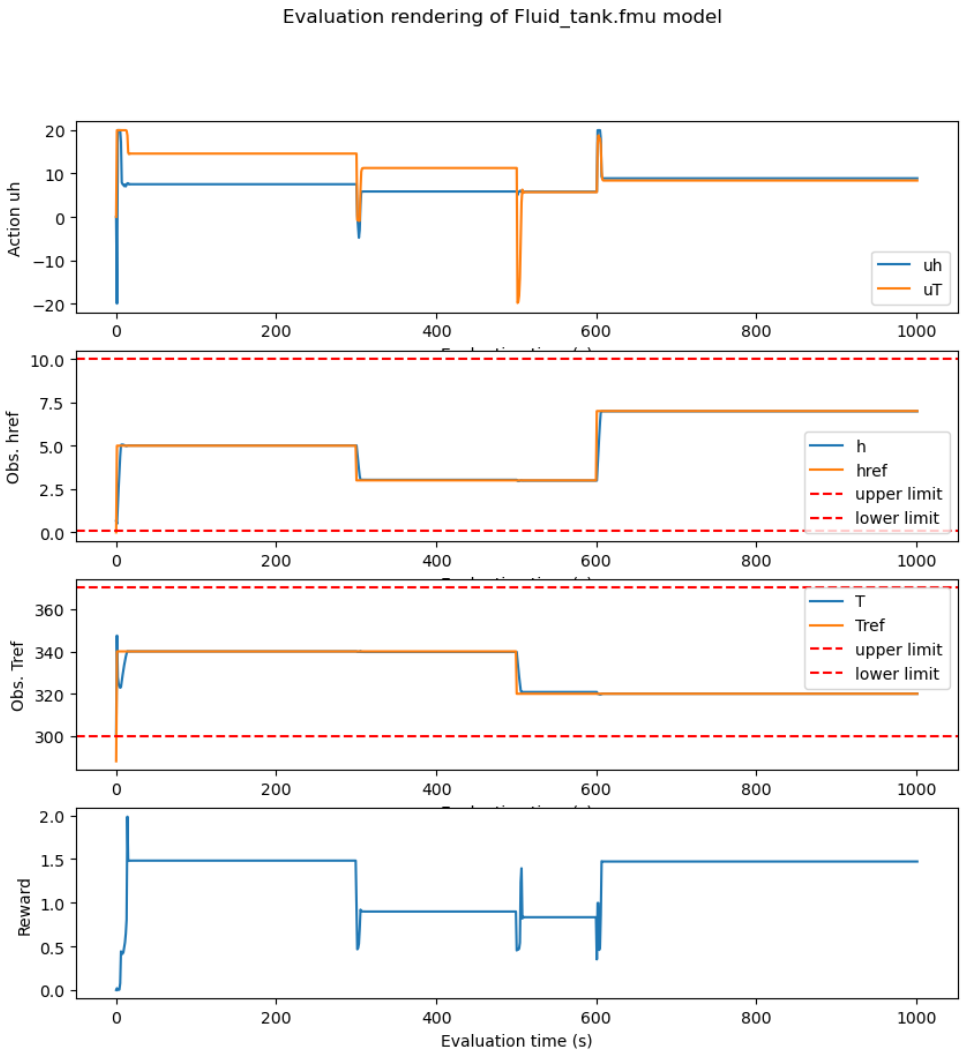


Figura 4.23: Entorno del depósito (sin ruido). Evaluación con `Deployment` de la política DDPG exportada a TensorFlow: seguimiento de consignas escalonadas en nivel y temperatura, acciones acotadas y recompensa alta estable.

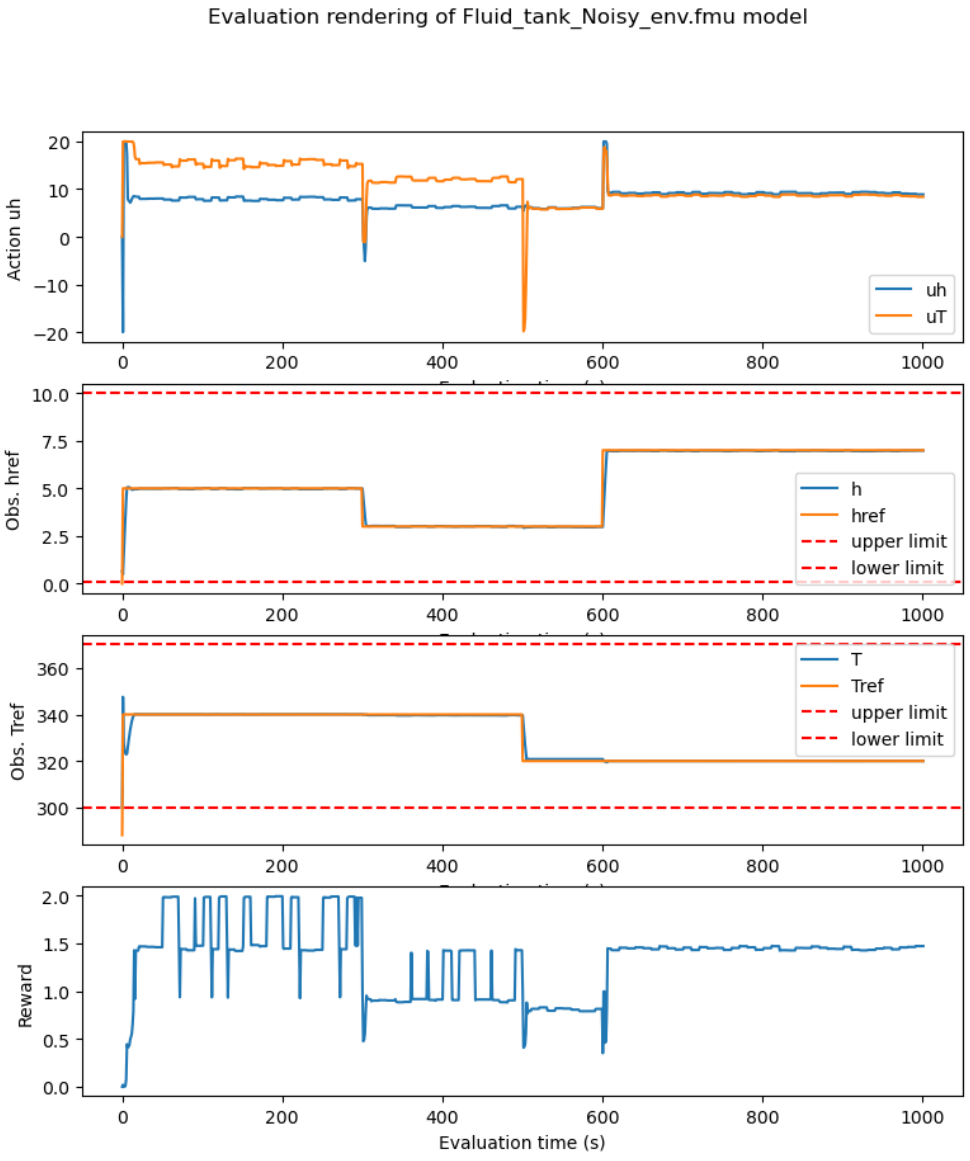


Figura 4.24: Depósito con perturbaciones: evaluación con `Deployment` de la política DDPG exportada; seguimiento bajo ruido, acciones acotadas y recompensa alta con oscilaciones.

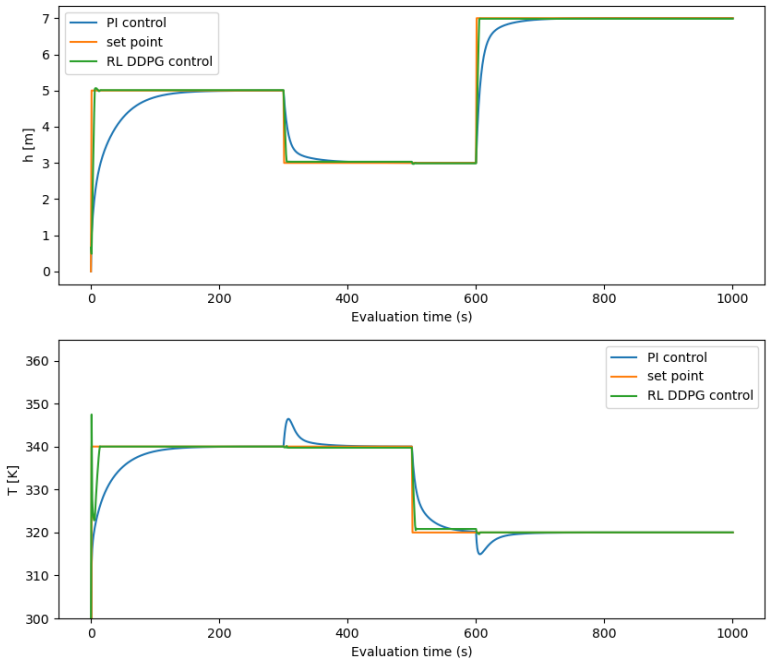


Figura 4.25: Depósito sin perturbaciones: DDPG sigue las consignas con subida más rápida y, a diferencia de PI, no muestra influencia apreciable del lazo de nivel sobre el de temperatura (mejor gestión del acoplamiento).

**Comparación con controlador PI.** En el modelo base (sin perturbaciones), la Figura 4.25 muestra que la política DDPG sigue las consignas de nivel y temperatura con tiempos de subida más cortos y menor sobreimpulso que el PI, especialmente en el escalón intermedio de temperatura. Como desventaja, se aprecia un *sesgo estacionario* pequeño en el controlador inteligente que, aunque reducido, podría ser determinante en procesos que exigen alta precisión o por acumulación de error a largo plazo; es un aspecto a mejorar en el futuro, ya sea mediante un diseño de recompensa más adecuado, un ajuste fino de hiperparámetros o incluso adoptando métodos más recientes. En contraste, una ventaja clara es que, ante escalones en el nivel, el DDPG prácticamente elimina las perturbaciones inducidas en la temperatura: actúa como un controlador *multivariable* coordinado, gestionando mejor el acoplamiento entre lazos que dos PI independientes.

En el modelo con perturbaciones no vistas durante el entrenamiento, la Figura 4.26 muestra que el DDPG mantiene, en promedio, un error menor frente a las consignas, evidenciando mayor robustez y capacidad de generalización. En particular, la política se

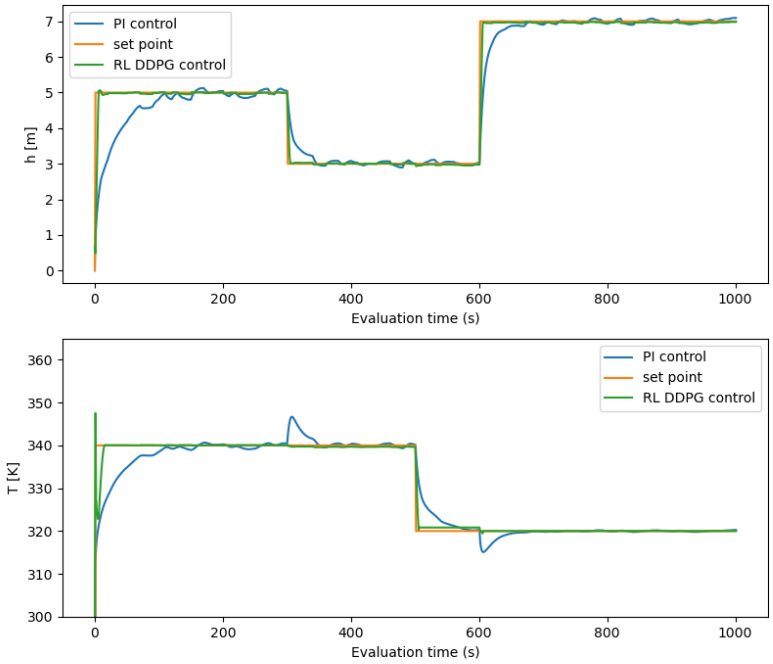


Figura 4.26: Depósito con perturbaciones: DDPG presenta menor error y mayor robustez frente a PI.

adapta mejor a un entorno con incertidumbre, aun cuando esa incertidumbre no ha sido vista durante el entrenamiento, conservando un seguimiento más ajustado tanto en nivel como en temperatura que el controlador PI.

Como se resume en la Tabla 4.2, la política DDPG obtiene recompensas totales superiores tanto en el modelo base como en el modelo con perturbaciones. Es importante subrayar que estas cifras *no* corresponden a métricas clásicas de control (p. ej., IAE/ISE/ITAE), sino a la recompensa del entorno de RL, que penaliza de forma muy severa estar lejos del *set-point* y bonifica la precisión fina. En consecuencia, el menor valor del PI se explica por su mayor tiempo de respuesta en ambos casos y por las alteraciones debidas al ruido en el segundo escenario. Por otro lado, conviene recordar que el controlador inteligente presenta un *sesgo estacionario* pequeño: en aplicaciones que requieran gran exactitud sostenida, este aspecto deberá mejorarse (p. ej., ajustando la recompensa, refinando hiperparámetros o adoptando métodos más recientes).

Tabla 4.2: Recompensa total acumulada (función de recompensa del entorno RL) en el depósito para la política DDPG y el controlador PI.

| **Entorno** | **RL / DDPG** | **PI** |
|---|---|---|
| Base (sin ruido) | 1275.7 | 678.8 |
| Con perturbaciones | 1335.4 | 184.9 |

### 4.3.4. Caso de estudio: línea de sobrecalentamiento

**Entorno y límites.** Se emplean como observaciones las cuatro señales de salida de los sensores descritos en el apartado 4.2.3: `outp_TSH2_in`, `outp_T_SH2_out`, `outp_T_SH3_in` y `outp_T_SH3_out`. Además, mediante la funcionalidad de *observaciones personalizadas* de `RLIC-FMU` se define la variable `outp_Delta_T_SH2_out_T_SH3_in`, calculada como la diferencia `outp_T_SH2_out` − `outp_T_SH3_in`. Esta observación adicional permite asociar explícitamente el *setpoint* de salto térmico (24 K) a una variable del espacio de observación. Los límites de las observaciones se han ajustado al rango real de variación del proceso para favorecer la normalización y la sensibilidad del aprendizaje: $[650, 750]$ K en `outp_TSH2_in`, $[770, 775]$ K en `outp_T_SH2_out`, $[700, 800]$ K en `outp_T_SH3_in`, $[820{,}15, 822{,}15]$ K en `outp_T_SH3_out` y $[-5, 60]$ K en `outp_Delta_T_SH2_out_T_SH3_in`.

El vector de acción es bidimensional y continuo, $u = [\mathit{opening1},\ \mathit{opening2}] \in [0, 1]^2$, correspondiente a las aperturas de los pulverizadores. Las consignas son $T^{\mathrm{ref}}_{\mathrm{SH3,out}} = 821{,}15\,\mathrm{K}$ y $\Delta T^{\mathrm{ref}} = 24\,\mathrm{K}$. El episodio comienza tras un breve *warm-up* del modelo ($t_0 = 100\,\mathrm{s}$).

**Recompensa.** La recompensa combina penalizaciones exponenciales sobre los errores de temperatura con un término adicional que penaliza cambios bruscos en las acciones:

$$r = 0{,}5e^{-20|T^{\mathrm{ref}}_{\mathrm{SH3,out}} - T_{\mathrm{SH3,out}}|} + 0{,}5e^{-20|\Delta T - \Delta T^{\mathrm{ref}}|} - 0{,}003 \sum |\Delta u|.$$

**Elección del agente (DDPG).** Se emplea un agente DDPG, adecuado para un espacio de acción continuo y bidimensional (correspondiente a las aperturas de los pulverizadores). La exploración se realiza con ruido de Ornstein–Uhlenbeck configurado con $\sigma = 0{,}2$ y $\theta = 0{,}15$ (`ou_stddev=0.2`, `ou_damping=0.15`).

**Configuración del entrenamiento.** La política se entrenó durante 200 episodios, inicializando el *replay buffer* con 1000 pasos aleatorios. Cada episodio constó de 4000 pasos

con paso de integración $ts = 1\,\text{s}$ (duración máxima por episodio: 4000 s).

**Resultados del entrenamiento.** La Figura 4.27 muestra la evaluación final del agente. Se representan las acciones (*opening1*, *opening2*), las señales de proceso y la recompensa. Las variables bajo control son `outp_T_SH3_out` y `outp_Delta_T_SH2_out_T_SH3_in`, ambas mostradas junto con sus consignas; las temperaturas `outp_TSH2_in` y `outp_T_SH2_out` se incluyen solo con carácter informativo. A lo largo del horizonte evaluado, el agente mantiene las consignas con error estacionario reducido y acciones acotadas, con desviaciones transitorias moderadas al comienzo.

En conjunto, los resultados obtenidos en la línea de sobrecalentamiento confirman que `RLIC-FMU` es aplicable a escenarios industriales realistas. El marco entrenó con éxito un agente DDPG sobre la FMU del *superheater* desarrollada en ClaRa+, integrando TF-Agents y exportando la política a TensorFlow para su evaluación. La política podría evaluarse en un entorno Modelica mediante SMArtInt; no obstante, en este caso debería hacerse en Dymola, ya que la FMU fue generada con ese entorno y su ejecución requiere licencia. Al no disponer de dicha licencia, no se ha realizado esa prueba en Modelica. Esto se alinea con los objetivos específicos acordados con XRG (desarrollo del entorno, casos de complejidad creciente y validación) y respalda la viabilidad del enfoque para control multivariable acoplado y su *benchmarking* frente a controladores PI en aplicaciones reales.

## 4.4. Evaluación de políticas de RL en OpenModelica mediante SMArtInt

En esta última sección se presenta la evaluación de las políticas entrenadas de aprendizaje por refuerzo dentro de un modelo desarrollado en OpenModelica, utilizando la biblioteca SMArtInt (*Simple Modelica Artificial Intelligence Interface Library*), desarrollada por XRG Simulation GmbH. Esta integración constituye la fase final del trabajo y permite demostrar el uso práctico de los agentes inteligentes entrenados sobre sistemas físicos simulados.

### 4.4.1. Conversión de la política a TensorFlow Lite

Una vez entrenados los agentes con TensorFlow Agents, las políticas se guardaron empleando la clase `policy_saver.PolicySaver`. Sin embargo, para su ejecución dentro de

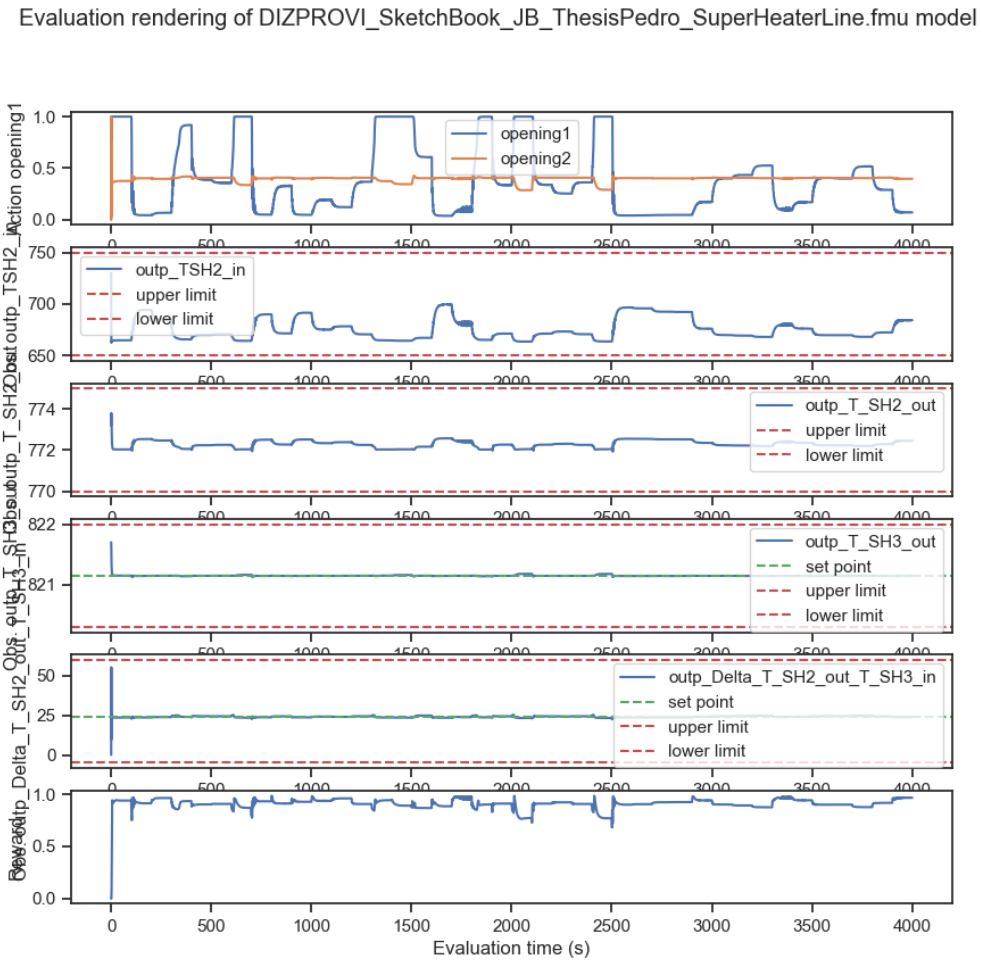


Figura 4.27: Evaluación final del agente DDPG en la línea de sobrecalentamiento: acciones (arriba), señales de proceso con las variables controladas `outp_T_SH3_out` y `outp_Delta_T_SH2_out_T_SH3_in` y sus consignas (centro), y recompensa (abajo). El controlador inteligente consigue mantener la temperatura de salida y el salto térmico entre etapas con errores estacionarios mínimos.

entornos de simulación como OpenModelica fue necesario convertir los modelos a formato TensorFlow Lite.

Esta conversión responde a dos motivos principales:

- Reducir el tamaño y la complejidad del modelo, lo cual facilita su ejecución en tiempo de simulación.
- Garantizar la compatibilidad con la biblioteca SMArtInt, que soporta de manera nativa modelos en formato Lite.

No obstante, el uso de TensorFlow Lite implica ciertas limitaciones, como la pérdida parcial de precisión numérica o la imposibilidad de utilizar algunas operaciones avanzadas disponibles en TensorFlow estándar. Estas limitaciones se tuvieron en cuenta en el diseño del agente y en el preprocesamiento de las señales.

### 4.4.2. Integración mediante la biblioteca SMArtInt

La biblioteca SMArtInt actúa como interfaz entre modelos de inteligencia artificial y entornos de simulación en Modelica. Para evaluar la integración de un controlador inteligente entrenado con `RLIC-FMU` para el entorno `CartPole`, se desarrolló el modelo de prueba `CartPoleDQNTest.mo`, cuyo diagrama se muestra en la Figura 4.28. Este modelo integra la política entrenada mediante el bloque `EvaluateSimpleFeedForwardNeuralNetwork` de SMArtInt, que permite cargar directamente el archivo `.tflite` de TensorFlow Lite.

El modelo incluye explícitamente el bloque `CartPole`, que recibe como entrada una fuerza $F$ aplicada al carro y proporciona cuatro salidas correspondientes a las observaciones empleadas durante el entrenamiento: `pos_cart`, `v`, `phi_pole` y `w_pole`. Estas cuatro señales deben ser enviadas como entradas del modelo TensorFlow Lite, mientras que la salida de la política se interpreta como la acción (fuerza) que cierra el bucle de control sobre el sistema, aunque se trata de una señal de índices discretos, ya que la conversión a señal continua durante el entrenamiento se realizaba en el entorno personalizado de `RLIC-FMU`, el cual no se exporta en el archivo `.tflite`.

Dado que el controlador inteligente y el modelo del sistema físico no pueden conectarse de forma directa (por diferencias en escalas, discretización y formato de señales), se diseñaron bloques adicionales y se emplearon bloques estándar de la *Modelica Standard Library* para adaptar las interfaces. En particular:

- `Normalize01`: bloque diseñado para normalizar y desnormalizar señales, escalando las observaciones a los rangos utilizados en el entrenamiento.
- `DiscreteIndexToForce`: bloque diseñado para convertir índices discretos de acción en valores de fuerza continua aplicados al carro.
- **Muestreo y retención (*sample-and-hold*):** se implementaron cuatro pares de bloques para muestrear y mantener constantes las observaciones entre pasos de control, garantizando estabilidad y sincronización:
  - `Modelica.Blocks.Discrete.Sampler`
  - `Modelica.Blocks.Discrete.ZeroOrderHold`

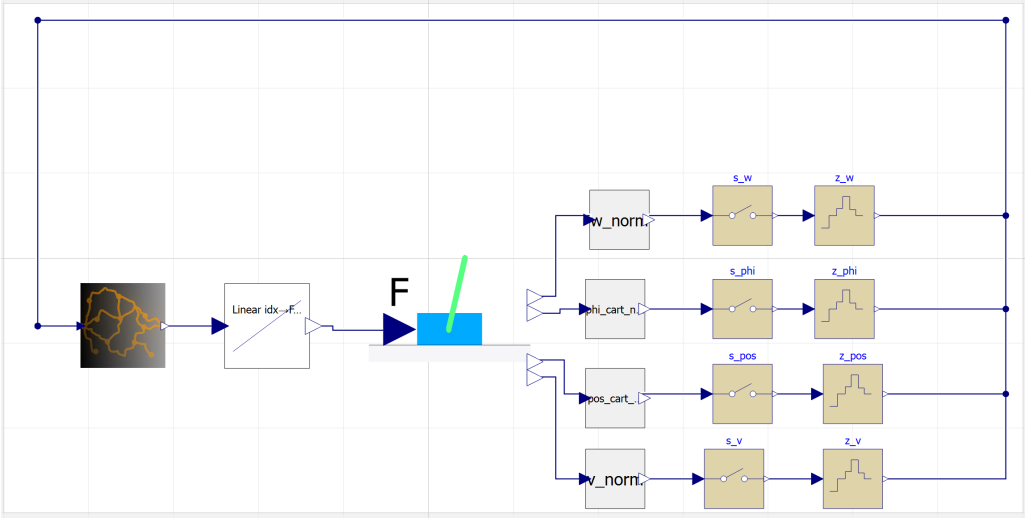


Figura 4.28: Modelo de integración del controlador inteligente en OpenModelica mediante SMArtInt. El bloque `CartPole` proporciona las cuatro observaciones que alimentan al modelo TensorFlow Lite de la política, cuya salida discreta se convierte en una señal de fuerza continua gracias al bloque `DiscreteIndexToForce`. Se incluyen además bloques de normalización y estructuras de *sample-and-hold* para garantizar compatibilidad y estabilidad en la simulación.

### 4.4.3. Resultados de la simulación

Finalmente, se validó el comportamiento de la política RL integrada en el modelo físico mediante OpenModelica y la librería SMArtInt. Para esta evaluación, se emplearon condiciones no vistas durante el entrenamiento: el ángulo inicial del péndulo se estableció

en 75° (valor fuera de los límites definidos en el *specification array*) y la masa del carro se incrementó a 12 kg (frente a los 10 kg usados en entrenamiento).

Las Figuras 4.29 y 4.30 muestran la evolución de las principales variables de salida bajo el control del agente entrenado.

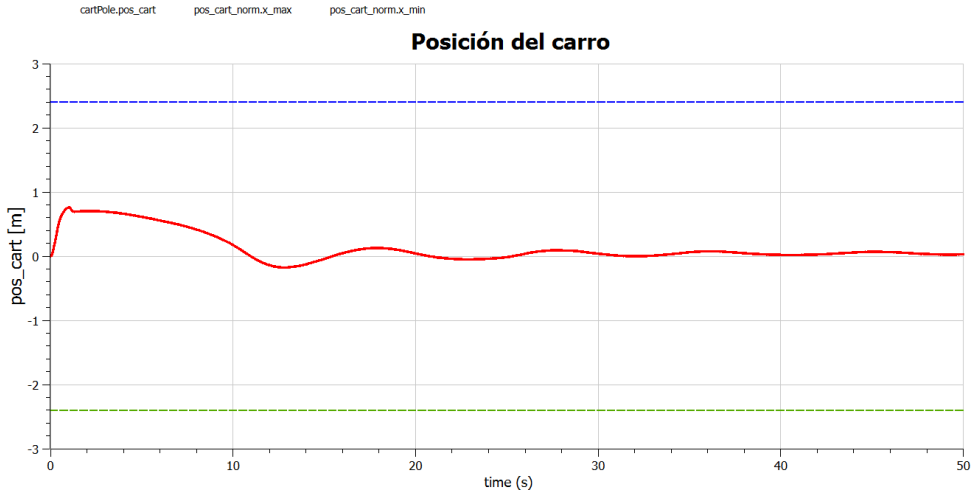


Figura 4.29: Evolución de la posición del carro con la política RL integrada en OpenModelica.

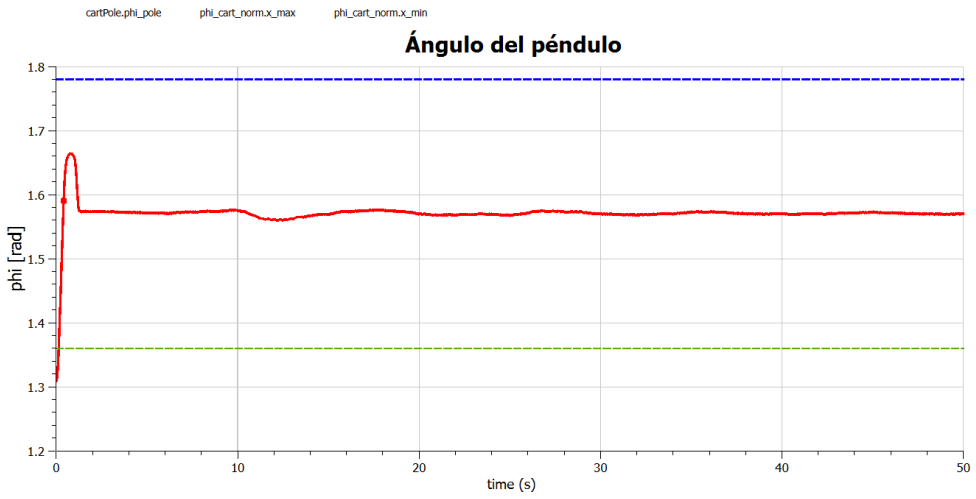


Figura 4.30: Evolución del ángulo del péndulo con la política RL integrada en OpenModelica.

Los resultados demuestran que la política entrenada es capaz de interactuar correctamente con el sistema físico simulado, aplicando acciones en tiempo real y manteniendo el control estable de las variables clave. Esto valida la robustez del flujo de trabajo propuesto, desde el entrenamiento del agente en Python hasta su despliegue en un entorno

de simulación física multidominio.

OpenModelica ofrece animaciones integradas para visualizar el comportamiento dinámico de los modelos. Estas animaciones se habilitan de forma especialmente sencilla cuando el modelo está construido con la biblioteca `Modelica.Mechanics.MultiBody`, cuyos componentes (cuerpos, juntas, fuerzas y visualizadores 3D) incluyen propiedades gráficas que permiten generar la animación automáticamente. El modelo `World` define el marco de referencia, la gravedad y los parámetros visuales por defecto necesarios para la representación. En OpenModelica, una vez ejecutada la simulación y seleccionado el archivo de resultados como activo, la animación puede mostrarse directamente en la interfaz. La Figura 4.31 ilustra esta capacidad mediante varios fotogramas extraídos de la evaluación del `CartPole` en OpenModelica con SMArtInt.

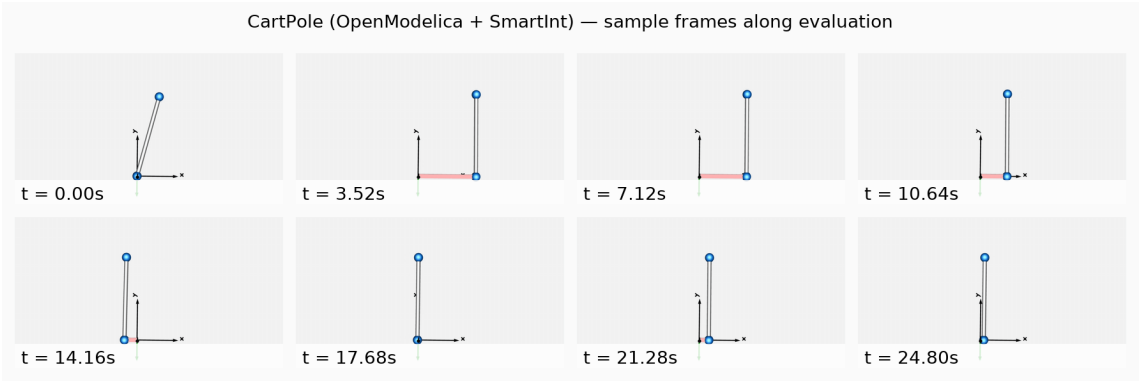


Figura 4.31: Selección de fotogramas de la animación del entorno `CartPole` evaluado en OpenModelica mediante la biblioteca SMArtInt.

La integración de políticas de RL en OpenModelica mediante SMArtInt confirma la viabilidad del enfoque planteado en este trabajo. Esta metodología permite extender el uso de técnicas de aprendizaje automático a sistemas modelados en Modelica, facilitando la exploración de estrategias de control avanzadas en entornos de simulación complejos. Asimismo, el trabajo ha permitido identificar limitaciones y líneas de mejora futuras, como la optimización del tiempo de entrenamiento, la reducción de la complejidad de los modelos exportados y la ampliación a casos industriales de mayor envergadura.

# Capítulo 5

# Conclusiones y Trabajo Futuro

Este trabajo demuestra la viabilidad de entrenar y desplegar controladores inteligentes de aprendizaje por refuerzo profundo (DRL) sobre modelos FMI con `RLIC-FMU`, cubriendo un vacío en el ecosistema Modelica e integrando de forma práctica TF-Agents y PyFMI. La herramienta reduce el esfuerzo de sintonía en sistemas acoplados y acelera la transferencia entre variantes de planta mediante *fine-tuning*.

Este trabajo ha presentado el desarrollo de `RLIC-FMU`, una herramienta software en Python diseñada para entrenar y evaluar agentes de DRL sobre modelos de sistemas físicos exportados en formato FMU bajo el estándar FMI. La propuesta cubre un vacío en el ecosistema Modelica, donde no existían hasta ahora soluciones abiertas que integraran bibliotecas de RL modernas con simulaciones multidominio, y proporciona una infraestructura modular que facilita tanto la investigación como el desarrollo de aplicaciones industriales.

La arquitectura de `RLIC-FMU`, organizada en los paquetes Agents, Core y Environments, permite separar responsabilidades y estandarizar la interacción entre agentes y modelos físicos. La integración nativa con TF-Agents y PyFMI ha permitido implementar entornos personalizados reutilizables y controladores inteligentes entrenados con algoritmos como DQN y DDPG.

Los casos de estudio desarrollados (carro con péndulo invertido, depósito de líquido y sobrecalentador) demuestran la viabilidad de la herramienta: los controladores entrenados logran resultados competitivos frente a controladores clásicos (como PI) y muestran capacidad de generalización frente a perturbaciones y condiciones no vistas. En particular, para el caso del carro con péndulo invertido se ha exportado la política entrenada en formato

TensorFlow Lite y se ha demostrado su uso directo en una simulación de OpenModelica mediante la biblioteca SMArtInt, cumpliendo así uno de los objetivos específicos más relevantes del proyecto. Esto evidencia el potencial del aprendizaje por refuerzo para el control avanzado de sistemas físicos complejos y valida el enfoque seguido en el desarrollo de la herramienta.

Más allá de sus limitaciones actuales —coste computacional, sensibilidad al ajuste y madurez de las herramientas—, una ventaja diferencial del enfoque con RL es la *síntesis automática de controladores*: basta con definir una función de recompensa y disponer de un entorno de simulación fiel para que el entrenamiento genere, con mayor o menor grado de precisión, políticas que *funcionan* sin recurrir a sintonías manuales ni a modelos explícitos del controlador. Esta capacidad de "arranque" reduce el tiempo de ingeniería, facilita trasladar controladores entre variantes de planta (vía *fine-tuning*) y ofrece una alternativa natural para sistemas multivariables con acoplamientos difíciles de tratar con controladores tradicionales.

En conjunto, el trabajo ha cumplido con los objetivos planteados: se ha realizado un estudio exhaustivo del estado del arte, diseñado e implementado un *framework* flexible que integra de manera fluida TF-Agents con el estándar FMI, y validado el enfoque a través de casos de uso progresivamente más complejos. Además, se han identificado limitaciones relevantes, como el coste computacional del entrenamiento, la sensibilidad a la elección de hiperparámetros y los retos asociados al uso de FMU, lo que sienta las bases para un plan claro de evolución de la herramienta.

## Grado de consecución de los objetivos.

1. **Fundamentos y algoritmos de RL:** revisados y aplicados (DQN, DDPG) con resultados positivos (véanse las figuras y tablas de resultados de los casos de estudio).

2. **Análisis de `TF-Agents` y compatibilidad con `SMArtInt`:** integración lograda; políticas exportadas a TensorFlow y demostración en flujo con Modelica (OpenModelica).

3. **Revisión de entornos RL+Modelica:** evaluados (p. ej., ModelicaGym) y justificada la creación de `RLIC-FMU` como marco flexible sobre FMI.

4. **Desarrollo del entorno Python sobre FMU:** `RLIC-FMU` implementado (Agents,

Core, Environments) y validado.

5. **Caso de prueba Cart-Pole:** agente entrenado y política exportada, con demostración de uso en simulación vía `SMArtInt`.

6. **Casos complejos (depósito y superheater):** desempeño competitivo frente a PI y generalización ante perturbaciones (véanse Fig. 4.21 y Fig. 4.27).

7. **Limitaciones y mejoras:** identificadas y concretadas en el apartado de trabajo futuro (coste, hiperparámetros, retos con FMU).

*Datos relevantes.* En los tres casos de estudio, las políticas aprendidas mantuvieron las variables dentro de límites y alcanzaron retornos sostenidos; remítase a las figuras y tablas de cada sección para los detalles cuantitativos, que confirman el comportamiento estable y la mejora frente a referencias clásicas en escenarios con acoplamientos o perturbaciones.

## Trabajo Futuro

A partir de los resultados obtenidos, se proponen varias líneas de investigación y desarrollo para ampliar las capacidades de `RLIC-FMU`:

- **Análisis sistemático de hiperparámetros.** Al igual que otros trabajos en RL, se podría estudiar el impacto de hiperparámetros clave en el aprendizaje, como la arquitectura de las redes neuronales, el tamaño del *batch*, el intervalo de tiempo (*time step*), la tasa de descuento, las estrategias de exploración o los parámetros de optimización.

- **Definición de observaciones enriquecidas.** Investigar si proporcionar al agente información adicional más allá del estado del sistema (por ejemplo, derivadas de las variables, errores acumulados, etc.) mejora la eficiencia del aprendizaje o introduce complejidad innecesaria.

- **Optimización de funciones de recompensa.** Explorar el diseño y ajuste de funciones de recompensa para mejorar la estabilidad y velocidad de convergencia, así como su adaptación dinámica durante el entrenamiento.

- **Integración en simuladores Modelica.** Aunque supone un desafío considerable, una extensión natural sería permitir el entrenamiento de agentes directamente desde

entornos como OpenModelica o Dymola, de forma análoga a la biblioteca SMArtInt de XRG Simulation. Esto simplificaría el flujo de trabajo y abriría nuevas aplicaciones industriales.

- **Paralelización y escalado.** Desarrollar soporte para simulaciones paralelas en CPU/GPU para acelerar entrenamientos, especialmente en modelos de alta fidelidad.
- **Benchmarks y casos de estudio industriales.** Ampliar el número de sistemas de prueba para cubrir procesos industriales reales (energía, automoción, aeronáutica) y establecer métricas de comparación con controladores clásicos y técnicas de control óptimo.
- **Exploración de métodos de RL basados en modelos.** Dado que este enfoque siempre dispone de un modelo físico explícito, es natural investigar algoritmos de RL basados en modelos que aprovechen esta información para mejorar la eficiencia de entrenamiento y reducir la dependencia del muestreo intensivo.
- **Implementación de algoritmos más eficientes en muestreo (p. ej. SAC).** Incluir métodos avanzados como *Soft Actor-Critic* (SAC), que requieren *replay buffers* más sofisticados como `Reverb` de Google DeepMind. Actualmente, `Reverb` tiene soporte oficial únicamente en Linux, lo que limita su adopción en entornos Windows, pero una futura adaptación multiplataforma permitiría explotar completamente estos algoritmos.

En resumen, este trabajo constituye una primera aproximación a la integración de aprendizaje por refuerzo profundo y simulación multidominio basada en Modelica/FMI. La herramienta desarrollada proporciona una base sólida para futuras investigaciones orientadas a acelerar el entrenamiento, mejorar la generalización de los controladores y facilitar la transferencia de estas técnicas a entornos industriales.

# Bibliografía


Cano Lopes, G., Ferreira, M., da Silva Simões, A., & Luna Colombini, E. (2018). Intelligent Control of a Quadrotor with Proximal Policy Optimization Reinforcement Learning. *2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE)*, 503-508. https://doi.org/10.1109/LARS/SBR/WRE.2018.00094

Chang, G. C., Luh, J. J., Liao, G. D., Lai, J. S., Cheng, C. K., Kuo, B. L., & Kuo, T. S. (1997). A neuro-control system for the knee joint position control with quadriceps stimulation. *IEEE transactions on rehabilitation engineering : a publication of the IEEE Engineering in Medicine and Biology Society*, *5*, 2-11.

Cisneros, M., & Leal, R. (2001). *Application Of Several Neurocontrol Schemes To A 2-DOF Manipulator.*

Diederichs, E. (2019). Reinforcement Learning - A Technical Introduction. *Journal of Autonomous Intelligence*, *2*, 25. https://doi.org/10.32629/jai.v2i2.45

Dong, H., Zhang, J., & Zhao, X. (2021). Intelligent wind farm control via deep reinforcement learning and high-fidelity simulations. *Applied Energy*, *292*, 116928. https://doi.org/https://doi.org/10.1016/j.apenergy.2021.116928

Dong, H., & Zhao, X. (2022). Composite Experience Replay-Based Deep Reinforcement Learning With Application in Wind Farm Control. *IEEE Transactions on Control Systems Technology*, *30*(3), 1281-1295. https://doi.org/10.1109/TCST.2021.3102476

Elmqvist, H., Otter, M., & Mattsson, S. (1998). Modelica—An International Effort to Design an Object-Oriented Modeling Language. *Paper presented at 1998 Summer Simulation Conference, 1998/07/15.*

Fernandez-Gauna, B., Graña, M., Osa-Amilibia, J.-L., & Larrucea, X. (2022). Actor-critic continuous state reinforcement learning for wind-turbine control robust optimiza-

tion. *Information Sciences*, *591*, 365-380. https://doi.org/https://doi.org/10.1016/j.ins.2022.01.047

Haarnoja, T., Zhou, A., Ha, S., Tan, J., Tucker, G., & Levine, S. (2018). Learning to Walk via Deep Reinforcement Learning. *arXiv preprint arXiv:1812.11103.*

Konda, V., & Tsitsiklis, J. (1999). Actor-Critic Algorithms. En S. Solla, T. Leen & K. Müller (Eds.), *Advances in Neural Information Processing Systems*. MIT Press. https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

KrishnaKumar, K., Rickard, S., & Bartholomew, S. (1995). Adaptive neuro-control for spacecraft attitude control [Control and Robotics, Part II]. *Neurocomputing*, *9*(2), 131-148. https://doi.org/https://doi.org/10.1016/0925-2312(94)00062-W

Li, F., Wang, Y., Wu, F., Huang, Y., Liu, Y., Zhang, X., & Ma, M. (2020). Review of Real-time Simulation of Power Electronics. *Journal of Modern Power Systems and Clean Energy*, *8*(4), 796-808. https://doi.org/10.35833/MPCE.2018.000560

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. En Y. Bengio & Y. LeCun (Eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. http://arxiv.org/abs/1509.02971

Linkens, D., & Nyongesa, H. (1996). Learning systems in intelligent control: an appraisal of fuzzy, neural and genetic algorithm control applications. *IEE Proceedings - Control Theory and Applications*, *143*(4), 367-386(19). https://digital-library.theiet.org/content/journals/10.1049/ip-cta_19960392

Lukianykhin, O., & Bogodorova, T. (2019). *ModelicaGym: Applying Reinforcement Learning to Modelica Models*. Association for Computing Machinery. https://doi.org/10.1145/3365984.3365985

Lundberg, K. H., & Barton, T. W. (2010). History of Inverted-Pendulum Systems [8th IFAC Symposium on Advances in Control Education]. *IFAC Proceedings Volumes*, *42*(24), 131-135. https://doi.org/https://doi.org/10.3182/20091021-3-JP-2009.00025

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassa-

bis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533. https://doi.org/10.1038/nature14236

Modelon. (2024). FMI Standard: Understand the two types of Functional Mock-up Units – CS v. ME. https://modelon.com/blog/fmi-functional-mock-up-unit-types/

OpenAI. (2018). *Spinning Up in Deep Reinforcement Learning, Part 2: Kinds of RL Algorithms* [Accessed: 2025-09-04]. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Passino, K. (1993). Bridging the Gap between Conventional and Intelligent Control. *IEEE Control Systems Magazine*, *13*(3), 12-18. https://doi.org/10.1109/37.214940

Pigott, A., Baker, K., Dorado-Rojas, S., & Vanfretti, L. (2022). Dymola-Enabled Reinforcement Learning for Real-time Generator Set-point Optimization. *2022 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, 1-5. https://doi.org/10.1109/ISGT50606.2022.9817464

Sierra-García, J. E., & Santos, M. (2021). Improving Wind Turbine Pitch Control by Effective Wind Neuro-Estimators. *IEEE Access*, *9*, 10413-10425. https://doi.org/10.1109/ACCESS.2021.3051063

Sierra-García, J. E., & Santos, M. (2020). Exploring Reward Strategies for Wind Turbine Pitch Control by Reinforcement Learning. *Applied Sciences*, *10*(21). https://doi.org/10.3390/app10217462

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, I–387–I-395.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press.

Tesauro, G. (1994). TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, *6*(2), 215-219. https://doi.org/10.1162/neco.1994.6.2.215

Tomin, N., Kurbatsky, V., & Guliyev, H. (2019). Intelligent Control of a Wind Turbine based on Reinforcement Learning. *2019 16th Conference on Electrical Machines, Drives and Power Systems (ELMA)*, 1-6. https://doi.org/10.1109/ELMA.2019.8771645

Urquía, A., & Martín, C. (2018). *Modelado orientado a objetos y simulación de sistemas físicos.* (2nd). UNED (Universidad Nacional de Educación a Distancia).

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards* (Tesis doctoral). University of Cambridge.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3), 279-292. https://doi.org/10.1007/BF00992698

Werbos, P. J. (1992). Neurocontrol and Supervised Learning: An Overview and Evaluation. En D. A. White & D. A. Solfge (Eds.), *Handbook of Intelligent Control: neural, fuzzy adaptive approaches* (pp. 65-89). Van Nostrand Reinhold.

Wrede, K., Huang, C., Wohlfahrt, T., & Hartmann, N. (2024). FMUGym: An Interface for Reinforcement Learning-based Control of Functional Mock-up Units under Uncertainties. https://doi.org/10.24406/publica-4387

Xie, J., Dong, H., Zhao, X., & Karcanias, A. (2022). Wind Farm Power Generation Control Via Double-Network-Based Deep Reinforcement Learning. *IEEE Transactions on Industrial Informatics*, *18*(4), 2321-2330. https://doi.org/10.1109/TII.2021.3095563

# Apéndice A

# Modelos Modelica del depósito de líquido

Los siguientes listados muestran la implementación en Modelica del modelo del depósito de líquido empleado como caso de estudio. En el Listado A.1 se presenta la versión base con control PI, utilizada como referencia para el análisis de desempeño. Por su parte, el Listado A.2 incluye perturbaciones aleatorias en el proceso de evaporación, lo que permite evaluar la robustez del controlador en escenarios más realistas.

Listado A.1: Código Modelica correspondiente al modelo del depósito de líquido con controladores PI.

```
model FluidTank
  import SI = Modelica.Units.SI;
  import g = Modelica.Constants.g_n;

  // parameters
  parameter SI.Area A = 2 "tank base cross section-area";
  parameter SI.Area a = 0.1 "tank bottom hole cross-section area";
  parameter Real k_f(unit="kg/(s.V)") =100;
  parameter Real k_P(unit="V/m") = 2;
  parameter Real k_I(unit="m.s/V") = 15;
  parameter Real kT_P(unit="V/K") = 0.3;
  parameter Real kT_I(unit="K.s/V") = 50;
  parameter SI.Density rho = 760;
  parameter SI.SpecificHeatCapacity Cp0 = 446;
  parameter Real Cp1(unit="J/(kg.K2)") = 5.36;
  parameter Real k_c(unit="W/V") = 8E+6 "heater parameter";
  parameter SI.Temperature Tin = 300 "liquid temperature at the source";
```

```
18    parameter SI.SpecificHeatCapacity Cp_in = Cp0 + Cp1*Tin "specific heat
          capacity of liquid at source conditions";
19
20    // variables
21    SI.Mass m(start=1e3, fixed=true) "mass of liquid in the tank";
22    SI.Height h "liquid level";
23    SI.MassFlowRate Fout "out mass flowrate";
24    SI.MassFlowRate Fin "in mass flowrate";
25    SI.Length e "calculated error at the level controller";
26    Real I(unit="m.s", start=0, fixed=true) "integral error";
27    SI.Voltage uh "level controller output voltage";
28    SI.Height href "level setpoint value";
29    SI.Temperature Tref "temperature setpoint";
30    SI.Temperature T(start = 300, fixed = true) "fluid temperature at the tank"
          ;
31    SI.HeatFlowRate Q "heat flowrate from heating element to fluid";
32    SI.SpecificHeatCapacity Cp "specific heat capacity of the liquid";
33    SI.TemperatureDifference eT "calculated error at temperature controller";
34    Real I_T(unit="K.s", start=0, fixed=true) "temperature error (eT) integral"
          ;
35    SI.Voltage uT "control input to heating element";
36    SI.EnthalpyFlowRate F_H_in, F_H_out "enthalpy flows";
37    SI.Enthalpy H "Total Enthalpy of liquid at tank";
38
39  equation
40
41    // Mass balance
42    der(m) = Fin - Fout;
43    Fin = max(0,k_f*uh); //Fin = if uh>0 then k_f*uh else 0;
44    Fout = a*rho*sqrt(2*g*h);
45    m = rho*A*h;
46
47    // Liquid level PI controller
48    href = if time<5*60 then 5 elseif time<10*60 then 3 else 7;
49    e = href - h;
50    der(I) = e;
51    uh = k_P*e + I/k_I;
52
53    // Temperature PI controller
54    Tref = if time < 500 then 340 else 320;
55    eT = Tref - T;
56    der(I_T) = eT;
```

```
  uT = kT_P*eT + I_T/kT_I;

  // Energy balance
  Cp = Cp0 + Cp1 * T;
  F_H_in = Fin*Cp_in*Tin;
  F_H_out = Fout*Cp*T;
  Q = max(0, k_c*uT);
  Q + F_H_in - F_H_out = der(H);
  H = m*Cp*T;

end FluidTank;
```

Listado A.2: Código del modelo Modelica del depósito de líquido controlado mediante un regulador PI, ejecutado en un entorno con perturbaciones aleatorias de evaporación.

```
model FluidTankWithNoisePIcontrol
  import SI = Modelica.Units.SI;
  import g = Modelica.Constants.g_n;
  import Generators = Modelica.Math.Random.Generators;

  // parameters
  parameter SI.Area A = 2 "tank base cross section-area";
  parameter SI.Area a = 0.1 "tank bottom hole cross-section area";
  parameter Real k_f(unit="kg/(s.V)") =100;
  parameter Real k_P(unit="V/m") = 2;
  parameter Real k_I(unit="m.s/V") = 15;
  parameter Real kT_P(unit="V/K") = 0.3;
  parameter Real kT_I(unit="K.s/V") = 50;
  parameter SI.Density rho = 760;
  parameter SI.SpecificHeatCapacity Cp0 = 446;
  parameter Real Cp1(unit="J/(kg.K2)") = 5.36;
  parameter Real k_c(unit="W/V") = 8E+6 "heater parameter";
  parameter SI.Temperature Tin = 300 "liquid temperature at the source";
  parameter SI.SpecificHeatCapacity Cp_in = Cp0 + Cp1*Tin "specific heat
      capacity of liquid at source conditions";

  // Global parameters
  parameter Modelica.Units.SI.Period samplePeriod = 10
    "Sample period for the generation of random numbers";
  parameter Integer globalSeed2 = 30020
    "Global seed to initialize random number generator";

// Random number generators with exposed state
```

```
28    parameter Integer localSeed = 614657;
29    parameter Integer localSeed_h = 498682;
30
31    // variables
32    SI.Mass m(start=1e3, fixed=true) "mass of liquid in the tank";
33    SI.Height h "liquid level";
34    SI.MassFlowRate Fout "out mass flowrate";
35    SI.MassFlowRate Fin "in mass flowrate";
36    SI.MassFlowRate ml "mass evaporation";
37    SI.Length e "calculated error at the level controller";
38    Real I(unit="m.s", start=0, fixed=true) "integral error";
39    SI.Voltage uh "level controller output voltage";
40    SI.Height href "level setpoint value";
41    SI.Temperature Tref "temperature setpoint";
42    SI.Temperature T(start = 300, fixed = true) "fluid temperature at the tank"
          ;
43    SI.HeatFlowRate Q "heat flowrate from heating element to fluid";
44    SI.HeatFlowRate Ql;
45    SI.SpecificHeatCapacity Cp "specific heat capacity of the liquid";
46    SI.TemperatureDifference eT "calculated error at temperature controller";
47    Real I_T(unit="K.s", start=0, fixed=true) "temperature error (eT) integral"
          ;
48    SI.Voltage uT "control input to heating element";
49    SI.EnthalpyFlowRate F_H_in, F_H_out "enthalpy flows";
50    SI.Enthalpy H "Total Enthalpy of liquid at tank";
51    SI.Temperature Tamb "ambient temperature";
52    output Real unifRndNo "Random number generated with Xorshift64star";
53    output Real wiener "Wiener Stochastic Process";
54    inner Modelica.Blocks.Noise.GlobalSeed globalSeed;
55    Modelica.Blocks.Noise.NormalNoise noise(
56      samplePeriod=samplePeriod,
57      mu=0,
58      sigma=1,
59      useAutomaticLocalSeed=false);
60  protected
61    discrete Integer state64[2](   each start=0, each fixed = true);
62
63  algorithm
64    when initial() then
65      // Generate initial state from localSeed and globalSeed
66      state64  := Generators.Xorshift64star.initialState(  localSeed_h,
          globalSeed2);
```

```
67      unifRndNo  := 0;
68      wiener     := 0;
69    elsewhen sample(0,samplePeriod) then
70      (unifRndNo,  state64)   := Generators.Xorshift64star.random( pre(state64)
          );
71      // Wiener stochastic process generation for ambient Temperature
          simulation
72      wiener := pre(wiener) + noise.y*sqrt(samplePeriod);
73    end when;
74
75  equation
76
77    // Mass balance
78    der(m) = Fin - Fout - ml;
79    //ml=max(0,unifRndNo*20);
80    ml=(T-Tamb)*unifRndNo*2.0;
81    Fin = max(0,k_f*uh); //Fin = if uh>0 then k_f*uh else 0;
82    Fout = a*rho*sqrt(2*g*h);
83    m = rho*A*h;
84
85    // Liquid level PI controller
86    href = if time<5*60 then 5 elseif time<10*60 then 3 else 7;
87    e = href - h;
88    der(I) = e;
89    uh = k_P*e + I/k_I;
90
91    // Temperature PI controller
92    Tref = if time < 500 then 340 else 320;
93    eT = Tref - T;
94    der(I_T) = eT;
95    uT = kT_P*eT + I_T/kT_I;
96
97    // Energy balance
98    Cp = Cp0 + Cp1 * T;
99    F_H_in = Fin*Cp_in*Tin;
100   F_H_out = Fout*Cp*T + ml*Cp*T;
101   Q = max(0, k_c*uT);
102   Q - Ql + F_H_in - F_H_out = der(H);
103   Ql =(T - Tamb)*0;
104   H = m*Cp*T;
105   Tamb = 293.15 + wiener*0.1;
106
```

```
107 end FluidTankWithNoisePIcontrol;
```

# Apéndice B

# Ejemplo de script de entrenamiento con `RLIC-FMU`

El Listado B.1 muestra el script `cart_pole_dqn.py`, que ejemplifica el flujo completo de entrenamiento de un agente DQN sobre el modelo `CartPole` desarrollado en este trabajo. Este archivo configura el entorno, inicializa el agente, ejecuta el proceso de entrenamiento y guarda la política entrenada para su posterior evaluación.

Listado B.1: Script de entrenamiento `cart_pole_dqn.py`.

```python
# examples/training/cart_pole_dqn.py

"""
Cart-Pole DQN training script.

Trains a Deep Q-Network (DQN) agent on the Cart-Pole FMU environment using
the RLIC-FMU framework. Configures environment parameters, defines the agent
and network architecture, runs training, and saves the trained policy.
"""

import logging
from pathlib import Path
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks")

from rlic_fmu.environments import EnvironmentConfig
from rlic_fmu.core import ExperimentRL

```

```
# folder name for saving the trained policy
policy = "cart_pole_DQN"

def main():
    # enable interactive plotting for live updates
    plt.ion()
    training_figure, training_ax = plt.subplots(2,1,figsize=(10, 8))

    # create environment config dictionary
    config = EnvironmentConfig(parameter_dict = None,
                               ts = 0.01,               # simulation step
                                   size (s)
                               steps_max = 1000,        # max steps per
                                   episode
                               display = True,
                               negative_reward = 0,
                               log_level=logging.DEBUG,
                               discretize=True,         # discretize
                                   continuous actions
                               num_actions=7,           # odd number so force
                                    includes 0
                               fmu_silent_mode=True).get_config()

    # for debugging turn it to `False` (training slows down though)
    use_tf_functions = True

    # define agent and experiment
    tf_agent_config = {
        'tf_agent_name': "DQN",
        'fc_layer_params': (100, 100, 100, 50)}
    model_name = "Cart Pole"
    num_eval_episodes = 1

    experimentDQN = ExperimentRL(
        tf_agent_config,
        model_name,
        config,
        num_eval_episodes,
        training_figure,
        training_ax,
        initial_collect_steps=10000,
        use_tf_functions = use_tf_functions
```

```
58      )
59
60      # train the agent
61      num_episodes=1000
62      trained_agent, episodes_length, exec_time = experimentDQN.train(
63          num_iterations=10000000,
64          training_end="episodes",
65          max_num_episodes=num_episodes,
66          eval_interval=100,
67          )
68
69      # save the trained policy
70      policy_dir = Path(__file__).resolve().parents[2] / "trained_policies" /
            policy
71      experimentDQN.save_policy(trained_agent.policy, policy_dir)
72
73  if __name__ == "__main__":
74      main()
```