Universidad Nacional
de Educación a Distancia

Escuela Técnica Superior de
Ingeniería Informática

Universidad
Complutense de Madrid

Facultad de
Informática

Diseño, desarrollo y aplicación de un controlador por aprendizaje por refuerzo para la región de baja velocidad de un aerogenerador

Roberto Díaz Barrio

Director/a: Matilde Santos Peñas
Co-director/a: Jesús Enrique Sierra García

Trabajo de Fin de Máster

Máster Universitario
en Ingeniería de Sistemas y de Control

Curso 2024-2025/Convocatoria de Junio

Diseño, desarrollo y aplicación de un controlador por aprendizaje por refuerzo para la región de baja velocidad de un aerogenerador

Roberto Díaz Barrio

Director/a: Matilde Santos Peñas
Co-director/a: Jesús Enrique Sierra García

Trabajo de Fin de Máster

Máster Universitario
en Ingeniería de Sistemas y de Control

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Roberto Díaz Bamio

Firma del alumno

Agradecimientos

Quisiera agradecer a Matilde y a Enrique su trabajo, sin el cual este proyecto no sería posible.

No puedo olvidarme tampoco de mis compañeros, lo que aquí se muestra es sólo una pequeña parte del gran conocimiento que poseen y que han logrado transmitirme.

Resumen

La energía eólica es una de las principales fuentes de generación en Europa y no ha dejado de crecer en los últimos años. Para contar con la capacidad de generación necesaria para abastecer el mercado en su estado actual, y las previsiones de crecimiento energéticas, es esencial contar con modelos de aerogenerador que funcionen de forma fiable y duradera.

Conjuntando estas necesidades con el crecimiento en los últimos años del campo de inteligencia artificial, y contando especialmente con modelos de aprendizaje por refuerzo, que son capaces de aprender comportamientos complejos que mejoran a los controles existentes, en el futuro se espera que estos comiencen a implantarse en modelos industriales.

Por ello, en este trabajo se diseñó e implementó un modelo de controlador empleando técnicas de aprendizaje por refuerzo, basándose en el esquema Deep Deterministic Policy Gradient. El objetivo principal de este controlador es el de regular un aerogenerador durante su período de trabajo en seguimiento de máxima potencia.Por ello, para su entrenamiento se han creado modelos representativos de una turbina eólica y de viento, con el fin de poder llevar a cabo éste de la forma computacionalmente más óptima. Con la idea de realizar un control que cumpla con múltiples objetivos, se ha desarrollado para el controlador un sistema de recompensas personalizadas y basadas en distintos comportamientos que se consideraron beneficiosos desde el punto de vista de diseño. Además, se han explorado diferentes técnicas y parámetros de entrenamiento que permitieron llevar éste a cabo de un modo satisfactorio.

Por último, se han comparado los resultados del controlador propuesto con una implementación basada en controles clásicos de aerogenerador. Pudiendo verificar que el aprendizaje que éste adquirió durante su entrenamiento le permitió cumplir los objetivos marcados e incluso superar al control de referencia en términos de potencia generada.

Palabras clave: Aprendizaje por refuerzo, Deep Deterministic Policy Gradient, Seguimiento de máxima potencia, Control de aerogeneradores, Modelado de aerogeneradores

Abstract

Nowadays, wind energy is one of the main power sources that exist inside Europe's energy mix, and it has been growing constantly over the years. In order to fulfill the required power generation capacity and possible future grows, it is essential to have wind turbine models that are capable of working with reliability and durability during their lifecycle.

Taking into account these needs, and combining them with the growth of the artificial intelligence environment, specially the reinforcement learning, which brings models which are capable of learning complex behaviours that improve the existing controllers. It is expected that in the near future the wind turbine industry will start to introduce these new brand controllers into their products.

In order to explore these options, in this work a reinfoircement learning controller based on Deep Determinisitc Policy Gradient technique is designed and implemented. The main control target is to ensure the proper operation of a wind turbine during its maximum point power tracking phase. For the sake of training the controller, a wind turbine model and a wind model were developed in order to have an optimal computation environment. The controller was designed to overcome multiple objectives, in order to do so a custom reward system was designed based on the desired behaviours criteria. Furthermore, multiple techniques and training parameters has been explored in order to achieve the proposed goals.

Finally, after a training period, the results of the designed controller has been compared with a benchmark controller that had been implemented based on classical techniques. By using these comparisons the learning acquierd by the controller was no only verified, but it could be seen that it had improved the benchmark controller power generation.

Keywords: Reinforcement Learning, Deep Deterministic Policy Gradient, Maximum Point Power Tracking, Wind turbine control, Wind turbine modeling

Índice general

Índice de figuras

Índice de tablas

Glosario

λ Equivalente a TSR.

C_p Coeficiente de Potencia.

BEMT Blade Element Momentum Theory.

CPU Central Processing Unit.

DDPG Deep Deterministic Policy Gradient.

FEM Finite Element Methods.

GPU Graphics Processing Unit.

LIDAR Laser Imaging Detection And Ranging.

MDP Markov Decision Process.

MPPT Maximum Power Point Tracking.

MSBE Mean-Squared Bellman Error.

OMG Optimal Mode Gain.

PID Proportional Integral Derivative.

PLC Programmable Logical Controller.

PSD Power Spectrum Density.

RL Reinforcement Learning.

TSR Tip Speed Ratio.

Capítulo 1

Introducción

1.1. Motivación

En los últimos años, diversos factores como la creciente preocupación por el medioambiente, las crisis energéticas causadas por conflictos políticos, y las búsquedas de energías o combustibles independientes, han hecho que la proliferación de distintos tipos de energías renovables haya sido notable.

Los principales tipos de energía empleados en la actualidad, y ya con un grado de implantación importante, son la solar y la eólica, alcanzando en 2019 los 651 GW instalados a nivel mundial, incrementando ese año un 10 % respecto a 2018 (Global Wind Energy Council [2020]). Dentro del ámbito institucional de la Unión Europea, debido a las preocupaciones anteriormente mencionadas, se está favoreciendo también este cambio. En el año 2023, se votó la Directiva 2018/2011 que emplaza a los paises miembros a obtener un 42,5 % de su energía de fuentes renovables en 2030, teniendo como objetivo final convertirse en el primer continente climáticamente neutro en el año 2050.

A la hora de redactar este documento, los últimos datos aportados en España por parte del operador de red, Red Eléctrica Española, ilustran esta tendencia ya que la energía eléctrica generada para la red en 2022 provino en un 42,2 % de fuentes de energía renovables (Red Eléctrica Española [2023]), marcando en su año un máximo histórico que ha sido superado en 2023, último año del que se tienen datos. En este año, un 50,3 % de la generación total provino de fuentes renovables, aportando 134321 GWh al sistema, y superando por primera vez a las fuentes no renovables (Red Eléctrica Española [2024]). Los datos de 2023 para los distintos tipos de generación se pueden consultar en la figura 1.1.

Este cambio de tendencia presenta unos grandes desafíos a distintos niveles. El primero es un cambio de paradigma, en el que se está transformando una red centralizada en ciertos nodos con gran capacidad generadora, a otro esquema distino cada vez más distribuido con nodos de menor potencia, lo que implica una mayor dificultad en la gestión de la misma. Además, este tipo de energía en la que la captación se hace de forma local y se vierte a la

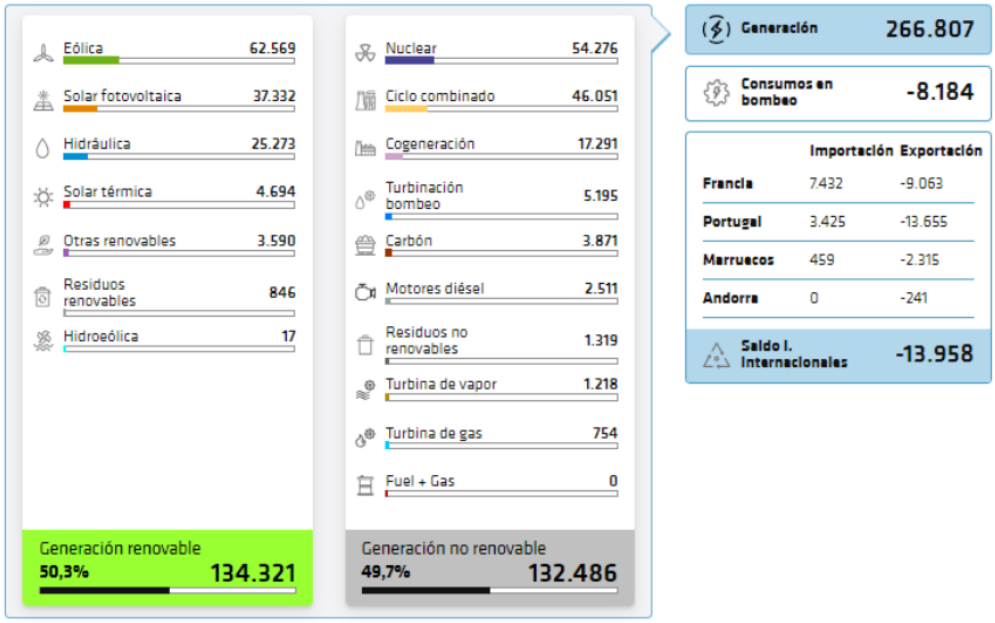

Figura 1.1: Infografía de la generación de la red eléctrica en España recogida en el Informe del Sistema Eléctrico de 2023 (Red Eléctrica Española [2023]).

red o se consume en mismo lugar, ha sido un gran caldo de cultivo para el nacimiento de las denominadas Microrredes (Lasseter [2002]). Estas son una representación a escala de la red eléctrica clásica, que está pensada para integrar la generación proveniente de las energías renovables con consumos cercanos, bien de forma sincronizada con la red principal o funcionando de forma aislada. Bajo este esquema se están desenvolviendo distintas investigaciones, por ejemplo para consumos de baja potencia en hogares y oficinas (Dragicevic et al. [2014]), o para la integración del coche eléctrico en el suministro (Rahman and Hiti [2019]). Debido a la versatilidad de los convertidores, se pueden encontrar en una misma zona distintos tipos de tensión y frecuencias de operación, incluida la operación en corriente continua, lo que aporta numerosas oportunidades pero también ofreciendo nuevos problemas, añadiendo así una capa de complejidad a la problemática y la algoritmia de control (Dragičević et al. [2016a], Dragičević et al. [2016b]).

Respecto a la energía eólica, existen numerosos tipos de aerogeneradores desde el punto de vista de su objetivo. Desde grandes generadores, situados en parques en tierra firme o en las zonas costeras, en los que cada máquina tiene capacidad de varios MW, como el que se puede ver en la figura 1.2; a otros de pequeña o mediana potencia que se pueden instalar de forma solitaria o en pequeños grupos, con el fin de realizar un apoyo a redes aisladas o llevar a cabo un autoconsumo parcial o total. Dado que este mercado está entrando en su madurez, la disputa entre distintos fabricantes está provocando que cada vez la optimización desde el punto de vista económico de las turbinas sea mayor, compitiendo entre ellos desde el punto de vista del coste de mantenimiento y fabricación, fiabilidad, y capacidad generadora. Esto conlleva a que se estén llevando a cabo investigaciones y se estén sacando nuevos productos

al mercado que obedecen a estos objetivos como cajas reductoras (Nejad et al. [2022]) o palas (Wagner, Hermann-Josef [2020]). El diseño de los controladores, por lo tanto, no puede ni debe ser ajeno a esta tendencia y deberá mejorar las capacidades de las máquinas futuras, o incluso de las ya fabricadas, para que éstas puedan tener un comportamiento mejor, lo que desembocará en diseños más baratos que llevarán a menores amortizaciones, y en último caso en una energía limpia y accesible tanto a nivel físico como económico para el futuro (Darrow [2010]).

Figura 1.2: Ejemplo de parque de aerogeneradores situado en Mondoñedo (Lugo, España).

1.2. Propuesta y objetivos

Como se ha mencionado, las líneas de fabricación de los aerogeneradores están tendiendo cada vez más a un óptimo económico. Desde el punto de vista de los controladores esto tiene varias implicaciones. Si bien hace años, con las primeras generaciones, los controladores se diseñaban con el ánimo de tener la máquina siempre lo más cerca posible de los puntos de operación en los que se pudiera extraer la máxima potencia (Bossanyi [2000]), actualmente se está optando por otras alternativas multiobjetivo, que se han demostrado más rentables ya que permiten no sólo obtener una cantidad de energía cercana al óptimo en todo momento, sino que también controlan otros parámetros como las cargas inducidas al dispositivo, o las vibraciones generadas, con el fin de minimizar también las averías y los gastos de mantenimiento, pudiendo alargar incluso el tiempo de vida útil del aparato (Wagner, Hermann-Josef [2020]).

En este trabajo se explorará un control de aprendizaje por refuerzo basado (Reinforcement Learning, RL) en un modelo Deep Deterministic Policy Gradient (DDPG), con el que se buscará hacer un seguimiento efectivo del punto de máxima potencia de un aerogenerador

en su zona de trabajo previa a la nominal, teniendo en cuenta no sólo la generación bruta de potencia sino que se le dará un enfoque multiobjetivo, intentando afrontar otros efectos adversos que puedan surgir como la entrada en zonas de trabajo peligrosas. Para ello, será clave la elaboración de un sistema de recompensas que haga que el modelo identifique su modo de funcionamiento óptimo y descarte comportamientos menos deseables.

Con este fin, se desarrollará un modelo de turbina simplificado, y se le dotará a éste con la alimentación de un viento turbulento representativo realista, creado a partir de un modelo meteorológico. Será además también necesaria la creación de una infraestructura que permita el entrenamiento y la verificación del modelo de control propuesto, así como la creación de numerosos componentes auxiliares que faciliten su gestión.

Para la consecución de estos objetivos, se tomará como referencia el modelo de aerogenerador IEA 15MW (Gaertner et al. [2020]). Un modelo propuesto por la International Energy Agency, y que fue diseñado mediante la colaboración de diversos grupos de investigación, formando parte de un catálogo de representaciones de turbinas eólicas realistas destinadas a ser utilizadas como estándares de referencia en investigaciones, ya que en los modelos comerciales la protección de datos por parte de los fabricantes imposibilita tener un modelo totalmente ajustado con la realidad.

Con el fin de representar los modelos físicos de dispositivo eólico y viento, se empleará el software Matlab/Simulink (The MathWorks Inc. [2024]) en su versión 2024b. Para trabajos que requieran de extracción de datos aerodinámicos del aerogenerador de referencia se utiliza el software OpenFAST (NREL [2025]), desarrollado por el instituto de investigacion National Renewable Energy Laboratory en su versión 3.5.3. Para trabajar con el modelo de controlador, y realizar actividades auxiliares de gestión, se emplea el software Python (van Rossum and Drake [2011]) en su versión 3.12.3. Dentro de Python, y para de gestionar los diferentes apartados necesarios referidos a las redes neuronales, se emplean las librerías de Tensorflow (Abadi et al. [2015]) en su versión 2.19.0, con Keras (Chollet [2015]) como interfaz en su versión 3.9.0.

1.3. Estructura del documento

Este documento se estructura en las siguientes partes. En el capítulo 2, se ofrecerá un breve repaso del estado del arte de los aerogeneradores, comentando las distintas tipoligías existentes, centrándose en las máquinas de eje horizontal, por ser las más comunes y la elegida como referencia. Se hablará de la forma en que estos dispositivos captan la energía eólica, y se hará con ello un repaso de sus diferentes áreas de trabajo y el control que se suele aplicar a cada una de ellas, haciendo especial hincapié en la zona de referencia de seguimiento de la máxima potencia sobre la que se basa este trabajo. Se comentará también los desafíos que implica tener un modelo realista de viento, y se sentarán las bases para obtener uno que sirva al propósito de generar estos durante las simulaciones. Por último, se comentarán los

distintos modelos de simulación que suelen emplearse para emular estas máquinas, desde los complejos modelos aeroelásticos hasta modelos simplificados como el que se propondrá más adelante.

En el capítulo 3, se hará un resumen del estado del arte de los controladores basados en aprendizaje por refuerzo, comentando sus características y particularidades esenciales, como la interacción con los entornos para obtener experiencia, o su fundamentación matemática. Se particularizará esto para el modelo Deep Deterministic Policy Gradient que se empleará, reseñando sus particularidades de planteamiento en el que un Actor realiza acciones y un Crítico valora si éstas han sido buenas o no.

En el capítulo 4, y basándose en las explicaciones de los capítulos previos, se irán desgranando y obteniendo los parámetros necesarios para, en un primer momento, representar fielmente el aerogenerador y el viento que serán necesarios para representar el comportamiento de los modelos físicos destinados a las simulaciones.

En el capítulo 5, se discutirán las cuestiones relativas al diseño e implementación de modelo de control que interactúe con el modelo de aerogenerador creado en el capítulo anterior. Teniendo especial relevancia en este aspecto la creación del sistema de recompensas, y el establecimiento de las parametrizaciones elegidas para el controlador para que este sea capaz de llevar a cabo un entrenamiento efectivo. Se explicará también de forma resumida el flujo de trabajo y de información de las distintas partes con el fin de tener una mejor comprensión del funcionamiento de éste en la práctica.

De esta forma, en el capítulo 6, se presentarán los resultados que arroja el modelo de controlador propuesto después de pasar por el proceso de entrenamiento, verificando su aprendizaje con distintas variables de control. Se propondrá la comparación con un controlador PID tomado como referencia, y se mostrará que el modelo desarrollado es equivalente o incluso superior en algunos ámbitos, lo que validará su aprendizaje.

Por último, en el capítulo 7, se comentarán las conclusiones de este trabajo, y se ofrecerán ideas para posibles nuevos trabajos o líneas de investigación.

1.4. Otros trabajos derivados de este documento

Del estudio requerido para llevar a cabo el trabajo propuesto, se ha generado el conocimiento para poder presentar un artículo relacionado con el control inteligente y la orientación de aerogeneradores que ha sido publicado en las XLV Jornadas de Automática (Barrio et al. [2024]) del año 2024.

Así mismo, está pendiente de revisión y publicación un extracto de este mismo trabajo en la conferencia 7th IFAC Conference on Intelligent Control and Automation Sciences del año 2025.

Capítulo 2

Funcionamiento de un Aerogenerador

Un aerogenerador es una máquina compleja que está formada por diversos equipos y sistemas, entre los que destacan los dominios aerodinámico, mecánico, y eléctrico. Estos dispositivos se basan en transformar la energía cinética inherente al viento en movimiento en energía eléctrica, haciendo rotar un eje acoplado a un motor eléctrico. Estos motores entonces, de la misma forma que se hace con otras centrales de generación movidas por turbinas de otro tipo, generan una potencia que vierten a la red, bien sea la principal o a una red aislada a la que estén conectados. Por lo tanto, para su uso y diseño, se han de emplear diferentes apartados de la física y emplear modelos de simulación que sean capaces de abarcar y combinar estos para tener un resultado de sus interacciones.

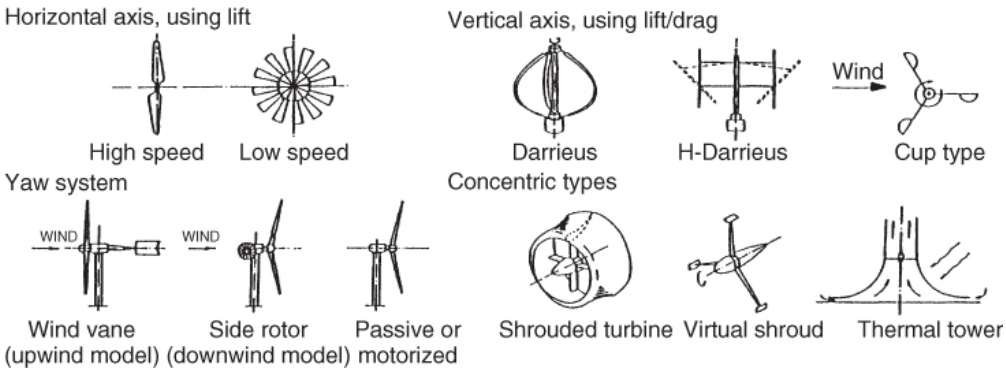

Figura 2.1: Ejemplo de diferentes tipos de turbinas eólicas. Imagen extraída de (Heier [2014]).

Existen diferentes tipos de turbinas eólicas como se puede ver en la figura 2.1, poseyendo cada uno de ellos sus particularidades y principios de funcionamiento (Burton et al. [2021], Heier [2014]). Cuando la tecnología estaba en sus primeras etapas de desarrollo, muchos diseñadores optaron por crear turbinas de eje vertical, por ser este un diseño muy continuista con los molinos de viento presentes en gran parte del mundo desde hace miles de años, y por ser su construcción mucho más sencilla e inmediata que los de eje horizontal. Estas máquinas suelen emplear el principio de drag (o arrastre) en el que la fuerza del viento arrastra el eje que se desea mover, aunque también las hay que emplean el principio de lift

(o sustentación), siendo estas mucho más eficientes a la hora de captar energía. Este tipo de aerogeneradores suele ser de baja potencia y su instalación suele limitarse a instalaciones domésticas, dado que por su construcción tienen una naturaleza en la que la adquisición de energía es independiente de la dirección, no requiriendo de un sistema complejo de medición y orientación. Además, presenta otro tipo de ventajas, como unos requisitos mínimos de mantenimiento, una baja emisión de ruidos, y una buena respuesta ante flujos sucios de viento, lo que permite su montaje en ambientes urbanos.

Otras turbinas que se pueden ver en la imagen 2.1, son las de tipo concéntrico, estas aprovechan un flujo de convección creado por aspiración en una chimenea o un flujo aéreo térmico para mover los unos álabes y de esa forma crear el movimiento del eje; no obstante, este tipo de instalaciones es muy anecdótico y residual, y su instalación se limita más a la investigación debido a su escaso rendimiento comparativo.

Por último, las máquinas predominantes en la generación industrial y en la producción a potencias medias, son los molinos de eje horizontal. Estas máquinas emplean el principio de lift para hacer girar un motor eléctrico situado sobre una torre. Por sus características constructivas, en las que las palas se sitúan perpendiculares al flujo de viento, estas son capaces de convertir la energía eólica en eléctrica de una forma mucho más eficiente ya que en todo momento estas están trabajando en un punto mucho más cercano al óptimo sin contar con parte del ciclo muerto durante una rotación como pasa con las de eje vertical. Además, al realizarse el montaje en torres, aprovechan vientos mucho más rápidos y con más energía que otras que se sitúan cerca de los suelos. Este trabajo se centra en este último tipo, y en las siguientes subsecciones se resumirán brevemente los principios de funcionamiento y de trabajo de esta clase de aerogeneradores; por lo tanto, cabe destacar que a partir de este momento las menciones que se hagan a los generadores eólicos sin especificar un tipo en concreto, han de tomarse como referidas a este tipo de máquinas y no a los otros expuestos.

2.1. Funcionamiento de un aerogenerador de eje horizontal.

Como ya se mencionó, los aerogeneradores de eje horizontal emplean el principio aerodinámico de lift para crear un par de rotación sobre un eje mecánico, que transmite la fuerza de rotación a un generador eléctrico, generando entonces potencia aprovechable para la red de distribución o el autoconsumo. Las partes más relevantes de este se muestran en la figura 2.2, y se comentarán a continuación dando una pincelada de su función. Las fuerzas aerodinámicas se captan en un rotor compuesto por palas que, normalmente, en aerogeneradores grandes, están formadas por perfiles aerodinámicos similares a los de las alas de los aviones. Estas palas suelen estar hechas de fibras y resinas para hacerlas ligeras y, al mismo tiempo, flexibles y resistentes a ciclos de fatiga. El número más común de palas suele ser 3,

ya que presenta una relación óptima de coste de fabricación (cuanto mayor es el número de palas, mayor es el coste), y de balance de cargas por simetría.

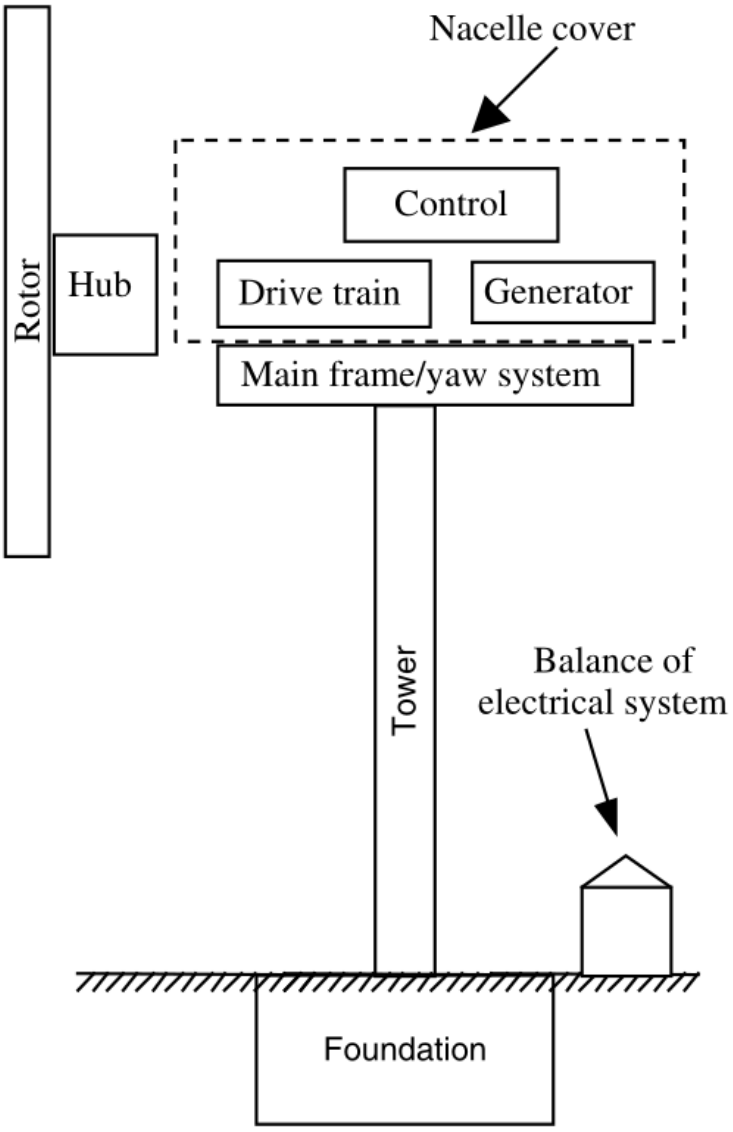

Figura 2.2: Partes de un aerogenerador de eje horizontal. Imagen extraída de (Manwell et al. [2010]).

Las palas se conectan al sistema mecánico mediante rodamientos que van anclados a un hub (o buje), que suele ser una pieza de acero recubierta por fibras, y en la que se pueden integrar sistemas de accionamiento de las palas para su colocación, así como instrumentos de medida. Este hub, es rotativo y va unido a un eje mecánico que se conecta a un generador, bien de forma directa (conexión Direct Drive), o con una multiplicadora (conexión Drive Train). Esta conexión, depende del número de polos y de la tipología elegida en el generador eléctrico, y si bien las máquinas con multiplicadora tienen un menor rendimiento mecánico y son

menos fiables que las de conexión directa, estas presentan otras ventajas como generadores más compactos y unas corrientes de salida menores al tener la posibilidad de trabajar con velocidades mayores y, por lo tanto un menor par nominal (Ackermann [2012]).

El conjunto anterior, suele denominarse nacelle (o góndola). En ella además de los componentes principales descritos en el párrafo anterior, suelen encuadrarse otros subsistemas como ventilaciones para disipar el calor; sensores que permiten conocer el estado de la máquina tales como acelerómetros, termómetros, etc.; equipos de medida que indican la velocidad y dirección del viento; balizas de señalización; y otros sistemas menores. También existe la posibilidad de colocar convertidores de electrónica de potencia que permite la regulación del par del generador, pudiendo maximizar de este modo su eficiencia. Este último es un aspecto clave sobre el que se profundizará en la sección 2.3.1.

Para el correcto funcionamiento de la máquina, y por su seguridad estructural, esta ha de orientarse de cara al viento de la mejor forma posible. Por lo tanto, las nacelles integran un sistema de uno o varios servomotores que les permiten girar sobre su eje para realizar esta función. Esto se conoce en el argot como sistema de yaw.

El bloque anterior, se empotra entonces en una torre que se encarga de sostener estructuralmente la máquina. Estas torres normalmente son tubos de acero de gran altitud para aprovechar la mayor velocidad del viento y una menor cantidad de turbulencia a grandes alturas, ya que este presenta un aire más limpio al carecer de obstáculos. También existen otro tipo de torres como torres de celosía o torres híbridas de acero y hormigón, pero estas son menos comunes y se limitan a modelos de potencias más bajas al tener peor calidad estructural y aerodinámica. En la parte baja de las torres, es frecuente encontrar otros equipos como las interfaces de acceso humano, transformador y celdas (en caso de contar con ellos), o PLCs; aunque cualquiera de ellos puede ir situado en la nacelle.

La torre entonces, irá empotrada al suelo en una cimentación, en caso de ser un aerogenerador situado en tierra; o irá anclada a un pilote o a plataformas flotantes, en caso de ser una máquina situada en el mar.

Cabe destacar que dentro de los aerogeneradores de eje horizontal, existen también varios principios de funcionamiento. Una de las catalogaciones más comunes, es discriminar el dispositivo según si este gira a una velocidad fija o variable. Otra gran distinción, sería en cómo realiza la gestión de la potencia aerodinámica, ya que en un momento dado, tiene que ser capaz de no captar todo el flujo que transporta el viento, dado que un diseño a tal efecto sería sumamente ineficiente por tener que hacer el conjunto demasiado robusto. Si bien hoy en día la mayoría de máquinas tienen velocidad y pitch variables (Apata and Oyedokun [2020]), siendo el pitch la inclinación de la pala para captar más o menos energía, y siendo esta configuración exclusiva en máquinas de cierto tamaño; existen otros principios de regulación como el stall (o pérdida), en los que esta se hace por pérdida en el diseño aerodinámico en los que a partir de ciertas velocidades de viento la pala entra en barrena dejando pasar energía. Esta pérdida puede ser completa en el caso de los sistemas pasivos

o parcial, en caso de los sistemas activos. Se puede ver una comparativa de los perfiles de potencia generada respecto al viento de entrada en función de las estrategias descritas en la figura 2.3, y como se observa, el control activo de pitch proporciona un perfil de potencia no sólo mayor, sino que es más estable en todo el rango, razón por la que es la opción más elegida. Dado que la inmensa mayoría de la bibliografía y de la industria cuando habla de un aerogenerador de eje horizontal se refiere a palas de velocidad de pitch y velocidad variables, así se hará también en este trabajo tomando este esquema como referencia para explicar el funcionamiento de esta sin entrar en las particularidades de otras configuraciones menos comunes.

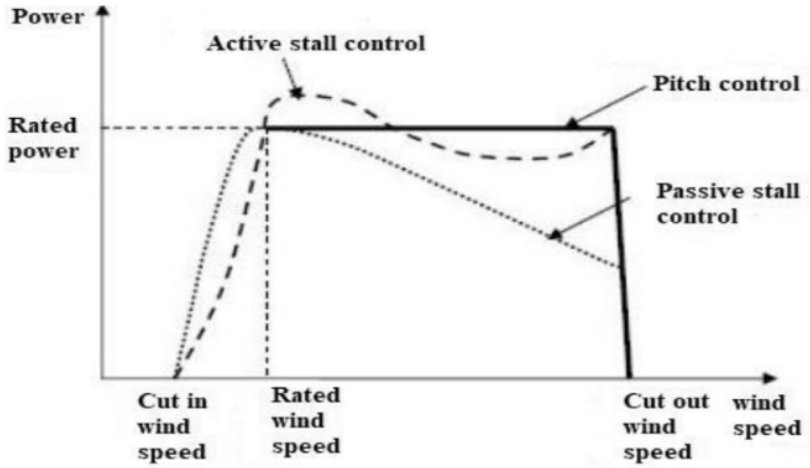

Figura 2.3: Perfil de potencia comparado entre estrategias de pitch y stall. Imagen extraída de (Apata and Oyedokun [2020]).

2.2. Captación y factores aerodinámicos

La teoría que se emplea para el diseño y el cálculo de las fuerzas aerodinámicas que se ejercen sobre un aerogenerador es conocida como Blade Element Momentum Theory (BEMT). Los primeros estudios se derivaron de la aeronáutica empleada por helicópteros dada la similitud en el planteamiento de ambos problemas. Si bien tanto la explicación como la comprensión de esta teoría no son esenciales para la lectura de este trabajo, en el siguiente párrafo se ofrece una breve y simplificada explicación por ser la principal teoría de estudio del movimiento de los aparatos en los que se basa el mismo. En esta explicación se obvian muchos elementos de influencia aerodinámica tales como estelas, vórtices, sombras, u otros de influencia mecánica como fuerzas gravitacionales e inerciales, que complican el problema y su comprensión, y se orienta solamente a entender la causa de por qué se mueve el rotor del aerogenerador.

El BEMT describe las fuerzas que se producen en el rotor del aparato al pasar el viento por él, asumiendo que las fuerzas de lift y drag creadas en los elementos diferenciales de pala por las presiones causadas al pasar el viento, son las responsables del ratio de cambio de los momentos axiales y angulares de todo el viento que pasa por el anillo, despreciando la interacción entre los distintos elementos diferenciales (Burton et al. [2021], Manwell et al. [2010]). Para realizar estos cálculos sin tener que recurrir a modelos extraordinariamente complejos, se asume la simplificacion de que todas las fuerzas que se provocan en un elemento de pala pueden ser calculadas por la media bidimensional de las características que ofrecen los perfiles aerodinámicos según el ángulo de ataque, que viene determinado por la velocidad incidente en el elemento. Entonces la combinación de los términos de velocidad de viento, factores de flujo sobre el perfil, y velocidad de rotacioń de la pala determinarán este ángulo de ataque como se muestra en la figura 2.4, y estos determinarán las presiones ejercidas en la pala, y por lo tanto las fuerzas de arrastre que provocan un par en el eje de rotación.

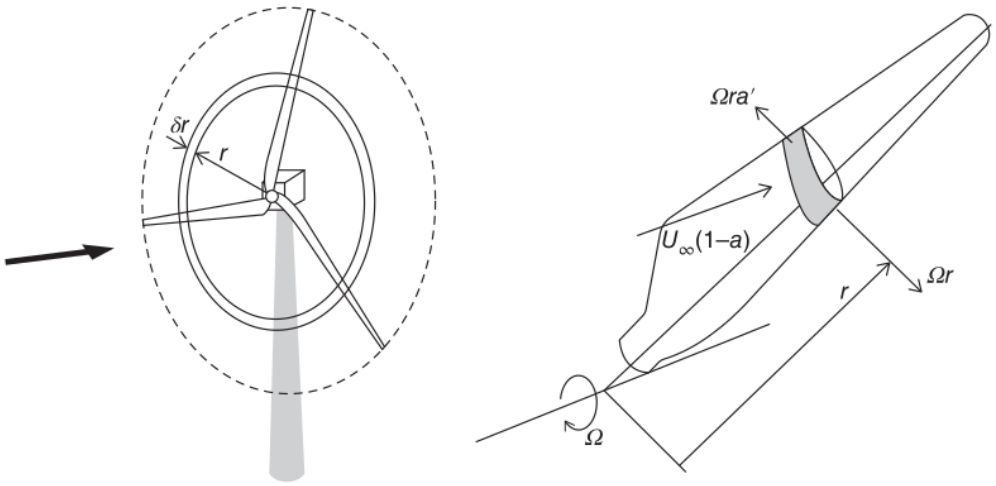

Figura 2.4: Descomposición de velocidades del BEMT. Imagen extraída de (Burton et al. [2021]).

Adoptando una simplificación sobre lo anterior, se puede considerar el rotor de una turbina eólica como disco que extrae la energía del viento. Por lo tanto a medida que el aire pasa a través de él, habrá una caída de presión de forma que, al salir, el viento estará a una presión inferior a la atmosférica y con una velocidad inferior a la de entrada tal y cómo se muestra en la figura 2.5. Entonces, se puede obtener un comportamiento reducido en funcion de ciertos factores que representan la velocidad relativa de la punta de pala respecto al viento (Tip Speed Ratio, TSR), y al ángulo de pitch al que se encuentra la pala, lo que definirá el ataque aerodinámico del viento sobre la misma.

Obteniendo entonces estos parámetros que se derivan de la aerodinámica de las palas, se podrán aproximar de una forma bastante exacta los comportamientos del rotor del aerogenerador, pudiendo obtener una serie de relaciones básicas que forman la base del diseño

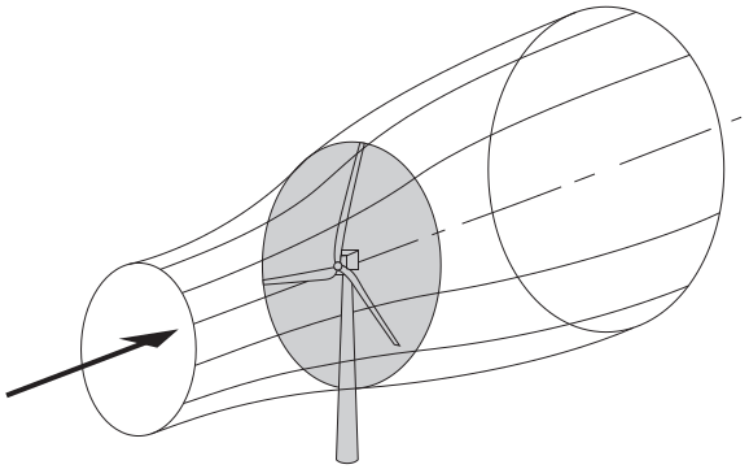

Figura 2.5: Modelo de actuador de disco aerodinámico. Imagen extraída de (Burton et al. [2021]).

de los mismos a todos los niveles, desde la mecánica al control, pasando por el generador y otros elementos.

La primera de estas relaciones describe la potencia aerodinámica que será transpasada como mecánica al eje de la máquina (2.1).

$$P_A = \frac{1}{2}\rho\pi R^2 {v_w}^3 C_p(\lambda, \beta) \tag{2.1}$$

Donde ρ es la densidad del aire, v_w la velocidad del viento, R el radio del rotor (no de la pala), y C_p es el coeficiente de potencia extraíble. Como puede verse, el coeficiente de potencia, es dependiente del TSR o λ, y del ángulo de pitch β, teniendo como máximo teórico un 0,593. Este valor, conocido como límite de Betz, está determinado por parámetros físicos y viene dado por la cantidad óptima de velocidad que puede extraerse al viento; ya que si se extrayera demasiado poca, se estaría dejando escapar potencia y, si por el contrario, se extrae demasiada, el flujo detrás del rotor se frenaría ralentizando la velocidad del que está delante. El TSR, también conocido en la literatura como λ, como se mencionó, es el ratio existente entre la velocidad del viento y la velocidad de punta de pala, y su relación está expresada en la ecuación (2.2), en la que ω es la velocidad del rotor.

$$\lambda = \frac{\omega R}{v_w} \tag{2.2}$$

La segunda de estas relaciones importantes, sería la relación de par presente en la máquina (2.3).

$$\tau_A = \frac{1}{2}\rho\pi R^3 {v_w}^2 C_\tau(\lambda, \beta) \tag{2.3}$$

Donde C_τ es el coeficiente de par de la máquina, que al igual que ocurre con el coeficiente de potencia, depende del TSR y del pitch de la máquina. Como se puede ver mediante una

simple relación física, en la que el par es igual a la potencia entre la velocidad angualar (2.4), esta relación no aporta información nueva respecto a la de la potencia y ambas están relacionadas. No obstante, se emplea en el estudio de casos en el que el conocimiento del par es relevante, por ejemplo en el estudio de los arranques de máquinas grandes.

$$\tau_A = \frac{P_A}{\omega} \tag{2.4}$$

La tercera y última de las relaciones que se extraen con coeficientes extraídos de la aerodinámica, sería la de thrust (o empuje) (2.5).

$$T = \frac{1}{2}\rho\pi R^2 {v_w}^2 C_T(\lambda, \beta) \tag{2.5}$$

Donde T es el empuje del viento sobre la turbina, y C_T es el coeficiente de empuje, dependiente del TSR y del pitch. Esta relación describe la fuerza que se va a recibir en función del punto de trabajo en el que se encuentre la máquina en ese momento, y es importante a nivel estructural para poder pronosticar las fuerzas a las que se tendrá que enfrentar el conjunto mecánico en función de la configuración elegida.

Como puede verse, tanto (2.1), como (2.3), y (2.5), dependen de coeficientes que a su vez son dependientes de los valores de pitch y del TSR; lo que permite conocer de antemano importante información sobre las dinámicas si estos son conocidos. En la siguiente sección 2.3, se explicará el funcionamiento dinámico de la turbina dependiendo del punto de trabajo en el que se encuentre, haciendo especial hincapié en el coeficiente de potencia, ya que es uno de los parámetros principales, sino el principal, de diseño de toda la máquina y uno de los factores más importante en la rentabilidad de las mismas.

Además, como último apunte, puede verse que la potencia crece de forma cúbica con el viento (2.1), mientras que las fuerzas (2.3) y (2.5) lo hacen de forma cuadrática. No es el objeto de este trabajo explicar esta relación, pero es interesante conocer esta información ya que es otro de los factores que influyen en el retorno económico de las máquinas, dado que se pueden diseñar máquinas que capturen enormes cantidades de potencia pero habría que crearlas de forma que resistieran una enorme cantidad de fuerza. Existe por lo tanto, una relación óptima según cada diseño en el que se conjuga un punto dulce en la relación de la potencia generada y las fuerzas a soportar.

2.3. Zonas de trabajo

En la anterior sección se han explicado de forma breve los principios de funcionamiento aerodinámico y como se capta la potencia en un aerogenerador. Partiendo de esa base, en esta sección se explicarán las zonas en las que este trabaja. Para ello se presentarán varias curvas de los parámetros más representativos del punto de operación de un aerogenerador respecto al viento. Estas curvas serán la de potencia, velocidad de rotor, par, y ángulo de

pitch, y han sido obtenidas mediante simulaciones en condiciones ideales y vientos estáticos con OpenFAST para la máquina de referencia IEA 15MW.

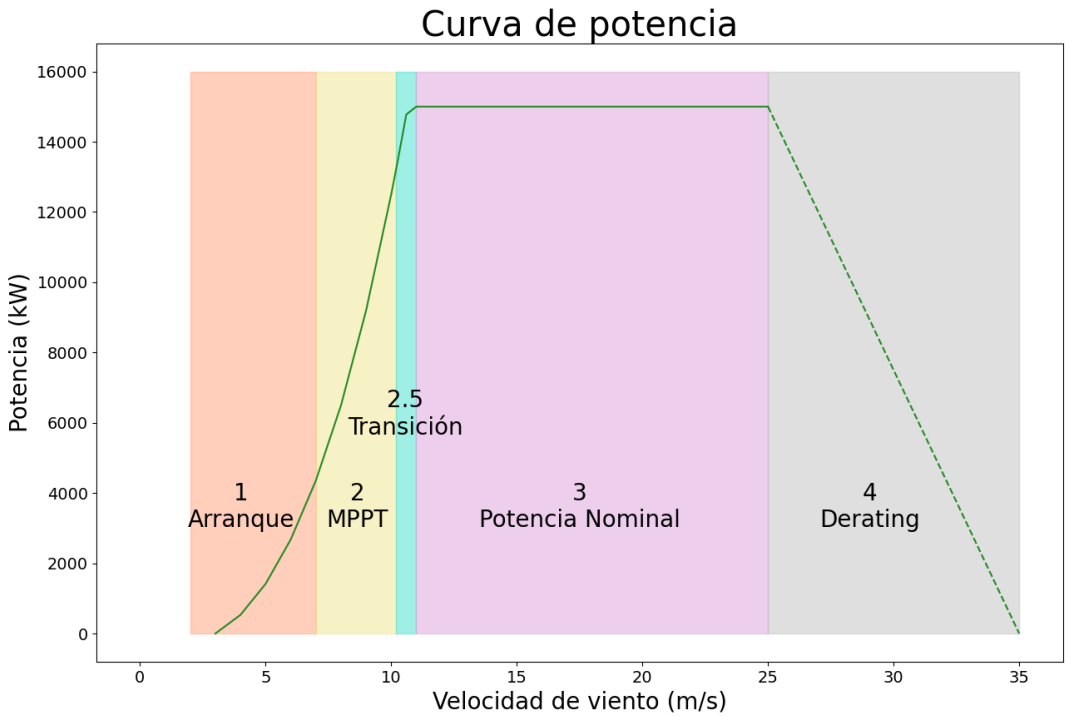

Figura 2.6: Curva de potencia de un aerogenerador.

En la figura 2.6, se pueden apreciar las potencias que entrega un aerogenerador en función del viento que recibe. Como puede verse, la potencia representada por la línea de color verde, va creciendo desde el punto de arranque hasta que llega a un momento de saturación en el que se entrega el valor nominal. Las distintas áreas numeradas representan los distintos modos de trabajo, cada uno con sus objetivos que se describen a continuación:

- **1 Arranque:** en esta zona la máquina está pasando de una inactividad, bien por bajos vientos, paradas programadas, paradas productivas, u otras; a producir energía de forma activa. Cuando una máquina está parada, el ángulo de pitch se situa en una posición en la que no se producen fuerzas de arrastre y que normalmente está a unos 90 deg, es decir el perfil se sitúa paralelo al viento. A este ángulo de parada se le suele conocer como ángulo de bandera (o feather). Cuando un aerogenerador detecta que el viento existente es óptimo para producir (tambien conocido como cut-in wind), este comienza a girar la pala para colocar el perfil aerodinámico de forma perpendicular al viento. Durante el intervalo de arranque, la máxima potencia no suele ser el objetivo principal, dado que en este punto la producción porcentual de la máquina es baja. El principal objetivo suele ser acumular energía en el eje maximizando el par, como

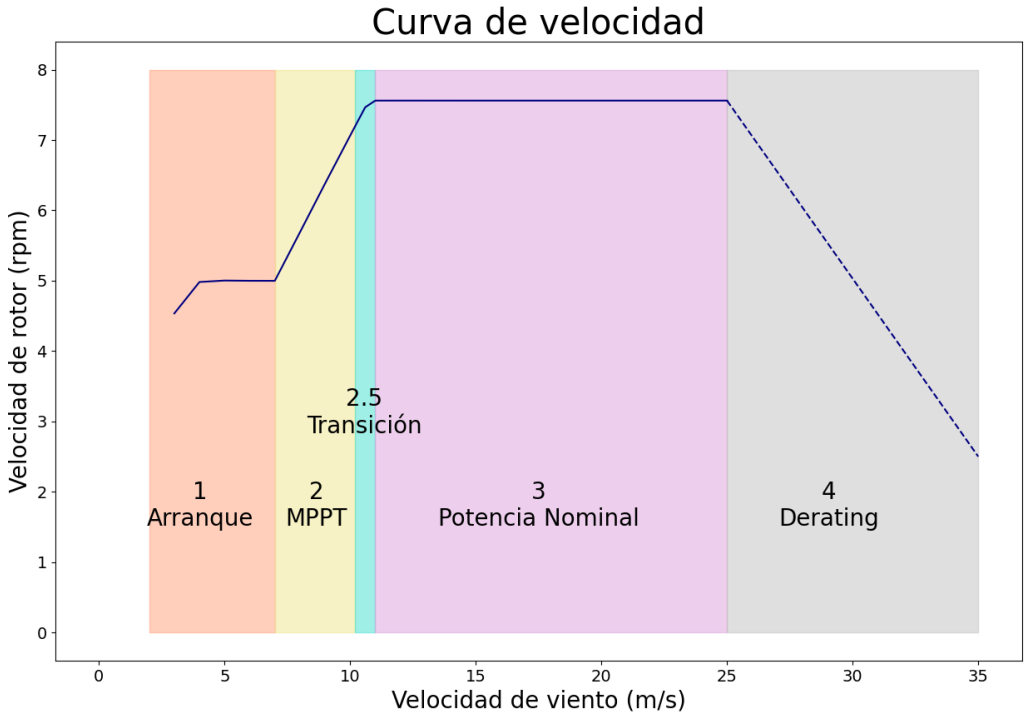

Figura 2.7: Curva de velocidad de un aerogenerador.

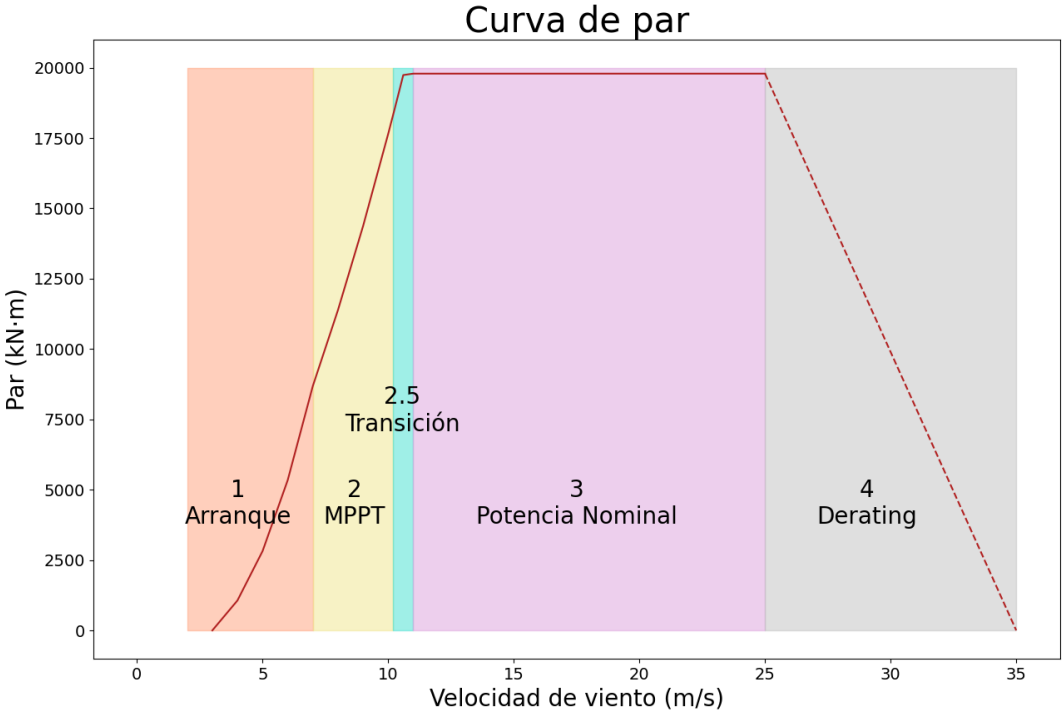

Figura 2.8: Curva de par de un aerogenerador.

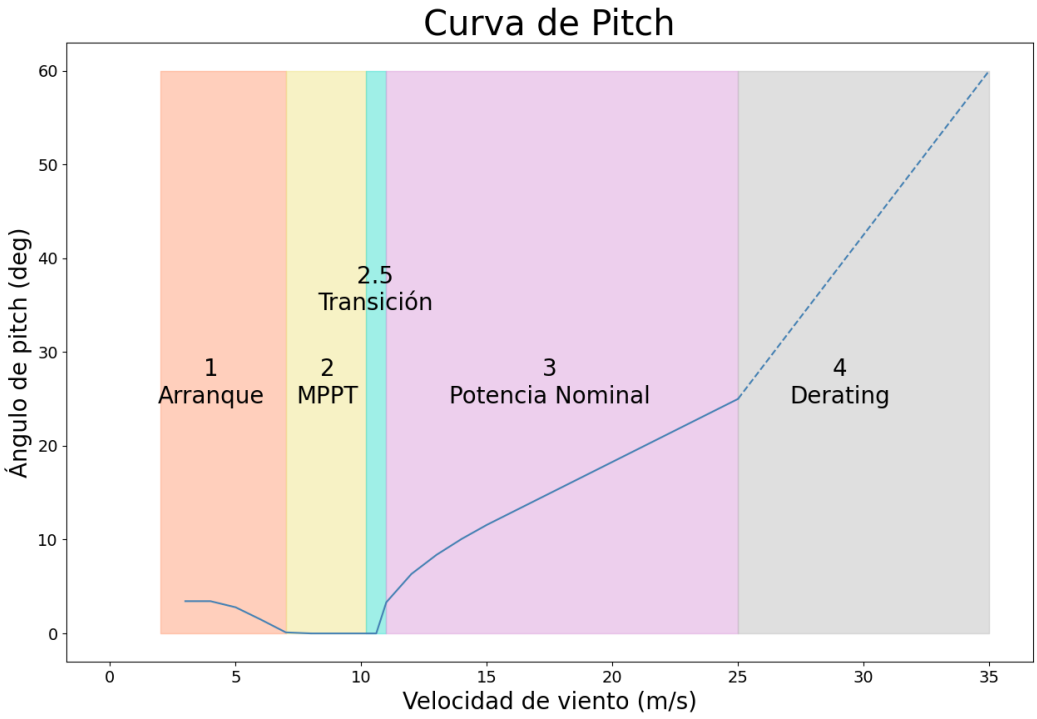

Figura 2.9: Curva de pitch de un aerogenerador.

puede verse en la imagen 2.8, en la que puede apreciarse como la pendiente del par cambia entre la primera zona y la segunda. Para acumular esta energía en el eje, suele hacerse a una velocidad constante, como puede verse en la figura 2.7, para evitar que las variaciones de viento lleven la máquina a un punto de casi apagado, o para evitar que esta velocidad al bajar entre en resonancia con elementos como la torre o las palas, dado que estas suelen tener modos de vibración que suelen acoplarse a bajas frecuencias.

- **2 MPPT:** una vez se considera que ya se tiene suficiente energía en el eje, se ha superado el arranque y se pasa a una zona en la que se intenta extraer la mayor cantidad de potencia disponible o Maximum Power Point Tracking (MPPT). Los aerogeneradores, al igual que los paneles solares, tienen una curva de funcionamiento óptima; esto es que para cada viento hay una relación de par y velocidad que maximiza la potencia. Durante este intervalo, el pitch de las palas se mantiene en una posición en la que captan la mayor cantidad posible de energía del viento, siendo este conocido como pitch óptimo (o fine pitch). El objetivo en este intervalo es maximizar la potencia disponible en cada momento cumpliendo la relación de par y velocidad; para ello se controla la velocidad del rotor empleando el par en función del viento, y se explicará con más detalle en la sección 2.3.1.

- **2.5 Transición:** entre las secciones de MPPT y de potencia nominal, se encuadra un pequeña zona de transición. Esta zona tiene como principal objetivo hacer un cambio seguro entre los objetivos de tener un seguimiento de punto óptimo o MPPT, y funcionar regulando para obtener la potencia nominal. Esta zona, a pesar de producirse en un intervalo corto de vientos, presenta grandes desafíos como es el funcionamiento desacoplado de distintos controladores. Además, en ella se producen ciertos efectos como el máximo empuje sobre la máquina, ya que cuando se llega a un viento en el que se produce la entrada en nominal de la máquina, los empujes son máximos por situarse la posición de pitch en máxima captación de energía.

- **3 Potencia Nominal:** una vez el viento llega a un valor en el que se puede ofrecer la potencia para la que ha sido diseñado el aerogenerador (también conocido como rated wind), la potencia que es generada ha de saturarse para no tener una sobrecarga en los sistemas eléctricos y mecánicos. Para ello, como el par ha de haber llegado a su nivel nominal siguiendo la curva de MPPT y transición, se ha de dejar de captar la energía del viento cambiando el ángulo de ataque modificando el pitch de las palas (recuérdese que este trabajo versa sobre máquinas de velocidad y pitch variables, como se dijo con anterioridad, en otras tipologías se emplean otras estrategias tales como las pérdidas aerodinámicas inducidas por diseño). Como puede verse en la figura 2.9, el pitch va aumentando cada vez más de forma que el ángulo de ataque de las palas provoca un par menor respecto a la energía captada con el viento con el fin de mantener el par y la velocidad constantes, lo que se puede apreciar viendo las imagenes 2.8, y 2.7 respectivamente. Esta, junto con el área de MPPT, es otra de las zonas más relevantes a la hora de controlar un aerogenerador, y será ampliada en la sección 2.3.2.

- **4 Derating:** algunas máquinas, especialmente las de grandes potencias, una vez llegan a un determinado viento comienzan a liberar más energía que la que tendrían que soltar para mantenerse a velocidad nominal. Esta línea se representa en las figuras 2.6, 2.7, 2.8, y 2.9 con un trazo discontinuo. Esta estrategia obedece a que a altos vientos las máquinas han de soportar cargas físicas mayores debidas a la turbulencia, no obstante, es deseable seguir generando durante el mayor rango posible de vientos para tener un mejor aprovechamiento de la energía disponible; por lo tanto, en vez de apagar los sistemas llegados a un determinado viento (conocido como cut out wind) y pasar a un estado de reposo, se va soltando cada vez más potencia, coordinando la velocidad y el par hasta que la máquina se apaga o hasta que se llega a un punto en el que decide apagarse (en este último caso habría una curva con una caída abrupta). Cabe destacar que esta zona de trabajo, al contrario que las anteriores, no está presente en todas las máquinas de velocidad y pitch variables; no obstante, es una estrategia que en los últimos años ha sido utilizada por grandes fabricantes de la industria como Vestas o Enercom con aerogeneradores de cierta envergadura. A pesar de que la má-

quina que se toma como referencia no incluye esta zona en su teorización de zona de operación, se ha incluido para ofrecer más información.

2.3.1. MPPT

En el comienzo de esta sección se han tratado las zonas de trabajo de un aerogenerador. Una de las principales teniendo en cuenta la capacidad de generación y las horas de trabajo que se estiman sobre el total sería la de MPPT. En esta zona como se mencionó el objetivo es aumentar lo máximo posible la cantidad de energía generada. Retrotrayéndose a las relaciones expresadas al inicio de este capítulo de relaciones aerodinámicas, puede observarse la relación (2.1); en ella se aprecia como la potencia depende de varios factores. Unos de ellos serán incontrolables por parte del diseñador o del controlador de la máquina como la densidad del viento o la velocidad del mismo; si bien es verdad que las máquinas se diseñan para una tipología de viento y se instalan en lugares adecuados a este, el valor instantáneo del mismo no es objeto de control y por la contra es la perturbación que hace que el dispositivo funcione. Otros como el radio del rotor tiene cambios minúsculos durante la operación debido a la flexibilidad de las palas, y el control de la potencia mediante el mismo sería imposible con el planteamiento actual de las máquinas, ya que serían necesarias palas con capacidad retráctil que serían económicamente inviables. Por lo tanto, el único factor sobre el que puede actual el controlador sería el coeficiente de potencia C_p. Asumiendo que existe un coeficiente de potencia máximo, se puede decir entonces que la potencia máxima para cada viento, se extraerá cuando el estado de la máquina se regule de forma que este factor se maximice (2.6).

$$P_{A_{\text{máx}}} = \frac{1}{2}\rho\pi R^2 {v_w}^3 C_{p_{\text{máx}}}(\lambda, \beta) \tag{2.6}$$

Como puede verse, este coeficiente de potencia depende de los factores de TSR y de ángulo de pitch. Para obtener este coeficiente de potencia máximo, la pala se ha de colocar en un ángulo óptimo en el que se maximiza la captura de energía del viento. Está más allá del espectro de estudio de este trabajo, y para más información se recomienda consultar la bibliografía, especialmente (Burton et al. [2021]), pero las palas se diseñan de forma que su ángulo de ataque varía a lo largo de los distintos elementos diferenciales aplicando una torsion (o twist), equilibrándose y no obteniendo el óptimo en casi ninguna de las secciones, pero haciendo la aerodinámica más estable en entornos reales, lo que provoca que en ninguna pala fabricable se alcance anteriormente mencionado límite de Betz, y que estas cuenten con coeficientes comprendidos entre un 0,45 y un 0,49. De esta forma, es posible obtener un ángulo para el cual la suma de los elementos diferenciales maximiza la potencia global otorgada. Este ángulo suele tomarse como referencia de 0 deg, y durante el MPPT la pala se bloquea en este ángulo al contrario que en el arranque, en el que se maximiza el par y la pala puede contar con otros ángulos como puede verse en la figura 2.9. Durante la transición

a potencia nominal, puede ser que este ángulo varíe también ligeramente para no entrar de forma abrupta en el cambio de controles, lo que podría provocar una sobreaceleración de la máquina.

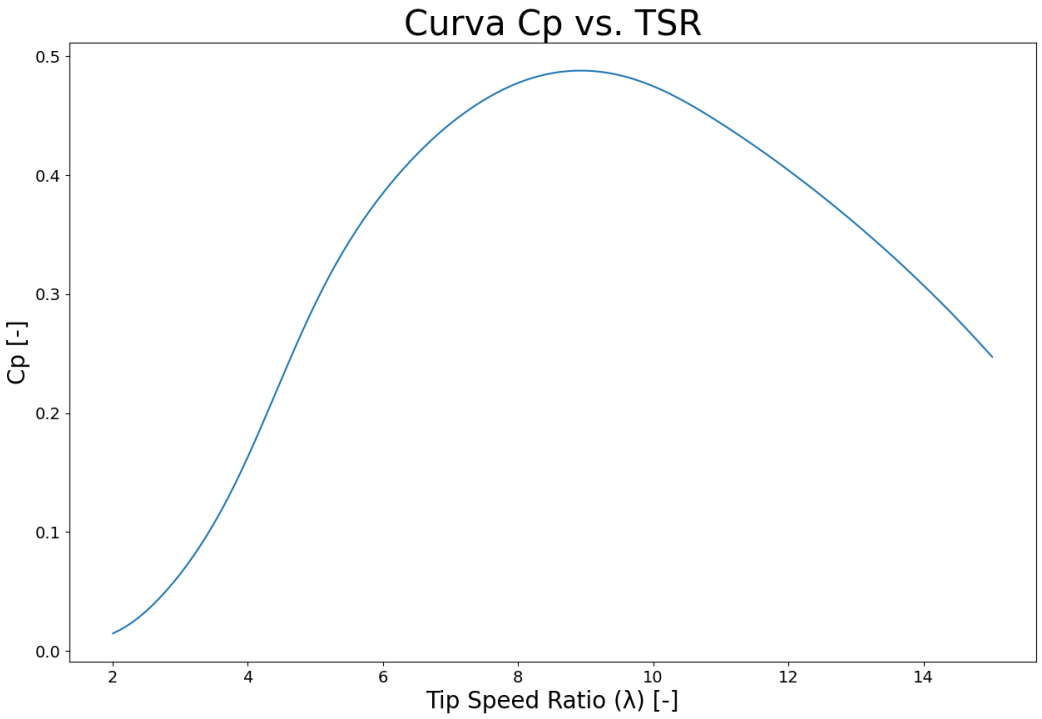

Figura 2.10: Curva de C_p en función de TSR (λ) para un pitch fijo.

Sabiendo que el coeficiente de potencia máximo se obtiene para un ángulo concreto, el otro factor del que depende este para ser maximizado sería la velocidad relativa de punta de pala. El TSR está expresado en la ecuación (2.2), en ella se puede ver que la velocidad relativa de punta de pala está ligada con el viento. Para cada pitch, existe un punto de TSR en el que se maximiza el coeficiente de potencia como puede verse en la figura 2.10, por lo que eligiendo el punto de trabajo para el cual el C_p es máximo existirá un pitch y un TSR determinados.Las palas modernas, se diseñan para tener un TSR óptimo en el cual se maximiza este coeficiente de potencia (Abbas et al. [2021], Manwell et al. [2010], Burton et al. [2021]), esto se debe a que es posible alcanzar coeficientes de potencia muy altos pero con curvas que terminan de forma excesivamente picuda; lo que provocará que cuando el viento se desvie ligeramente por causa de variación o turbulencia, la potencia puede llegar a caer de forma muy drástica. Por lo tanto, en los diseños modernos se prioriza tener superficies más con forma de meseta, en las que el C_p puede no ser tan alto, pero con un comportamiento mucho más estable frente a los cambios, lo que a la larga induce menos problemática de fatiga por variación de fuerzas y, además, incrementa la potencia generada a lo largo del tiempo.

Visto lo anterior, sabiendo que el pitch se encuentra fijado, y que el TSR varía en función del viento y de la velocidad del rotor, puede intuirse que para maximizar el C_p existe una relación entre la velocidad de viento y la velocidad de máquina para la cual el TSR es constante y deseado. Entonces, para controlar la velocidad de rotación, se podrá regular el par electromagnético del eje con el fin de ofrecer más o menos carga. Esto puede ser expresado como la diferencia entre el par aportado y el extraido obviando las distintas pérdidas eléctricas en la ecuación (2.7):

$$\frac{\partial \omega_G}{\partial t} = \frac{N}{J}(\tau_A - \tau_E N \eta_M) \tag{2.7}$$

Donde ω_G es la velocidad del generador que se relaciona con la de rotor mediante la relación de transmisión, τ_E es el par eléctrico extraído, N es la relación de la caja de cambios (1 en caso de acoplamiento directo), J es el momento de inercia total del rotor visto desde el generador, y η_M es el rendimiento mecánico de la caja de cambios y los distintos elementos. Entonces, fijándose en la relación (2.3), puede llegarse a la expresión de un torque ideal para cada velocidad de rotor (2.8):

$$\tau_E(t) = K{\omega_G}^2(t) \tag{2.8}$$

Donde ω_G sería la velocidad del generador, la constante K sería obtenida de despejar el par en la ecuación (2.7) en estado estacionario en combinación con (2.3) asumiendo un TSR óptimo, de forma que este valor sería (2.9):

$$K = \frac{\pi \rho R^5 C_{p_{\text{máx}}}}{2{\lambda_{opt}}^3 N^3 \eta_M} \tag{2.9}$$

Esta relación planteada por (Bossanyi [2000]), es la base sobre la que se basa el control de par moderno en los aerogeneradores, y es conocida como Optimal Mode Gain (OMG).

2.3.2. Potencia Nominal

Como se ha mencionado con anterioridad, la otra zona más importante a nivel de producción de energía de un generador eólico, sería la de potencia nominal. En esta zona el viento ha alcanzado un valor para el cual la potencia ha de saturarse con el fin de evitar daños por sobrecarga continuada en sistemas eléctricos, o daños estructurales provocados por intentar absorber potencia mecánica más allá de los límites para los que está diseñado el aerogenerador. Para cumplir esto, la estrategia mayoritaria es saturar el par a su valor nominal una vez se alcanza este punto, y regular la velocidad con la su valor nominal como objetivo. Esto provocará que la potencia no sea exactamente la nominal debido a las variaciones turbulentas del viento tal y como puede verse en la figura 2.11.

Observando la ecuación (2.7), se puede ver que la velocidad del generador depende tanto del par eléctrico que se extrae, así como del par aerodinámico aportado por el viento. Como

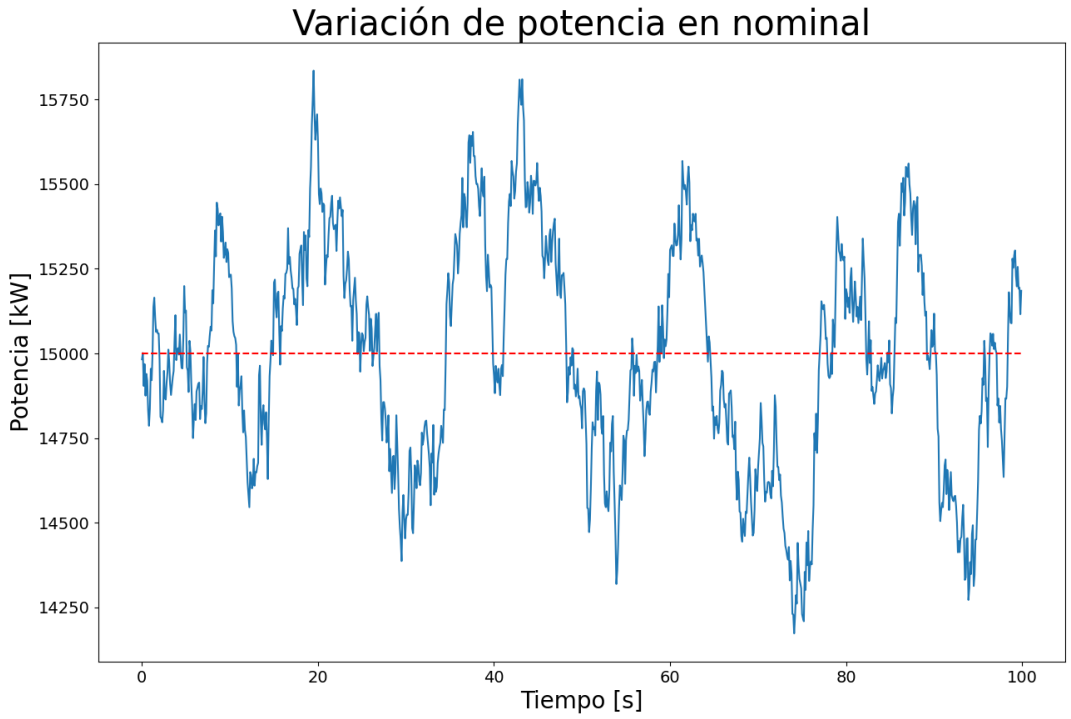

Figura 2.11: Variación de la potencia sobre la nominal.

en este caso el par eléctrico se mantiene constante, la otra opción para gobernar el valor de la velocidad de rotor, sería la de incrementar o decrementar el par aerodinámico igualando ambos. Por lo tanto, para alcanzar este objetivo se regula el pitch de pala haciendo que este atrape más energía aproximándolo a su punto de captación óptima, en el que trabaja en MPPT; o haciendo lo contrario y alejándolo desde ese punto en dirección de captación nula o punto de bandera. Obviamente, dado que en este caso lo deseado es ir perdiendo potencia a medida que crece el viento, en este caso los conceptos planteados en la zona de trabajo MPPT de TSR y de C_p no carecerán de relevancia. Como puede verse en la figura 2.9, debido a las complejas relaciones aerodinámicas esta relación no es lineal, y el control en esta parte puede presentar varios desafíos de los que se hablará en la siguiente sección.

2.4. Esquemas de control básicos

Dado que la explicación de un control integral de máquina exigiría una cantidad de literatura que se escapa del ámbito de este trabajo, en esta sección se ofrecerá una breve explicación de las estrategias de control más empleadas para las zonas de trabajo de MPPT y de potencia nominal, así como de la transición entre ambas.

En la figura 2.12 se puede observar un esquema básico y genérico de las distintas partes que compondrían el control de un aerogenerador. Como se mencionó en la anterior sección 2.3,

los objetivos para las distintas zonas están diferenciados y actúan sobre diferentes variables, por lo tanto se contará con dos controladores básicos en la mayoría de las máquinas. El primero de ellos actuará en la zona de MPPT y se encargará de regular el par para obtener la velocidad deseada según la estrategia seguida empleando para ello el generador eléctrico como actuador. El segundo, será un control del ángulo de pitch que regulará la velocidad para mantener esta en su estado nominal utilizando servomotores que giran la pala como actuadores. Para coordinar los esfuerzos entre estos controladores, será necesaria una capa superior de control supervisorio que los coordine impidiendo que estos tomen el mando sobre la principal variable de control, que sería la velocidad, y permitiendo que ambos coexistan en la región de transición. Esquemas como el mostrado, pueden verse a lo largo de la literatura con ligeras variaciones, pero siguiendo todos este principio básico (Bossanyi [2000], Wright [2004a], Abbas et al. [2021], Burton et al. [2021], Wright and Fingersh [2008a], Bianchi et al. [2007]). Además, junto con estos controles principales de operación, será necesario contar con una serie de controles secundarios que coordinen los esfuerzos de servicios auxiliares, como la orientación respecto al viento, las señales de balizamiento, etc.

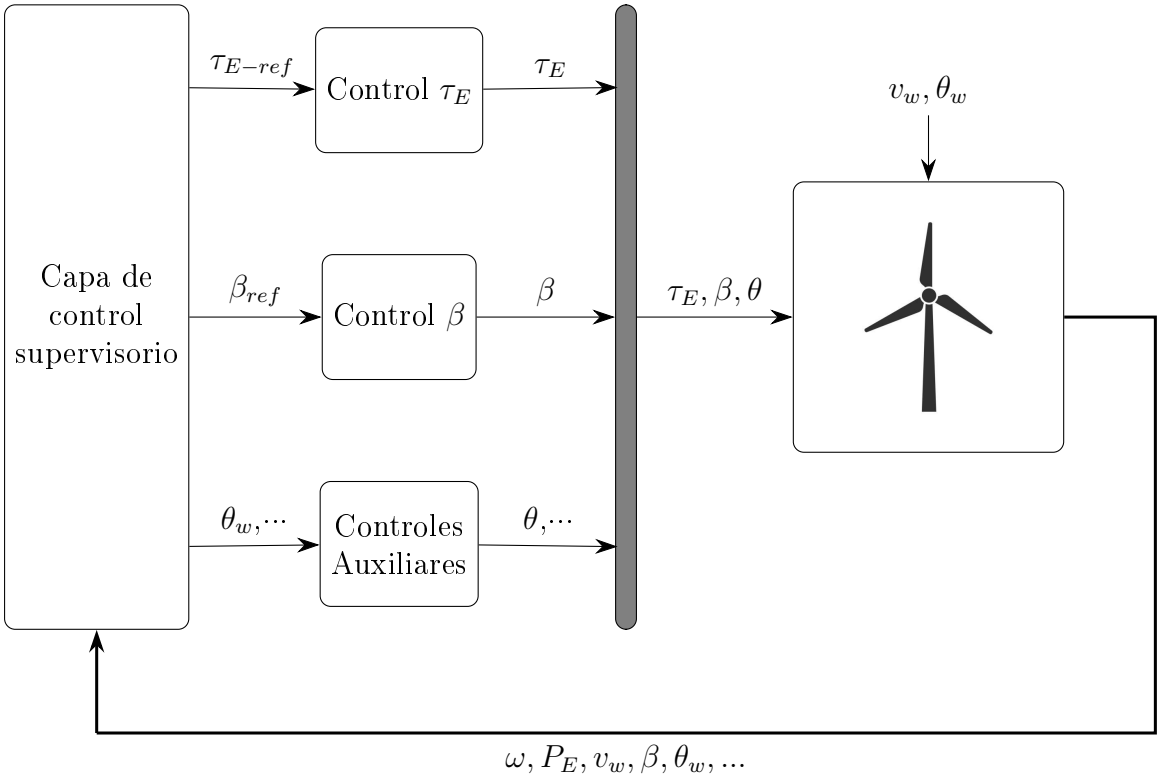

Figura 2.12: Esquema básico de capas de control.

Los controladores más empleados en ambos rangos de actuación de la máquina se basan en controladores Proporcional Integral Derivativo (PID). Como se ha visto en las anteriores secciones 2.3.1 y 2.3.2,el objetivo de control marcado en ambas suele ser controlar la velocidad, bien con fin de maximizar la energía; o bien con la idea de mantener esta en su punto

más cercano al nominal para, mientras se mantiene el par, entregar la potencia de diseño del dispositivo.

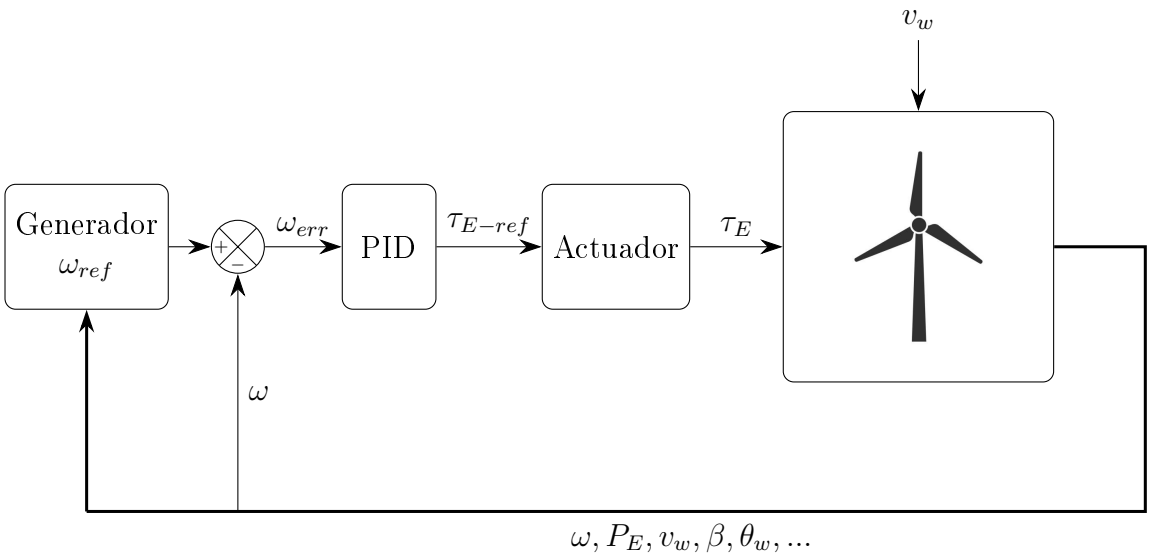

Figura 2.13: Control en la zona de MPPT.

En el caso del funcionamiento en el área de MPPT, el esquema maś común es el que se muestra en la figura 2.13. En esta configuración, las variables del punto de operación se realimentan a un generador de referencia de la velocidad de rotor que intentará maximizar la potencia generada por la máquina. Para ello, existen numerosas alternativas, una de las opciones más clásicas sería tomar como referencia la velocidad que se obtiene del OMG expuesto en la ecuación (2.9); no obstante, en las máquinas modernas de grandes rotores y palas flexibles, este supuesto presenta ciertas pérdidas, por lo que se suele obtar por hacer un seguimiento del TSR de la máquina tal y como se propone en (Abbas et al. [2021]), tomando como base la ecuación (2.2), y reordenando los términos para obtener la velocidad ideal según el TSR óptimo (2.10). Para hacer un seguimiento efectivo del TSR, se necesitan mediciones fiables de viento, lo cual no es una cuestión trivial dada la naturaleza estocástica de la perturbación, a lo que se puede sumar el ruido introducido por las palas en las turbinas upwind (que se enfrentan al viento de cara). Con el fin de mitigar lo máximo posible estas incertidumbres, se pueden utilizar equipos de medida de Laser Imaging Detection And Ranging (LIDAR) en contraposición con los sensores sónicos normalmente empleados, estando ya los sensores los mecánicos prácticamente en desuso empleándose sólo para máquinas de muy baja potencia. Otra opción, ya que el LIDAR presenta la desventaja de que su precio es muchas veces superior en orden de magnitud al del resto de sensórica, sería obtener una estimación del viento empleando un filtro de Kalman, tal y como se propone en (Knudsen et al. [2011]). Para este controlador además, cabe citar la posible problemática de que la velocidad que genere coincida con un punto de resonancia mecánica de alguno de los componentes a bajas frecuencias de rotación, encontrandose estos puntos normalmente en la torre. Con el

fin de no incurrir en esta problemática que degradaría la vida últil de la máquina, habrá que considerar la adición de bandas de exclusión de velocidades, por ejemplo con generadores de histéresis para la señal de referencia.

$$\omega_{ref}(t) = \frac{\lambda_{opt} v_w(t)}{R} \tag{2.10}$$

Entonces, una vez generada la señal de referencia de velocidad en zona de par, esta pasa a traves del controlador PID, y entonces se genera una señal de par que se envía a los convertidores de potencia, que a su vez mediante un control en cascada, provocan que se extraiga un par eléctrirco del generador. Siendo entonces este último elemento el actuador que regula la velocidad de la máquina según la relación (2.7).

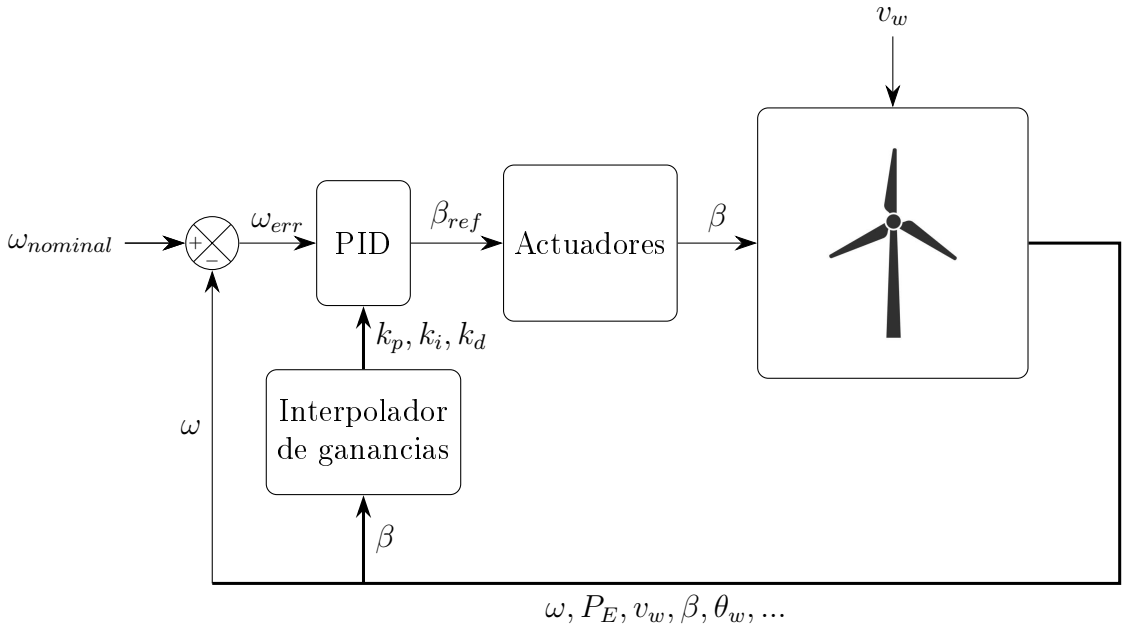

Figura 2.14: Control en la zona de potencia nominal.

Para la zona de potencia nominal, se emplea un esquema similar al que se puede ver en la figura 2.14. Como se comentó con anterioridad, en este rango de operación lo que se busca es mantener una velocidad de máquina constante mientras que se extrae el par nominal. Para ello, se emplea la velocidad de diseño como referencia. Con el fin de mantener esta velocidad lo más próxima posible al punto de operación, la diferencia entre esta y la real de la máquina se procesa con un PID, que genera una referencia para que las palas cambien su ángulo de ataque para captar más o menos energía. Esta operación está comandada, bien por un sistema colectivo, o bien por servomotores individuales que establecen el ángulo de cada pala. Dado que existe una alta componente no lineal en la aerodinámica, es una práctica común ajustar las ganancias para distintos puntos de operación según el ángulo que existe en ese momento (Bossanyi [2000], Bianchi et al. [2007]). Esta estrategia, conocida como Gain Scheduling, propone hacer el ajuste según los criterios deseados en cada punto,

para luego más tarde interpolar entre ellos, bien de forma lineal, con ajuste de curvas, o con otros métodos para, de esta forma, obtener un comportamiento suave a lo largo de todo el rango de operación nominal de la máquina.

Como puede verse en la curva de potencia mostrada en la figura 2.6, existe una zona de transición entre ambos controladores principales. Esta zona es especialmente delicada desde el punto de vista del control general, ya que exige cortar de forma abrupta la escalada de potencia que se produce de forma cúbica con la velocidad de viento, como puede verse en la ecuación (2.1), para entrar en una zona plana. Un mal diseño de este intercambio, puede tener efectos nefastos a la hora de gestionar un aerogenerador, pudiendo incurrir en escenarios peligrosos como sobrevelocidades que disparen sistemas de emergencia. Además es delicada desde el punto de vista de la interacción, ya que ambos controles intentan ejercer su dominio sobre la misma variable de velocidad de rotor, por lo que se podrían generar inestabilidades. Uno de las primeras estrategias que se empleó fue similar a las de arranque, reteniendo la velocidad antes de saltar a nominal mientras se acumula energía inercial en el eje, pero esta estrategia es muy subóptima desde el punto de vista energético y estructural, pudiendo incluso inducir vibraciones de alta frecuencia por chattering. Otra alternativa mejor y más común, consiste en cambiar la curva de seguimiento a partir de un momento óptimo para crear un aterrizaje suave con una pendiente distinta; esta opción suele emplear una aproximación lineal por una recta para ejercer de interfaz entre los dos controles y se ha empleado en numerosos trabajos Jonkman et al. [2009]. Más recientemente, se han desarrollado desacopladores que, empleando la saturación de los controladores, son capaces de elegir uno u otro en función de su punto individual de operación como el propuesto por Schlipf [2019] en el que empleando los límites de operación de pitch y potencia se generan referencias que no colisionan entre si.

Existen además, otros controles que se obvian en este trabajo pero que también son vitales para el funcionamiento normal de los aerogeneradores, como controladores de orientación que suelen estar basados en sencillas reglas lógicas, o incluso veletas mecánicas en modelos de muy baja potencia; pero con variantes incluso de control inteligentes preparadas para nuevas generaciones de máquinas con necesidades más complejas Barrio et al. [2024]; sin el buen funcionamiento de este sencillo control la operación de los aerogeneradores de eje vertical sería inviable ya que no conseguirían captar la energía necesaria y correría riesgo incluso su seguridad estructural. Tienen que contar también con una gama de controladores de emergencia capaces de detener las máquinas en caso de que se presente algún efecto adverso o avería, por ejemplo el bloqueo de uno de los actuadores de pitch, y estos suelen estar implementados en diversos elementos de seguridad con capacidad de funcionar de forma independiente como los servomotores.

Se pueden encontrar también una gama de controles auxiliares, o mejoras para los controladores principales, que pueden intentar mejorar la respuesta ante uno o varios efectos que se producen durante la operación de los aerogeneradores. Entre este último grupo, po-

drían encuadrarse controles auxiliares que persiguen el amortiguamiento de vibraciones para atenuar estas en elementos como los trenes de transmisiones o torres (Wright [2008], Wright and Fingersh [2008b]); u otro modo de operación especial conocido como Individual Pitch Controller (IPC), que consiste en operar de forma independiente las distintas palas del aerogenerador, modificando su pitch en conjunto pero con leves variaciones individuales para intentar mitigar los efectos de las fuerzas de cortadura del viento en las distintas posiciones azimutales, lo que sería de especial interés para máquinas de gran tamaño (Wright [2003]).

En esta sección se ha tratado de forma superficial y genérica los controladores más comunes. No obstante, existen numerosas alternativas a los explicados, sobre todo en el ámbito académico. Existe un abanico de controladores para el MPPT que van desde Lookup Tables (LUT), que generan un par proporcional a la velocidad en la que se encuentra la máquina según la relación (2.1). Otros autores, siguiendo una estructura similar a la expuesta, han hecho controles modernos basados en Linear Quadratic Regulators (LQR) con un enfoque multiobjetivo (Wright [2004b]). Existen también algoritmos denominados Perturb & Observation que consisten en introducir perturbaciones en el rotor y evaluar si estas han sido efectivas teniendo en cuenta la potencia generada, evaluando estos contra las superficies de potencia (Alhamdawee and Hussain [2024]). También hay otras alternativas basadas en controladores inteligentes que combinan Redes Neuronales con Fuzzy Logic, en los cuales la referencia se establece de forma no lineal dependiendo de como de lejos el generador se sitúa de su punto de trabajo ideal (Muñoz-Palomeque et al. [2024]).

Respecto a los controladores de pitch, existe también una amplia gama de controladores disponibles. Entre ellas se puede encontrar el control moderno (Wright [2004b]). El control no lineal basado en Feedback Linearization (Ralph et al. [2011]), el el que se crea una ley de control basada en los elementos no lineales de la máquina. O controladores inteligentes basádos en Fuzzy Logic con configuraciones híbridas de control (Sierra-García and Santos [2021]).

2.5. Modelo de viento

Hasta ahora en este trabajo se ha explicado el funcionamiento de un aerogenerador desde el punto de vista de la máquina, pero esta sería inservible sin el viento que la mueve. El viento, es un fenómeno puramente estocástico que varía en cuestión de segundos, y que produce una perturbación en el sistema mecánico. Es un recurso que es altamente variable, tanto geográfica como temporalmente, teniendo variaciones distintas a lo largo de la estación del año para distintos emplazamientos (Burton et al. [2021]). En él influyen no sólo factores climáticos, sino también topográficos, por lo que tener una predicción de la velocidad de viento disponible en un lugar es una tarea compleja teniendo que recurrir a grandes bancos de datos y complejos modelos meteorológicos. Entonces, para emplazar correctamente un aerogenerador, se tomará en consideración datos previos de medida de torres con sensores

que se instalan, y que llevan a cabo un estudio que genera gran cantidad de información para evaluar la viabilidad del recurso. Empleando estos informes, se obtiene un esquema de vientos propios de un lugar tal y como se puede ver en la figura 2.15, que vale para realizar no sólo el estudio de viabilidad, sino para decidir que tipo de máquina utilizar, y que orientación preferente tendrá esta ya que, si bien es cierto que poseen un sistema de orientación, su ángulo de referencia se colocará siempre con respecto al viento dominante.

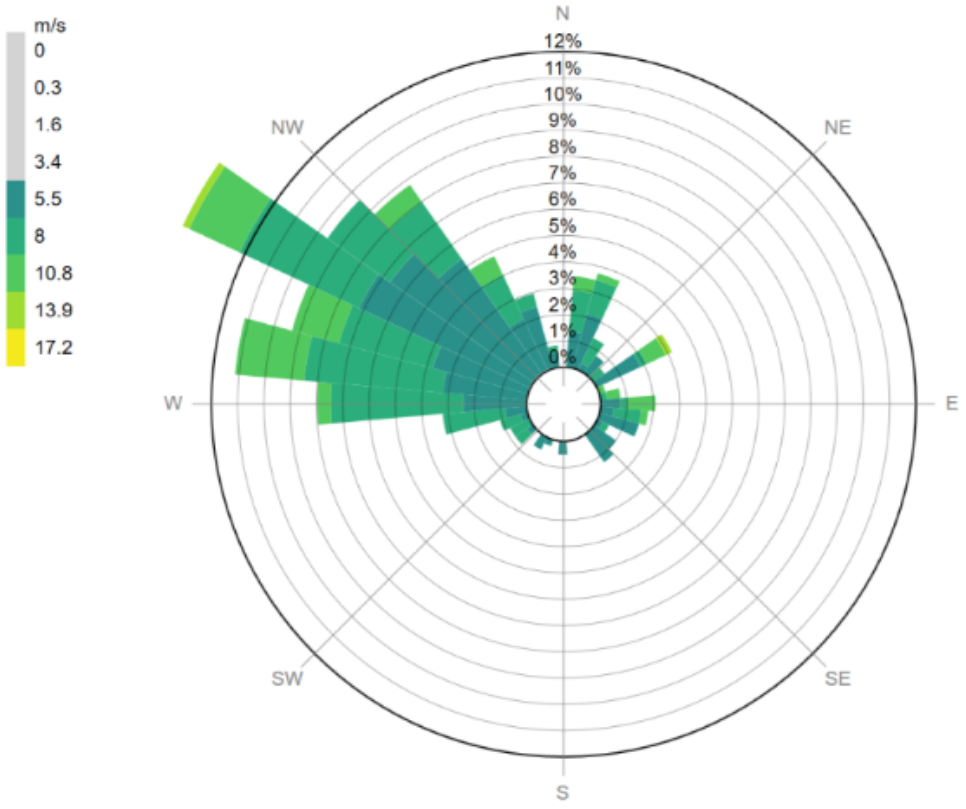

Figura 2.15: Ejemplo de mapa de orientación y velocidad de viento para un emplazamiento.

Para llevar entonces a cabo el diseño y simulaciones de los dispositivos eólicos, se precisan de modelos de viento que representen con cierta verosimilitud las cargas y las fuerzas a las que tendrán que hacer frente los aerogeneradores. De hecho, es un requerimiento imprescindible para la certificación de aparatos marcado por la normativa IEC [2019] comprobar el comportamiento de las turbinas en simulaciones que presenten un viento turbulento. Existen entonces estudios que intentan identificar el comportamiento del viento basándose en su comportamiento turbulento. La turbulencia, se refiere a las variaciones rápidas de viento que pueden abarcar desde una escala de segundos hasta minutos, y está generada principalmente por dos factores, fricción con la superficie terrestre debida a la circulación del flujo por lugares con accidentes geográficos como montañas, valles, bosques, etc. que crean efectos de aceleración y frenado de masas de aire debido a la compresión del fluido; y otros efectos térmicos que causan que las masas de aire se muevan de forma vertical creando variaciones de densidad (Burton et al. [2021]).

Debido a la complejidad de modelado devenida de las múltiples influencias, es un efecto que no se puede representar con ecuaciones determinísticas; si bien partiendo de la base que obedece a leyes físicas como las de conservación de la masa, conservación del momento, y conservación de la energía. Es posible entonces hacer una representación plausible de este fenómeno empleando ecuaciones diferenciales con ciertas restricciones, ya que en un modelo tan caótico pequeñas diferencias en las entradas producen grandes diferencias en las salidas en un espacio relativamente corto de tiempo (Burton et al. [2021]). Estos modelos suelen referenciarse a una intensidad de turbulencia, que es la desviación estándar de la velocidad del viento en un periodo de 10 minutos dividida por la media de su valor, ya que en ese intervalo se puede aproximar de forma más o menos correcta estos valores por una distribución gausiana (2.11).

$$I = \frac{\sigma}{\bar{V}_w} \tag{2.11}$$

Otro factor de gran importancia a la hora de caracterizar los vientos es el shear (o cortadura). Este efecto se produce por el rozamiento con el terreno, y provoca que los vientos en altitudes superiores tengan más velocidad que en inferiores, ya que a bajas cotas se ven frenados por el rozamiento con el suelo y con obstáculos. Entonces las velocidades en las distintas alturas se pueden aproximar según la expresión (2.12).

$$v_w(z) = v_{w-ref}\left(\frac{z}{z_{ref}}\right)^{\alpha} \tag{2.12}$$

Donde $v_w(z)$ es la velocidad en un punto de altura determinado; v_{w-ref} es la velocidad del punto de referencia; z es la altura a la que se quiere obtener la velocidad; z_{ref} es la altura de referencia, normalmente la altura a la que se sitúa el centro del rotor; y α es un factor que depende de la rugosidad y características del terreno.

Hay varias tipologías de viento para simular, comenzando por vientos estacionarios o steady que se emplean para obtener o verificar parámetros ideales y que no contienen turbulencia. Otros simulan efectos adversos como vientos de ráfaga o gust, que son popularmente conocidos como “gorros mexicanos” ya que su perfil se asemeja bastante a esta prenda de vestir, y que se utilizan para predecir que cargas va a soportar la máquina en situaciones atípicas de producción. Y por último existen modelos de turbulencia compleja, en los que se realiza un cálculo para obtener un bloque tridimensional mallado en el que cada punto representa una velocidad de viento, pero todos estos han de ser coherentes entre si. Estos bloques de viento complejos se suelen generar de forma independiente en las simulaciones y, durante el transcurso de las mismas, van avanzando a través de la turbina generando las fuerzas aerodinámicas y de arrastre, un ejemplo de estos bloques se puede ver en la figura 2.16. De esta tipología de modelos los más conocidos son los modelos Kaimal, von Kármán, y Mann. El modelo von Kármán es ampliamente utilizado en diseño aerodinámico y se caracteriza por describir de forma muy realista los efectos de las ráfagas. El modelo Kaimal es uno de

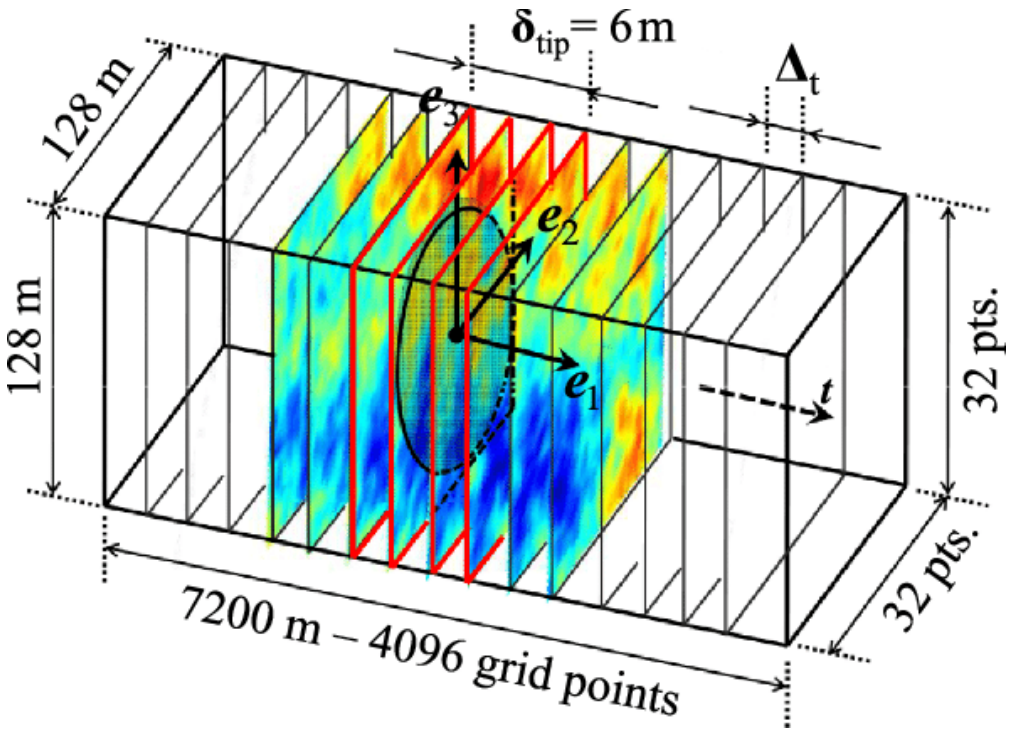

Figura 2.16: Ejemplo de un bloque de viento de Mann para simulacion. Imagen extraída de (Sabale and Gopal [2019]).

los más utilizados y debido a su coherencia y sencillez a la hora de ser generado. Por último, el modelo de Mann de tensor espectral se caracteríza por simular la turbulencia con gran fidelidad capturando con gran exactitud las correlaciones temporales en las velocidades de viento.

2.5.1. Vientos equivalentes

Si bien es cierto que, como se puede ver en esta sección, existen vientos que capturan con una gran precisión las distintas características del viento; también es cierto que para aplicarlos correctamente se requiere una gestión laboriosa, ya que habría que generar un gran número de ellos para tener variabilidad, y referenciar estos durante las simulaciones. Además, como puede intuirse, dado que tienen que componer complejos modelos de malla, estos modelos pueden ser relativamente costosos computacionalmente. Se pueden emplear entonces otros modelos basados en el viento equivalente. Para entender el viento equivalente, habría que observar los bloques y tomando una sección instantánea de los mismos cabría hacerse la pregunta de cual es el viento que existe en esos momentos en el rotor. Como podría verse, en cada punto existiría una velocidad distinta, entonces para tener un efecto similar al que producen todas ellas de forma conjunta se introduce el concepto viento equivalente (2.13).

$$v_{w-eq}(t) = \sqrt[3]{\sum_{i=1}^{n} \frac{A_i}{A} v_{w-i}(t)^3} \tag{2.13}$$

Donde $v_{w-eq}(t)$ es el viento equivalente en un instante temporal, n es el número de puntos de la malla considerados, A_i es el área de influencia que ocupa el punto dentro de la malla, A es el área total del rotor, y $v_{w-i}(t)$ es la velocidad de viento instantánea en un punto de la malla. Esta representación del viento, substituida en la ecuación (2.1), debería producir un resultado aproximado al que se obtendría haciendo una evaluación de la actuación aerodinámica de la malla sobre los perfiles aerodinámicos. Por lo tanto, podría substituirse un complejo modelo tridimensional como el que se veía en la figura 2.16 para pruebas como por ejemplo de control que no requieren de la máxima fidelidad.

Un ejemplo de esta aproximación puede verse en (Thomsen [2006]). Este modelo se tomará como referencia para los vientos generados en este trabajo, y está centrado en obtener una perturbación realista del valor de viento, conservando la coherencia dentro del valor del mismo basándose en estudios meteorológicos previos. Para ello, se toma el valor efectivo del viento como una suma entre el valor medio de la velocidad de viento y el valor de un proceso de ruido blanco que se pasa a través de un filtro de segundo orden. De forma que el valor del viento equivalente que incide sobre el rotor sería el mostrado en la ecuación (2.14).

$$v_{w-eq} = v_m + v_s \tag{2.14}$$

Donde v_{w-eq} sería el viento equivalente, v_m la velocidad media del viento, y v_s la turbulencia aplicada a través del filtro de segundo orden. El valor de esta segunda componente vendría dado por el sistema de ecuaciones diferenciales (2.15).

$$\begin{cases} \dot{w}_s = w_{aux} \\ \dot{w}_{aux} = -a_0 w_s - a_1 w_{aux} + K_v e \end{cases} \tag{2.15}$$

Donde v_s es la variable que se suma para obtener el viento equivalente, e es el proceso de ruido blanco, y a_n y K_v son parámetros dados por las consideraciones del viento que se obtendrán en la sección 4.2 y cuya procedencia se explica a continuación.

La reducción de una representación de área a un punto de viento como el propuesto, se basa en ajustar los parámetros del filtro para que se conserve la misma potencia de intensidad de espectro (Power Spectrum Density, PSD) en el campo de las frecuencias que un viento realista. Basándose entonces en la aproximación de (Højstrup [1982]), se define la potencia de espectro del punto de viento en la ecuación (2.16).

$$S_p(f)\frac{f}{v_{fr}^2} = \left(\frac{0{,}5f_i}{1+2{,}2f_i^{\frac{5}{3}}}\right)\left(\frac{h_i}{-L}\right) + \left(\frac{105f_{ru}}{(1+33f_{ru})^{5/3}}\right)\left(\frac{1-\frac{h}{h_i}}{(1+15\frac{h}{h_i})^{2/3}}\right) \tag{2.16}$$

En la que:

$$\begin{cases} f_i = f\frac{h_i}{v_m} \\ f_{ru} = \frac{f}{1+15\frac{h}{h_i}} \end{cases}$$

Donde f es la frecuencia, h es la altura del centro del rotor, h_i es la altura de menor inversión (altura a la se sitúa una primera capa de aire más cálido por encima de otra de aire más frío de la superfice terrestre), L es la longitud Monin-Obukhov, y v_{fr} es la velocidad de fricción. La ecuación (2.16) se divide en dos partes, la primera parte antes de la suma que sería la dominante en vientos estáticos, y la segunda que sería la dominante en condiciones inestables de viento. Por lo tanto, se hace una aproximación para las condiciones inestables teniendo (2.17).

$$S'_p(f)\frac{f}{v_{fr}^2} = \left(\frac{105f_{ru}}{(1+33f_{ru})^{5/3}}\right)\left(\frac{1-\frac{h}{h_i}}{(1+15\frac{h}{h_i})^{2/3}}\right) \tag{2.17}$$

El único parámetro por conocer de la relación (2.17) sería la velocidad de fricción v_{fr}, ya que la frecuencia será un valor que se explorará. Entonces esta se calcula mediante la varianza del punto de viento (2.18).

$$\sigma^2 = \int_{-\infty}^{+\infty} S'_p(f)\,df = \frac{105}{22}\left(\frac{1-\frac{h}{h_i}}{(1+15\frac{h}{h_i})^{2/3}}\right)v_{fr}^2 \tag{2.18}$$

Sabiendo entonces que la varianza del viento se ha definido en (2.11), combinando esta relación con (2.18) se obtiene el valor de esta en función de la intensidad de turbulencia y la velocidad media (2.19).

$$v_{fr}^2 = (Iv_m)^2\frac{22}{105}\left(\frac{(1+15\frac{h}{h_i})^{2/3}}{1-\frac{h}{h_i}}\right) \tag{2.19}$$

Con lo que el espectro de potencia para definir la turbulencia quedará finalmente definido por (2.20).

$$S'_p(f)f = (Iv_m)^2\frac{22f_{ru}}{(1+33f_{ru})^{5/3}} \tag{2.20}$$

Entonces, de acuerdo con (Knudsen [1983]), el espectro de la velocidad efectiva del viento, se puede aproximar filtrando un punto de viento con el filtro indicado en (2.21).

$$F(f) = \frac{1}{\left(1 + \frac{8\sqrt{\pi}R}{3v_m} f\right)\left(1 + \frac{4\sqrt{\pi}R}{3v_m} f\right)} \tag{2.21}$$

Donde R es el radio de rotor. Entonces, combinando el espectro de la turbulencia con el de la velocidad se obtiene el espectro efectivo del viento (2.22).

$$S_e(f) = S'_p(f)F(f) \tag{2.22}$$

Sobre esta relación, se hará la implementación del modelo de viento, lo que se discutirá en el capítulo dedicado a la implementación del controlador y del modelo, en la sección 4.2.

2.6. Modelos de simulación de máquina

Una vez se han presentado de forma resumida tanto el funcionamiento del aerogenerador como los modelos de viento, se procede en esta sección a repasar brevemente las distintas opciones que existen para llevar a cabo una simulación que represente de forma fehaciente el comportamiento que se puede extrapolar a la realidad. Las simulaciones en este campo son especialmente importantes, ya que el diseño de los prototipos está puramente condicionado a las dotes que hayan adquirido los creadores basándose en experiencias previas y a los resultados extraídos de las mismas. Esto es así por varios factores, el primero que son máquinas que por su tamaño son caras de producir, con lo que realizar iteraciones de diseño sobre prototipos sería sumamente antieconómico. Por ejemplo, el costo de una unidad de molde de pala puede estimarse entre varios cientos de miles de euros hasta millones, dependiendo del tamaño de estas. Otra de las principales razones, sería la gran cantidad de restricciones para su montaje entre las que se pueden encontrar las de la concesión de un área, o conexiones a red, pasando por costosos transportes llenos de burocracia.

Por ello contar con sistemas que sean capaces de representar fidedignamente la física, y predecir con exactitud los valores que se van a obtener, son clave a la hora de diseñar los aerogeneradores, tanto en su parte hardware como en su parte de control. Para ello, en esta sección se comentarán tres tipos de modelos que pueden ser aptos para disintos usos. Los primeros complejos modelos aeroelásticos multifísica, que son capaces de conjugar los distintos elementos pertenecientes a distintos ámbitos y ofrecer simulaciones realistas; los segundos, modelos linealizados que ofrecen un comportamiento representativo de la máquina en un punto de operación; y por último, un modelo simplificado que se utilizará en este trabajo, y que es capaz de ofrecer ciertas variables con una buena precisión pero obvia ciertos efectos.

2.6.1. Modelos aeroelásticos

Los modelos de simulación conocidos como aeroelásticos son representaciones que conjugan distintos ámbitos de la física y que son capaces de representar el funcionamiento de un aerogenerador de una forma muy exacta basándose simplemente en datos numéricos de las distintas partes. Para ello existen entornos de simulación como OpenFAST (NREL [2025]), HAWC2 (DTU [2025]), o Bladed (DNV [2025]) entre otros, siendo el primero de ellos de gran interés académico por ser de código abierto y poder realizar modificaciones sobre el código fuente para disintas investigaciones.

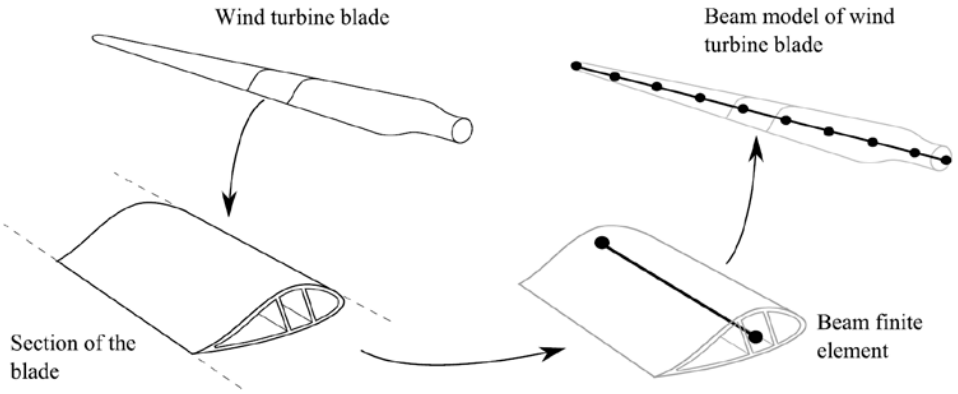

Figura 2.17: Ejemplo de una pala vista como una viga. Imagen extraída de (Branner et al. [2012]).

En estos modelos, se hace un cálculo estructural basado en beams (vigas), descomponiendo los sistemas en distintas secciones como puede apreciarse en la figura 2.17. A estas, se les dota de características físicas introduciendo sus dimensiones y propiedades de los materiales, como las matrices de rigideces. Cabe reseñar que también cuentan con una parte aerodinámica que es capaz de estimar las fuerzas que se generan debido al viento en los distintos elementos, desde la torre hasta las palas, sobre las que se generan las fuerzas de empuje. Este viento puede ser cualquiera de los mencionados con anterioridad, desde los simples a los complejos turbulentos que incidirán en cada una de las secciones a evaluar y generarán una fuerza.

Las fuerzas generadas por la aerodinámica, se transmitirán entonces de nodo a nodo empleando una implementación de la generalización de la ecuación de fuerzas nodales (2.23).

$$F(t) = M\ddot{x} + C\dot{x} + Kx \tag{2.23}$$

Donde $F(t)$ son las fuerzas generadas, x es la posición, M es la matriz de masas, C la matriz de amortiguamiento, y K la matriz de rigideces. Resolviendo esta ecuación para todos los nodos, se obtendrá el conjunto de fuerzas, momentos, desplazamientos, velocidades, y aceleraciones que operan sobre el sistema físico.

Incluyen además estos programas, una representación de la parte eléctrica que permite simular los efectos del generador y los convertidores de una forma simplificada pero bastante exacta, teniendo en cuenta que los anchos de banda del sistema mecánico y del sistema eléctrico suelen estar desacoplados por ser el primero mucho más lento que el segundo. Además, la capacidad de incluir dinámicas de actuadores, como por ejemplo los servomotores.

Respecto al control, suelen incluir una capacidad de control simple, que es ampliable a un control programado por el usuario que suele interactuar con los programas en forma de fichero de librería ".dll" o ".so", dependiendo del sistema operativo en el que se ejecuten. Para la creación de estas librerías auxiliares, existen también otros programas con capacidad de interacción con los códigos aeroelásticos, que ejercen de interfaz con ellos y son capaces de generar una librería y hacer un tuning de los controladores, como sería la suite de ROSCO (Abbas et al. [2021]), generada como controlador de referencia de código abierto por el centro de investigación NREL, también creador de OpenFAST.

Como puede intuirse, dada la explicación del planteamiento y capacidades de estos softwares, estos programas tienen la ventaja de ser muy exactos. Debido a ello sus salidas se emplean para validar efectos de producción, o para verificar los componentes posteriormente mediante análisis por métodos de elementos finitos (Finite Element Methods, FEM). No obstante, cuentan con una desventaja, son modelos poco flexibles en los que no se puede modificar el código para introducir personalizaciones, o en caso de poder hacerlo es un proceso farragoso. Además, son computacionalmente costosos. Por ello, para otro tipo de análisis en los que se va a trabajar sobre una zona operacional concreta, se puede linealizar un punto de operación, capacidad con la que suelen contar estos programas y del es de modelo que se hablará a continuación.

2.6.2. Modelos linealizados

Con el fin de reducir la complejidad del modelo y poder estudiar ciertos efectos de forma local, en ocasiones se extraen modelos linealizados de ciertos puntos de funcionamiento. Para ello, los códigos aeroelásticos suelen implementar rutinas que permiten obtener estos valores. Por ejemplo OpenFAST cuenta con un modo de linealización que deja bastante libertad al usuario (Jonkman and Jonkman [2016]). El objetivo es obtener una representación del sistema en el espacio de estados (2.24).

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) + B_d u_d(t) \\ y(t) = Cx(t) + Du(t) + D_d u_d(t) \end{cases} \tag{2.24}$$

No obstante, dada la alta no linealidad de los aerogeneradores, esta representación lineal no es viable para todo el rango de operación del aparato, por lo que se aproxima en los puntos elegidos (2.25).

$$\begin{cases} x(t) \approx x_Q + \Delta x(t) \\ y(t) \approx y_Q + \Delta y(t) \end{cases} \tag{2.25}$$

Lo que se hace para ello en los softwares aeroelásticos, es marcar un punto elegido con un viento constante para que se alcance un estado estacionario en él (2.26).

$$\begin{cases} \dot{x}_Q = f(x_Q(t), u_Q(t), u_{d-Q}(t)) \\ y_Q = h(x_Q(t), u_Q(t), u_{d-Q}(t)) \end{cases} \tag{2.26}$$

Una vez la máquina está en reposo en ese estado, se perturba una de las entradas para obtener los Jacobianos numéricamente, lo que dará las matrices de comportamiento (2.27). Por ejemplo, si se quiere obtener un punto linealizado en el entorno del MPPT, lo que se haría es llevar la máquina a estado estacionario, y ahí perturbar la entrada de par ligeramente $\Delta\tau_E$, y con ella medir la respuesta del sistema derivando de forma numérica.

$$\begin{cases} A(t) = \frac{\partial f}{\partial x}\big|_{(x_Q, u_Q, u_{d-Q})} \\ B(t) = \frac{\partial f}{\partial u}\big|_{(x_Q, u_Q, u_{d-Q})} \\ B_d(t) = \frac{\partial f}{\partial u}\big|_{(x_Q, u_Q, u_{d-Q})} \\ C(t) = \frac{\partial h}{\partial x}\big|_{(x_Q, u_Q, u_{d-Q})} \\ D(t) = \frac{\partial h}{\partial u}\big|_{(x_Q, u_Q, u_{d-Q})} \\ D_d(t) = \frac{\partial h}{\partial u}\big|_{(x_Q, u_Q, u_{d-Q})} \end{cases} \tag{2.27}$$

Se obtendría entonces un modelo que representa la perturbación en torno al punto de equilibrio (2.28), que combinado con la respuesta en estado estacionario en el mismo, daría como resultado el comportamiento aproximado del sistema en este (2.25).

$$\begin{cases} \Delta\dot{x}(t) = A(t)\Delta x(t) + B(t)\Delta u(t) + B_d(t)\Delta u_d(t) \\ \Delta y(t) = C(t)\Delta x(t) + D(t)\Delta u(t) + D_d(t)\Delta u_d(t) \end{cases} \tag{2.28}$$

Estos modelos son de especial interes para los ingenieros de control, ya que se emplean para, por ejemplo, llevar a cabo la sintonización de controladores, como el de pitch que se realiza en varios puntos como se comentaba con anterioridad. Además, sirven para estudiar otras dinámicas en zonas puntuales, pudiendo generar así otros controles auxiliares como amortiguadores de vibraciones en ciertos momentos de operación (Wright [2008]). Tienen la ventaja de ser mucho más ágiles numéricamente que su contraparte aeroelástica, por lo que serían de gran ayuda para plantear ciertos casos. No obstante, tienen la desventaja de que a medida que se alejan del punto sobre el que se ha producido la linealización, estos pueden llegar a perder mucha resolución. Por lo tanto, la representación total de la máquina en base a los mismos sería muy compleja, teniendo que interpolar en todo el rango pudiendo perder

exactitud.

2.6.3. Modelo simplificado

En otros ámbitos, como el estudio de funcionamiento de los motores, puede ser beneficioso contar con otro tipo de modelos que obvien efectos como vibraciones, desplazamientos, o fuerzas locales, que no forman parte del ámbito de influencia. Un ejemplo de creación de modelos específicos que obvian parte de los efectos para centrarse en otros de interés puede encontrarse en (Elbeji et al. [2014]). Basándose en esta filosofía, en este trabajo se desarrolla un modelo que representa con una buena resolución variables clave como las potencias aerodinámicas totales del rotor y la velocidad de este. Este modelo tiene la finalidad de representar el estado operacional de la máquina de forma simplificada en todo el rango de funcionamiento, y se basa en aplicar ecuaciones de la dinámica con parámetros extraidos del modelo aeroelástico, que son alimentados por un viento representativo. La estructura del modelo propuesto se puede ver en la figura 2.18.

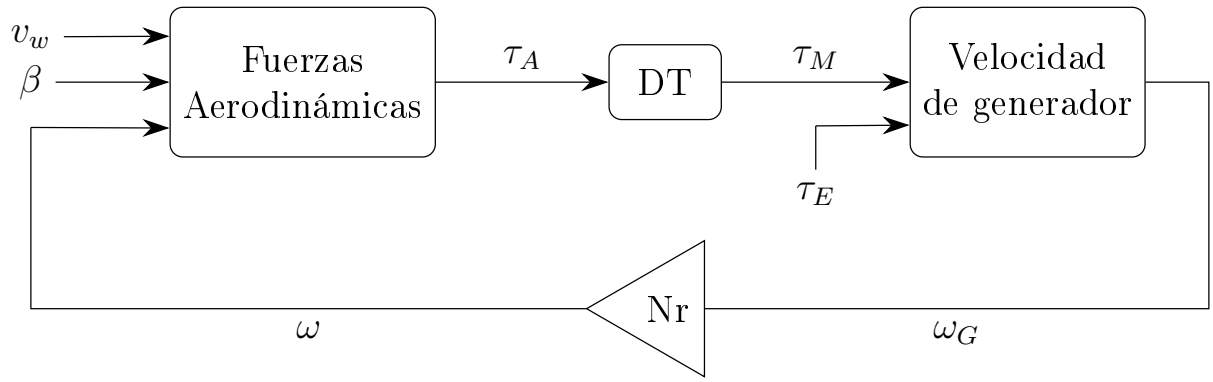

Figura 2.18: Estructura de un modelo de aerogenerador simplificado.

En el modelo propuesto, se toman en consideración la potencia aerodinámica que genera un viento en concreto que se enfrenta a un rotor con un determinado pitch, con lo que se obtendría un par aerodinámico que se presenta en el eje de rotación. Este par se pasa a través de un modelo de drive train, en el que se pueden calcular pérdidas mecánicas, y se obtiene entonces un par mecánico de entrada al eje del generador. Con este par mecánico, y con un par eléctrico, se puede calcular la velocidad de rotación del generador, de la que se puede obtener la velocidad del rotor empleando la relación de transmisión. Esta velocidad de rotor es entonces realimentada a la representación del modelo aerodinámico ya que como se vió el TSR presenta influencia en la adquisición de potencia.

En el modelo de fuerzas aerodinámicas, se implementan las ecuaciones (2.1), y (2.4). Como puede verse en (2.1), esta relación presenta dependencia con ciertas variables. Una de ellas sería un modelo de viento, que se puede extraer de forma a lo propuesto en la sección 2.5.1. La otra sería con el coeficiente de potencia C_p; para obtener este, se pueden lanzar diversas simulaciones con el código aeroelástico para obtener los coeficientes aerodinámicos del rotor

en distinto rango de operación en función del pitch y del TSR, con lo que se obtendría lo conocido como superficie de C_p, que se obtendrá y mostrará más adelante en este trabajo para la máquina elegida para realizar las pruebas. Entonces, si esta superficie se ha obtenido con una resolución suficiente, se podrán interpolar los valores para que el error cometido sea mínimo, con lo que se podrá hallar el Cp con los parámetros disponibles.

El modelo de drive train dependería mucho de la máquina en cuestión, ya que las cajas de cambios son diseños específicos para cada una de ellas. El más sencillo sería el de transmisión directa, en el que se puede aplicar simplemente el rendimiento mecánico de la máquina, o si se quisiera ser muy exacto una rigidez y un amortiguamiento para representar ciertas pérdidas por la deformación del material en operación.

Por último, para obtener la velocidad del generador, se implementa la ecuación (2.7), tomando la potencia que sale del par de arrastre como potencia de entrada, y la potencia que estaría generando el generador como par eléctrico. El coeficiente de inercia rotacional referida al generador, debería ser un parámetro constante y conocido, y los códigos aeroelásticos hacen la suma pertinente de los distintos valores como la inercia de las palas, la inercia del eje, o la inercia del motor; otorgando un valor total de inercia que sería el que hay que introducir en este campo.

Este modelo, como ya se dijo, presenta las ventajas de representar fielmente ciertos comportamientos de la máquina ante un viento realista con un coste computacional mucho más bajo que los códigos aeroelásticos. Mientras que las simulaciones en un código aeroelástico pueden durar varios minutos, en este modelo se resume a unos pocos segundos, lo que los hace sobresalir en ciertas tareas como comprobación de secuencias de control, validación de máquinas de estados, o como lo que se pretende en este caso, hacer un gran número de iteraciones para representar unas dinámicas reducidas. Además, tienen la ventaja respecto a los modelos linealizados de poder representar el funcionamiento de estas dinámicas de forma continua para todo el rango de operación. En la sección 4.1, se expondrá el modelo concreto que se ha creado para la máquina de referencia de este trabajo y el camino que se ha seguido para obtener los parámetros necesarios en función del código aeroelástico.

Capítulo 3

Aprendizaje por refuerzo

En el estudio de la Inteligencia Artificial, existen numerosos campos, desde el estudio de modelos predictivos, hasta el procesamiento de imágenes, pasando por los modelos de lenguaje y los modelos generativos, teniendo algunas de estas aplicaciones un uso interesante en la ingeniería de control (Brunton and Kutz [2019]). Uno de los más relevantes desde el punto de vista de los controladores, es el del aprendizaje por refuerzo (Sutton and Barto [2018]). El aprendizaje por refuerzo (Reinforcement Learning,RL), son modelos de inteligencia que emulan los principios de aprendizaje biológico de los seres vivos, tomando noción de sus acciones y evaluando como estas afectan a su entorno y sus intereses. Con estas técnicas se han conseguido hitos complejos, como entrenar modelos para vencer a humanos en juegos por equipos (OpenAI et al. [2019]), y han producido avances en la robótica pudiendo ejercer control sobre sistemas muy complejos sobre los que sería muy complicado coordinar los diferences controladores usando los métodos tradicionales (Djeffal et al. [2024]).

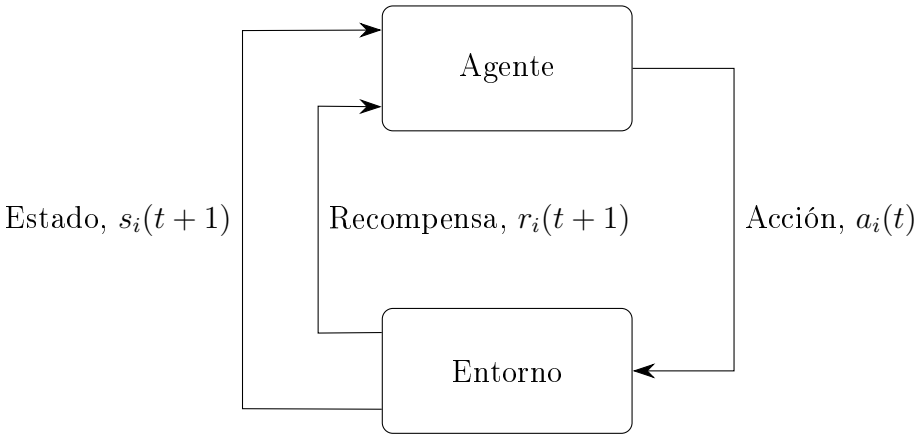

Figura 3.1: Esquema básico de la interacción en el aprendizaje por refuerzo.

El esquema básico de los modelos de aprendizaje por refuerzo se muestra en la figura 3.1. En ellos el controlador, conocido como agente en la literatura, toma la decisión de realizar una acción a_i en el instante determinado t. Esta acción, impactará de alguna forma en

un entorno, cambiando su estado s_i en el siguiente momento de evaluación $t+1$. Además, dependiendo de los cambios producidos, se producirán recompensas r_i para el agente, de forma que éste pueda valorar si el curso de acción que ha tomado es positivo o no.

Los agentes, de la misma forma que los seres vivos, tratarán de obtener la máxima recompensa por sus acciones aprendiendo políticas de control. Por lo tanto, con el paso del tiempo, tenderán a desarrollar estrategias que maximicen las ganancias y minimicen los castigos. Estos se obtienen de una evaluación matemática que ha de establecer el diseñador del sistema. Por ejemplo, en el clásico ejemplo de control del péndulo invertido, se puede otorgar una recompensa en función del ángulo del mismo de tal forma que sea mayor cuanto más alineado con la vertical esté; con esto el sistema, si tiene una capacidad de aprendizaje correcta, comandará a sus actuadores para obtener la máxima cantidad posible de recompensa. Por ello, puede intuirse que un diseño correcto de las recompensas es una de las claves para que estos sistemas funcionen correctamente y, sobre todo, de forma segura (Amodei et al. [2016]).

Puede decirse entonces que el objetivo, matemáticamente hablando, es alcanzar una política de control en el agente que maximice la recompensa esperada a lo largo del tiempo (3.1) para una trayectoria de acciones $\tau = (s_0, a_0, s_1, a_1, \dots)$, que serían una secuencia de estados provocados por acciones que pueden ser determinísticas o probabilísticas.

$$\pi^* = \arg\max \underset{\tau\sim\pi}{E}[R(\tau)] \tag{3.1}$$

Donde $\pi*$ es la política óptima, E es el retorno esperado, y R son las recompensas. Para alcanzar este óptimo, no sólo bastaría con fijarse en el instante temporal en el que se encuentra el sistema en el momento de evaluación, sino que debe aprender a predecir los efectos que tendrán sus acciones más allá, preferiblemente con un factor de horizonte infinito. Esto matemáticamente es un problema complejo que llevaría a una suma infinita. Para ello, se emplea una estrategia de descuento para el sumatorio de recompensas en la que el valor pierde peso cuanto más alejado se encuentra del punto temporal en estudio (3.2).

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{3.2}$$

Donde $\gamma \in (0, 1)$ es el factor de descuento que se aplica a futuro para asegurar la convergencia de la suma, y r_t es la recompensa de cada instante temporal prevista. Tomando ahora el resultado esperado de la trayectoria estado-acción (3.3), que es el resultado esperable de ejecutar una acción a en un estado determinado s.

$$Q^\pi(s, a) = \underset{\tau\sim\pi}{E}[R(\tau)|s_0 = s, a_0 = a] \tag{3.3}$$

Conociendo esto se puede definir la ecuacion de Bellman (3.4), que proporciona la recompensa inmediata esperable $r(s, a)$ de ejecutar una acción a' en un estado determinado s,

y la recompensa futura esperable ponderada por el factor γ.

$$Q^{\pi}(s,a) = \underset{s' \sim P}{E}\Big[r(s,a) + \gamma \underset{a' \sim \pi}{E}[Q^{\pi}(s',a')]\Big] \tag{3.4}$$

El término $\underset{a' \sim \pi}{E}$, define las recompensas esperadas por las acciones y los estados que se obtendrán en el futuro y que vienen definidos por una trayectoria en la que cada acción conlleva a un estado o a un estado probable siguiendo un proceso de decisión de Markov (Markov Decision Process,MDP). Mientras que $\underset{s' \sim P}{E}$, se refiere al siguiente estado o estado probable definido por la acción actual.

Definiendo ahora los valores de función como (3.5), que es el valor esperado de estar en un estado determinado s, y actuar siempre respecto a la política de actuación π.

$$V^{\pi}(s) = \underset{\tau \sim \pi}{E}[R(\tau)|s_0 = s] \tag{3.5}$$

Y obteniendo la ecuación de Bellman del valor (3.6), que daría el resultado actual y el esperable de seguir la trayectoria definida por la política π.

$$V^{\pi}(s) = \underset{\substack{a \sim \pi \\ s' \sim P}}{E}[r(s,a) + \gamma V^{\pi}(s')] \tag{3.6}$$

Se puede obtener el valor de la función de ventaja (3.7) que corresponde a la política π, y que describe como es de bueno tomar una acción específica a en un estado s, asumiendo que se actuará según π de forma infinita.

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{3.7}$$

Este valor es clave en los métodos de gradiente tal y como el que se explora en este trabajo. En ellos, el entrenamiento de las inteligencias artificiales se centran en minimizar la diferencia entre las predicciones que han sido ofrecidas previamente y los resultados de interactuar con el entorno.

Para que lo anterior funcione, los Actores necesitan adquirir experiencia, y para ello es necesario que tengan una serie de datos de anteriores ejecuciones para procesar y aprender de ellas con el fin de extrapolar las anteriores relaciones dentro de las redes neuronales. En contraposición con otros sistemas de inteligencia artificial, en la que los entrenamientos se hacen con una serie de datos que se han obtenido con anterioridad, en el caso del aprendizaje por refuerzo es muy común dejar que los actuadores interactúen con el entorno para generar la información necesaria en base a su experiencia. Para ello, normalmente en entornos virtuales se suelen inicializar las redes neuronales de forma aleatoria, para a continuación realizar una serie de simulaciones en las que realizan acciones aleatorias o pseudoaleatorias. En etapas tempranas del aprendizaje, se necesita que estas acciones tengan una aleatoriedad mayor, ya que se considera que el sistema es inexperto y por lo tanto necesita un número de experiencias

más variado para abstraer el problema y establecer una tendencia. Por la contra, una vez el sistema tiene ya un cierto grado de conocimiento, estas acciones han de ser cada vez menores en número y rango permitiendo explorar zonas cerca del óptimo descubierto (Sutton and Barto [2018]). Esto es conocido como el dilema de exploración-explotación, y propone que ha de existir un equilibrio entre estos términos para un aprendizaje correcto. Si la exploración es demasiado alta el aprendizaje no será capaz de converger, ya que estará ejecutando bandazos de forma continua. Por la contra, si este parámetro es demasiado bajo, el agente es probable que opte por una estrategia hedonista en la que se dedique a actuar en su zona de confort. Lo más común para afrontar este problema, es ir reduciendo el parámetro de ruido con una estrategia denominada ϵ-greedy, que consiste en multiplicar el valor del ruido por un escalar ϵ que se va reduciendo con cada paso de entrenamiento. Cabe reseñar, que existen también modelos que se entrenan a partir de datos externos como los que podría generar otro controlador basado por ejemplo en PID, datos de operadores humanos, etc. esto se conoce como aprendizaje basado en expertos. Esto es especialmente interesante en entornos con una curva de aprendizaje especialmente compleja al principio, pudiendo aprender de las acciones que toman otros que ya operan en un entorno con conocimiento, y una vez ha alcanzado un nivel de entrenamiento suficiente, pasar a entrenarse por su cuenta o directamente ejecutarse en el entorno real.

En (OpenAI [2025a]) se puede ver un esquema taxonómico simplificado de los algoritmos de aprendizaje por refuerzo y que se reproduce en la figura 3.2. Como puede observarse existen dos grandes familias, una que requiere del modelado del entorno y otra que no. Los que tienen el modelo disponible, cuentan con la ventaja de poder adivinar con más exactitud lo que va a pasar en el futuro en función de sus acciones, no obstante, el requerimiento de un modelo puede plantear ciertos retos como que ante ciertas variaciones entre el modelo con el que se ha aprendido y el modelo real puede llevar a comportamientos subóptimos en implementaciones reales.

Respecto a la familia que funciona libre de modelos, esta se divide también en dos subgrupos, el de algoritmos que se basan en la optimizacion de la política, y algoritmos Q-Learning. Los algoritmos que se basan en la optimización de la política, representan una política de forma explícita $\pi_\theta/(a|s)$, y se dedican a optimizar sus parámetros mediante gradientes para transformar esta política en óptima; siendo estas optimizaciones llevadas a cabo normalmente sólo con datos tomados de la última política disponible. Por otra parte, los algoritmos basados en Q-Learning, se centran en aprender un aproximador $Q_\theta(s, a)$ para la función óptima de acción-valor, utilizando tipicamente una función objetivo basada en la ecuación de Bellman. Sus optimizaciones suelene llevarse a cabo sin política determinada, lo que significa que cualquier punto obtenido durante el entrenamiento puede emplearse para el entrenamiento.

Las aproximaciones de optimización de política y Q-Learning pueden coexistir, ya que no son incompatibles, existiendo un rango de algoritmos que se sitúan entre ambas aprovechándose de sus fortalezas. Entre estos, se sitúa el Deep Deterministic Policy Gradient

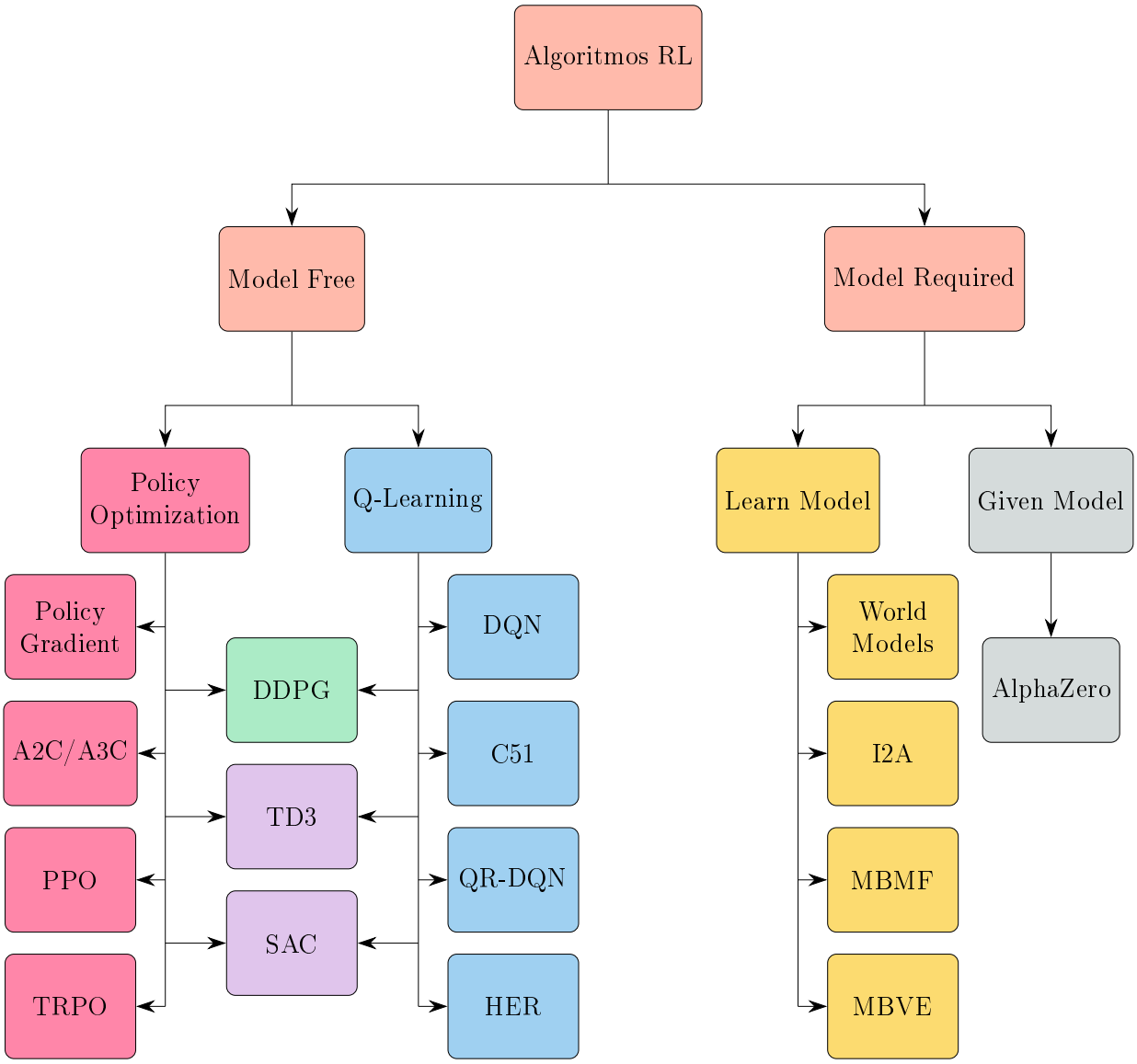

Figura 3.2: Taxonomía simplificada de los métodos de aprendizaje por refuerzo.Extraído de (OpenAI [2025a]).

(DDPG), resaltado en verde en la figura 3.2. Este algoritmo es el empleado en este trabajo y se profundizará sobre él en la siguiente sección.

3.1. Deep Deterministic Policy Gradient

Como se vio en la sección anterior, el algoritmo Deep Deterministic Policy Gradient (Lillicrap et al. [2019]), convive entre la optimización de política y el Q-Learning. Es un algoritmo capaz de manejar numerosos estados y acciones, siendo además capaz de trabajar en espacios continuos, en contraposición a otros que sólo pueden manejar acciones discretas como el Deep Q Network. El DDPG cuenta con dos redes neuronales, un Actor que tiene una política independiente, y un Crítico que evalúa las acciones del Actor y retroalimenta su pérdida mediante la estimación de los valores de recompensa obtenidos por la ecuación de Bellman mostrada en (3.4).

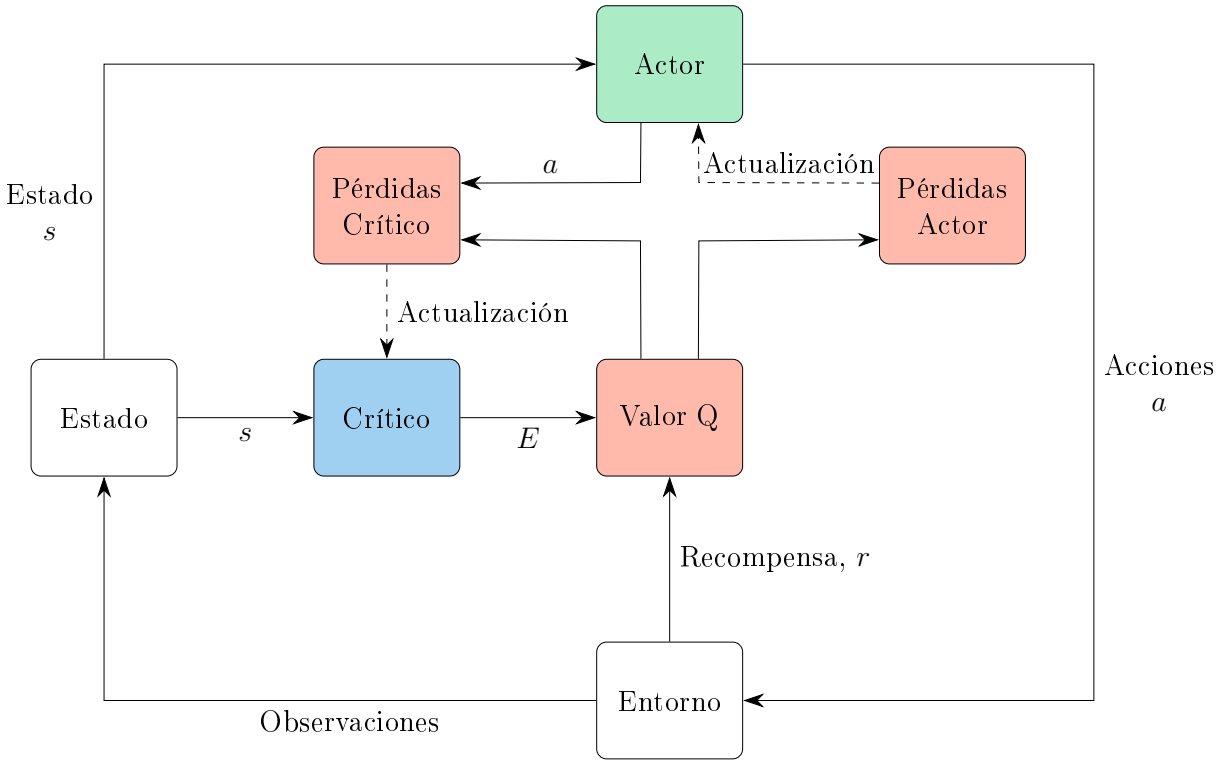

Figura 3.3: Esquema de funcionamiento del controlador DDPG.

El esquema de funcionamiento de las interacciones que se producen en un controlador DDPG se puede ver en la figura 3.3. En él, un Actor genera acciones a que impactan sobre un entorno, y que producen un cambio de estado S en el mismo. A este cambio de estado, se asocia una recompensa r en función del criterio del diseñador. Entonces, la red del Crítico estima el valor de Bellman que habría de producirse para ese cambio. Esa estimación se compara con la recompensa real y se produce una evaluación de las pérdidas del Crítico que relacionan la exactitud de la predicción que éste ha ofrecido, y con esta evaluación se produce

una actualización de la red para que corrija su error. De la misma forma, apoyándose en las estimaciones del Crítico, se produce una evaluación de las pérdidas del Actor, que califican las consecuencias de la acción que se ha tomado en términos de recompensa, y actualizan la red para que esta en base a su política tome mejores decisiones en el futuro.

Durante el entrenamiento, el proceso anteriormente citado se va repitiendo un número indefinido de veces. Para cada una de las iteraciones, los resultados del estado anterior s_{i-1}, estado actual s_i, acciones que han llevado el entorno al estado actual desde el anterior a_i, y recompensas r_i, se almacenan comunmente en un buffer, conocido normalmente en la literatura como Replay Buffer. Este buffer se emplea para que en cada paso de entrenamiento, las redes neuronales tengan un número de experiencias de las cuales aprender. La longitud de este buffer ha de ser lo suficientemente grande como para poder contar con experiencias de carácter variado, unas buenas y otras malas, y deseablemente con una zona de exploración del entorno lo más grande posible para poder abstraer el problema de la mejor forma posible. Elegir el tamaño del buffer no es una cuestión trivial, ya que en el caso de almacenar sólo las últimas experiencias, las redes tenderán a hacer un overfit que puede conducir a un olvido de la tarea y perder las habilidades que habían adquirido, lo que en la literatura se conoce como catastrophical forgetting, de lo que se pueden ver ejemplos en (Asperti and Brutto [2022]). Por la contra, un tamaño de buffer demasiado grande, tenderá a ralentizar el entrenamiento ya que las experiencias muy tempranas no tienen interés una vez el modelo ha comenzado a converger. Basándose en la información del buffer, para cada paso de entrenamiento, se seleccionan experiencias aleatorias que se almacenan y que estarían marcadas por el conjunto $R[i] = \{s_{i-1}, s_i, a_i, r_i\}$, en el que R representa el buffer, a y s son vectores que pueden tener distinta longitud, y r sería el escalar de la recompensa. Entonces, con la selección de N recompensas se alimentará al proceso de entrenamiento para que las redes neuronales que forman el Crítico y el Actor puedan actualizarse convenientemente.

A pesar de que hasta ahora se haya hablado de Crítico y Actor como una sóla red neuronal, durante el entrenamiento de este tipo de inteligencias artificiales, se crean en realidad un par de redes neuronales unas conocidas como objetivo y otras como modelo. Es decir, coexistirían dos Actores, un Actor modelo y un Actor objetivo; y dos Críticos, un Crítico modelo y un Crítico objetivo. La razón que subyace a esta decisión es que si los cambios de cada paso de entrenamiento se aplicaran sobre sólo una red para cada uno de los elementos, éste sería altamente inestable y la convergencia sería imposible en muchos casos, ya que la optimización que se lleva a cabo para la red neuronal depende de los mismos valores de esa red. Entonces, existirá una red de cambio rápido sobre la que se produce el entrenamiento directo, y una de cambio lento que se va actualizando en función de los valores de esta ponderados por un factor conocido como soft update $\tau \in (0, 1)$, pero con un valor normalmente próximo a 0 en el orden de 10^{-3}. Las redes objetivo, que se conocen así porque son el objetivo a minimizar, serían las de cambio lento, mientras que las redes de modelo sobre el que se produce el entrenamiento, serían las de cambio rápido. Entonces, la actualización de los

pesos de cualquiera de las redes neuronales se producirá de la forma descrita por (3.8).

$$\theta_{Objetivo}[i,j] = \tau\theta_{Modelo}[i,j] + (1-\tau)\theta_{Objetivo}[i,j] \tag{3.8}$$

Estabilizando el entrenamiento de la forma descrita, entonces se puede pasar a hablar de las funciones a minimizar. Dado que se pueden ver las redes objetivo como redes que se van retrasando respecto a las de modelo, se podría establecer una analogía por se puede considerar como cuando se introduce un retraso para romper un loop algebraico.

La función de pérdidas a minimizar del Crítico estará basada entonces el error cuadrático medio de la diferencia entre la recompensa estimada por el Crítico objetivo, y la recompensa obtenida del Crítico modelo (3.9). Esta función también se refiere en la literatura como Mean-Squared Bellman Error (MSBE), ya que está basada en la obtención de los valores de la función de Bellman, y se suele intentar resolver su minimización con métodos estocásticos de gradiente descendente.

$$L = \frac{1}{N}\sum_{i}^{N}\Big(r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) - Q(s_i, a_i|\theta^{Q})\Big)^2 \tag{3.9}$$

Donde L es la función de pérdida, N es el número de valores que se han empleado para obtener las diferentes estimaciones,γ es el factor de descuento para recompensas futuras, μ' es la política actual del Actor, Q' es la red neuronal objetivo del Crítico, y Q la red neuronal modelo del Crítico. Cabe destacar, que pese a lo compleja que pueda parecer la nomenclatura, el primer término de la diferencia $r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'}))$, es simplemente la recompensa obtenida del entorno más la futura estimada por la red neuronal objetivo del Crítico y se correspondería con un escalar, mientras que el segundo término $Q(s_i, a_i|\theta^{Q})$, se corresponde con la recompensa que ha generado la red neuronal modelo del Crítico, lo que sería otro escalar; con lo que realmente la implementación de la fórmula es mucho más sencilla de lo que puede parecer. Cabe reseñar también, que en el Crítico objetivo, se introduce la acción que produciría el Crítico del Actor, mientras que en el Crítico modelo, se introduce la acción que realmente se ha producido. De esa forma también se es capaz de valorar el efecto de la diferencia de acciones entre la proporcionada por la política actual y la que se introduce, lo que será clave para el entrenamiento de la red del Actor.

La función objetivo en el caso del Actor, vendrá dada para obtener una política de control determinística que devuelva acciones que maximicen la recompensa (3.10).

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_{i}\nabla_a Q(s, a|\theta^{Q})|_{s=s_i, a=\mu(s_i)}\nabla_{\theta^{\mu}}\mu(s|\theta^{\mu})|_{s_i} \tag{3.10}$$

Donde $\nabla_{\theta^{\mu}} J$ es el gradiente a ser maximizado, y $\nabla_a Q(s, a|\theta^{Q})|_{s=s_i, a=\mu(s_i)}$ es el valor de recompensa estimado por el modelo del Crítico. Para trabajar con esta función desde el punto de vista de la optimización, lo que se hace entonces es minimizar su negativo empleando también métodos estocásticos de gradiente descendente al igual que con el Crítico, por lo

que se puede definir la función de pérdida del Actor como (3.11).

$$L_a = -\frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i} \tag{3.11}$$

Como se mencionó anteriormente, para que todo esto funcione, el modelo DDPG ha de explorar el entorno para de esa forma poder obtener la información necesaria que le permita abstraer el problema y desarrollar un comportamiento adecuado. Con ese fin, en cada paso de iteración se toma una acción y se realiza una exploración, de forma que esta sea la acción que se obtiene de la política actual del Actor modelo más un valor aleatorio (3.12).

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t \tag{3.12}$$

Este valor aleatorio puede ser de diverso origen. Los autores originales en (Lillicrap et al. [2019]) han empleado un proceso de ruido de Ornstein-Uhlenbeck, pero pueden emplearse muchos otros como ruido gausiano. Como se también se citó con anterioridad, para hacer la adición del ruido más óptima, es costumbre común tener esta en un valor alto al principio, para realizar una exploración mayor, mientras que va cayendo a futuro para permitir explorar puntos más cercanos al óptimo. Entonces el ruido se multiplicaría por un factor ϵ (3.13).

$$a_t = \mu(s_t|\theta^\mu) + \epsilon\mathcal{N}_t \tag{3.13}$$

Este factor, se puede ir actualizando a placer durante cada iteración, y una costumbre es multiplicar por un número cercano a 1 para ir decayendo de forma asintótica hacia 0, por ejemplo, $\epsilon_i = 0{,}999\epsilon_{i-1}$, para una mayor claridad se ha hecho una representación en la figura 3.4 con distintos ratios de decaimiento para un mísmo número de pasos de entrenamiento.

No obstante, existe otra propuesta que en ciertos escenarios de aprendizaje ha resultado conseguido mejorar de forma muy visible los resultados, acelerando la convergencia de los modelos y consiguiendo resultados mejores que la mencionada (Plappert et al. [2018]). Esta estrategia consiste en añadir un ruido gausiano a los pesos de la red neuronal del Actor, de forma que éste genere una acción de exploración levemente distinta a la que generaría en el estado normal como puede verse en la figura 3.5. Estos pesos entonces se asignan a una red de Actor ruidosa auxiliar con la misma estructura que las redes modelo y objetivo, y se le proporciona la información del estado para generar la acción modificada (3.14).

$$\theta^\mu_{noise}[i,j] = \theta^\mu[i,j] + \mathcal{N}(0,1)S_f \tag{3.14}$$

Para definir la cantidad de ruido deseada, se emplea S_f como factor de escala, de esa definiendo una distancia δ entre las acciones deseadas y las realizadas, este factor puede ser modificado por un parámetro α con el fin de ir introduciendo mayor o menor cantidad de ruido en la red como podría realizarse siguiendo la expresión (3.15).

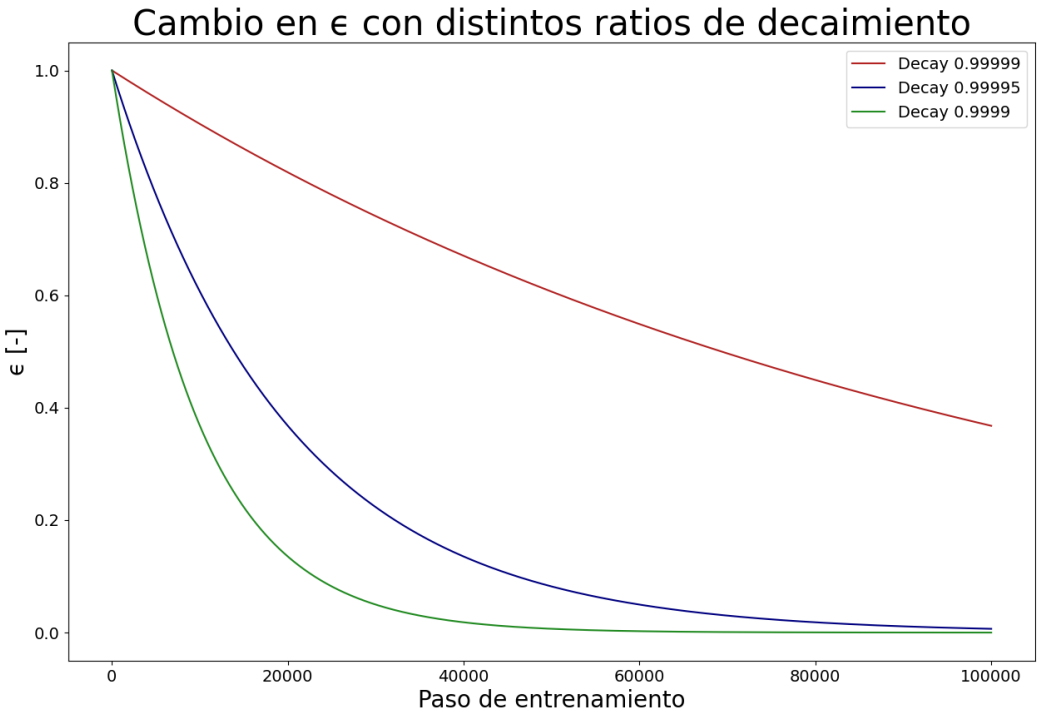

Figura 3.4: Variación del valor de amortiguamiento de ruido ϵ en función del ratio de decaimiento.

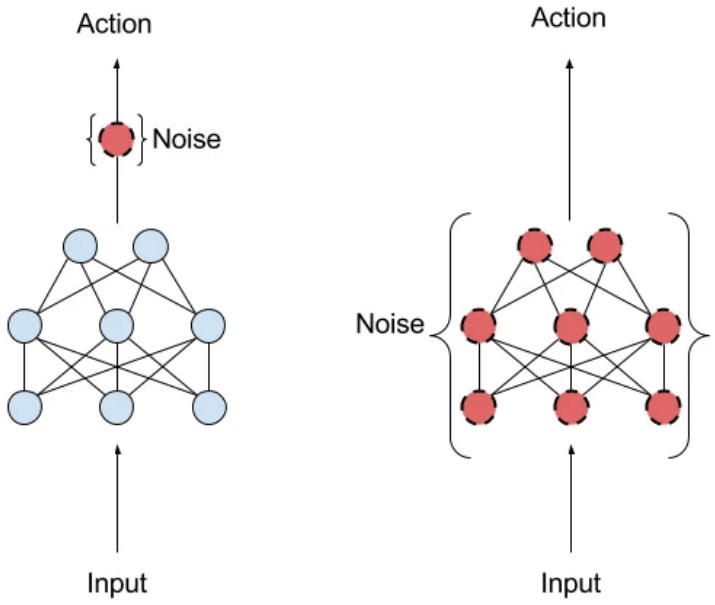

Figura 3.5: Esquema de la variación de ruido en parámetros. Imagen extraída de (OpenAI [2025b]).

$$S_{f_{k+1}} = \begin{cases} \alpha S_{f_k} & \text{si } \sqrt{\frac{1}{N_a}\sum_i^{N_a}(a_i - \widetilde{a_i})^2} \leq \delta \\ \frac{1}{\alpha} S_{f_k} & \text{en otro caso} \end{cases} \tag{3.15}$$

Con el fin de conjuntar lo anteriormente explicado, se expone a continuación en el pseudocódigo 1 un extracto del artículo publicado por los autores originales, en el que se muestra el proceso de entrenamiento de un modelo DDPG para un número de episodios determinado M que duran un tiempo T. En él se puede ver como se inicializan las redes neuronales y el buffer en un primer momento, para luego entrar en un bucle para el número de episodios, dentro del cual se entra en otro bucle para el tiempo de simulación y en el que se van ejecutando acciones, guardando los resultados en el buffer, y realizando el entrenamiento.

Algorithm 1 DDPG, extraído de la propuesta original de (Lillicrap et al. [2019])

→Inicializar de forma aleatoria la red del Crítico $Q(s, a|\theta^Q)$ y del Actor $\mu(s|\theta^\mu)$ con pesos θ^Q and θ^μ.
→Inicializar las redes objetivo Q' y μ' con pesos $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
→Inicializar el buffer de experiencias previas R
for episodio = 1, M **do**
→Inicializar un proceso aleatorio $\mathcal{N}$ para la exploración de acciones
→Obtener el estado inicial de las observaciones s_1
for t = 1, T **do**
→Seleccionar una acción $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ de acuerdo con la política actual y el ruido de exploración
→Ejecutar la acción a_t y observar la recompensa r_t y el nuevo estado s_{t+1}
→Almacenar la transición (s_t, a_t, r_t, s_{t+1}) en el buffer R
→Tomar de forma aleatoria un minibatch de N transiciones (s_i, a_i, r_i, s_{i+1}) de R
→Establecer el valor $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
→Actualizar el Crítico minimizando la función de pérdidas:

$$L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$$

→Actualizar la política del Actor utilizando el gradiente de la política muestreado:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

→Actualizar las redes objetivo:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

end for
end for

Por último, cabe hablar sobre las ventajas y desventajas que puede presentar la utilización de este tipo de modelos basados en DDPG. La principal ventaja, es que tiene la capacidad de trabajar con espacios continuos, lo que ofrece un control fino de las operaciones y la capacidad de enfrentarse a perturbaciones en caso de tener una convergencia de modelo adecuada. Además, es capaz de manejar diversas acciones al mismo tiempo, lo que lo hace ideal para estrategias de control multiobjetivo en las que se pueden coordinar esfuerzos de distintos actuadores. Las principal desventaja sería que es un modelo muy dependiente de encontrar valores correctos para los hiperparámetros, tanto a nivel de convergencia como de funcionamiento final. Además entrenar modelos de este tipo con grandes redes neuronales puede ser computacionalmente costoso y llevar un largo tiempo.

3.2. Aplicación al control de aerogeneradores de modelos de aprendizaje por refuerzo

La aplicación de los modelos de aprendizaje por refuerzo es una técnica relativamente nueva en el campo de la ingeniería eólica. Esto se debe a múltiples factores, el primero de ellos es que el propio campo de la inteligencia artificial ha sufrido una amplia transformación en los últimos años, en la que se han producido numerosos desarrollos auspiciados por el auge de las tecnologías de la información y las capacidades computacionales, haciendo que el crecimiento de los modelos y las opciones para trabajar con ellos crezca de forma exponencial. Otro factor podría ser la escasa preparación que tienen los códigos aeroelásticos que se mencionaban anteriormente, dado que la interfaz de control se suele hacer mediante una librería auxiliar, exigiría un desarrollo importante el poder introducir en una de ellas un modelo como el que se ha expuesto en este capítulo para su entrenamiento. Por último, el desarrollo de estas máquinas es un proceso iterativo y costoso que puede llevar años de trabajo de un equipo, por lo tanto los cambios que se producen en el mundo académico pueden tardar en aparecer en la industria. De hecho, según el conocimiento del autor, no existe ninguna máquina comercial que implemente un algoritmo de inteligencia artificial en sus capas de control principales; no obstante, esto es probable que cambie en los próximos años.

En el ámbito académico, por la contra sí que se puede encontrar ya un florecimiento de estas técnicas aplicadas a los modelos de control de los aerogeneradores. Por ejemplo en (Kushwaha et al. [2020]) y en (Wei et al. [2015]), se han desarrollado sendos modelos de seguimiento de MPPT empleando modelos de Q-Learning sin sensores. En (Chen and Han [2023]) se desarrolló un modelo DDPG para cotrolar la máquina en todo el rango operacional, haciendo un tracking del TSR durante todo el período MPPT, y utilizando pitch cuando la se entra en la zona de control de potencia nominal. En (Zholtayev et al. [2024]) se creó un control para un motor sincrono de imanes permanentes de un aerogenerador para ejercer el control de par en MPPT utilizando el algoritmo TD3. En (Muñoz-Palomeque et al. [2022]),

se exploran técnicas híbridas con métodos de aprendizaje no supervisado para obtener un mejor seguimiento del punto de potencia.

En el apartado del control de potencia nominal se pueden también encontrar algunos ejemplos. En (Sierra-Garcia et al. [2022]) se propone un control híbrido con generación de referencias con un modelo de aprendizaje por refuerzo. En (Coquelet et al. [2022]), se exploran técnicas para ejercer un control de pitch individual en las palas con el fin de obtener unas cargas más equilibradas. Se han evaluado también técnicas de diseño de recompensas para controladores de pitch en potencia nominal (Sierra-García and Santos [2020]).

Como puede verse, muchos de los artículos citados se centran en implementaciones de los modelos para resolver problemas en comparación con modelos clásicos. Otros emplean modelos híbridos para que el control generado por el aprendizaje por refuerzo genere las referencias a controladores clásicos como los PID. en este trabajo se optará por la primera vía, y el control clásico será reemplazado por un control íntegramente compuesto por un desarrollo de modelo DDPG.

Capítulo 4

Desarrollo e implementación de modelos de simulación

En este capítulo se abordará el desarrollo de los modelos de entorno de aerogenerador y viento necesarios para implementación del modelo de controlador DDPG. Con el fin de representar ambos, se ha contado con el software de Matlab/Simulink (The MathWorks Inc. [2024]) en su versión 2024b, implementando éstos sobre un mismo fichero. Con el fin de obtener valores necesarios para ellos, en ocasiones será necesario recurrir a un código aeroelástico, para lo que se usará OpenFAST. OpenFAST es un programa de código abierto creado por el centro de investigación NREL que combina varios módulos para hacer simulaciones completas de máquina eólica.

Durante el desarrollo del presente capítulo y del siguiente, se desarrollarán cuestiones técnicas de la adaptación de los términos previamente expuestos y, para no oscurecer éstas, la implementación real en código se añadirá en anexos que se irán citando, ya que por la longitud y la complejidad del código podrían hacer el documento de difícil lectura si se incluyeran de forma secuencial dentro del mismo.

En la figura 4.1, se puede ver un esquema básico de los distintos módulos que se han creado y de su jerarquía de funcionamiento. En la figura, los módulos que se ejecutan en Python están en verde, mientras que los que se ejecutan sobre base Matlab aparecen representados en color salmón. En ella se distinguen dos grupos de módulos, el de la izquierda que estaría compuesto por aquellos que componen el modelo de controlador, que se expondrá en el capítulo 5; y en la derecha los que componen el modelo de entorno, que son los que se tratarán en el presente capítulo. El punto de entrada y eje sobre el que gira toda la ejecución, sería el punto denominado como entorno de entrenamiento, que es el módulo que contiene las redes neuronales del controlador y a partir del que se dirige su secuencia de entrenamiento. Éste cuenta con herramientas permiten invocar al resto de funcionalidades. Una de ellas será el Buffer, en éste como se comentó en la sección 3.1, se irán almacenando las experiencias previas y entrenando las redes neuronales del modelo. Otro de los módulos que sirven al

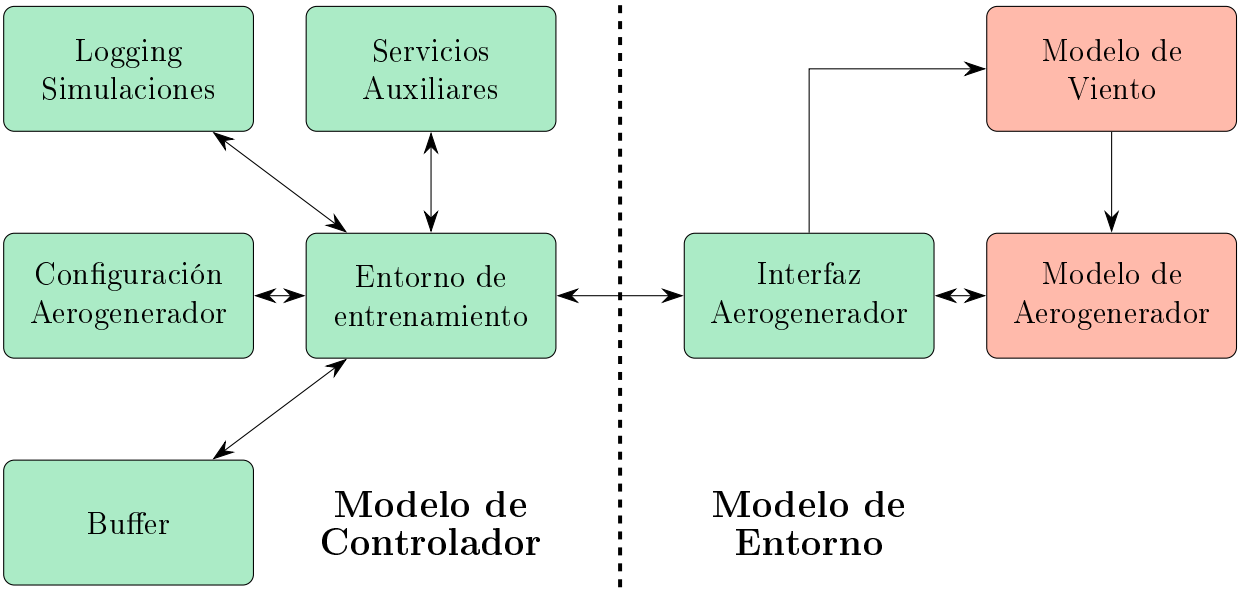

Figura 4.1: Esquema general de los distintos módulos de la implementación.

principal será el que se denomina en el esquema como de Servicios Auxiliares; en él se han implementado acciones que permiten la gestión de operaciones basadas en la red neuronal, tales como inicializaciones o backups para poder trabajar con ella de forma cómoda. El módulo auxiliar de Configuración Aerogenerador se empleará para generar la configuración inicial de los parámetros fijos de las simulaciones, así como para obtener los parámetros de inicialización de las mismas. El módulo de Logging Simulaciones permite el almacenamiento y visualización de los datos de las simulaciones a medida que éstas se van produciendo. Uno de los módulos clave, será el de Interfaz Aerogenerador, éste se encargará de iniciar una instancia de Matlab dentro de la ejecución de Python, e irá ejerciendo de intermediario para la ejecución entre ambos entornos, pasando órdenes y datos entre uno y otro. Los bloques de Modelo de Viento y Modelo de Generador, se ejecutan sobre el mismo modelo de Simulink, y mientras el modelo de viento se inicializa en el primer paso de cada simulación y funciona de forma autónoma, el modelo de aerogenerador irá interactuando con la interfaz recibiendo órdenes de simulación y parada, generando así los datos necesarios para la ejecución del controlador.

A lo largo de este capítulo se irán desgranando estos últimos bloques dedicados al modelado físico, ofreciendo los pormenores del funcionamiento de cada uno de ellos.

4.1. Implementación del modelo de máquina

La máquina que se ha tomado como referencia es la IEA 15MW, y los parámetros de la misma que tienen relevancia para el modelo se listan en el apéndice A con su valor correspondiente.

Para crear un modelo de la misma apto para simulación se toma como idea base el explicado en la sección 2.6.3, en la que se introducían modelos simplificados de máquina. Éste se utiliza por varias razones, la primera es que el código aeroelástico no cuenta con la capacidad de implementar directamente el control sin realizar un desarrollo muy extenso. Otra de las razones vendría dada por el incremento significativo del coste computacional que implicaría realizar las simulaciones teniendo en cuenta todos los efectos aeroelásticos. Además, como se comentó anteriormente, este tipo de modelo cuenta con la ventaja de poder representar ciertos valores de la máquina de forma fehaciente en todo el rango de operacional.

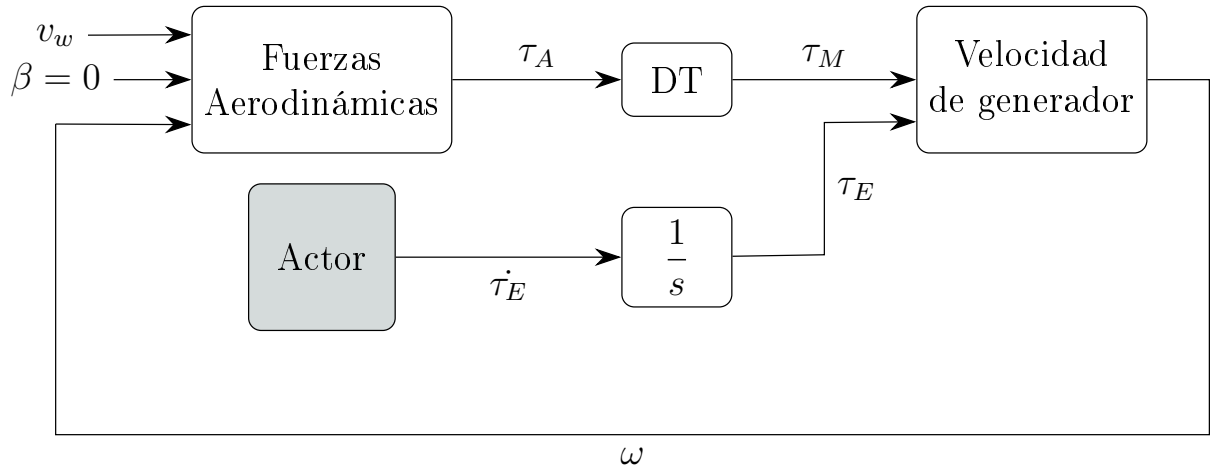

Figura 4.2: Estructura del modelo de aerogenerador implementado.

En la figura 4.2 se puede ver la implementación particularizada para el modelo de máquina. En ella se incluye la señal de control que generaría el Actor DDPG de forma sombreada, ya que ésta será proporcionada a través de una interfaz y se le dejará un punto de entrada para que pueda escribir el valor, si bien el Actor no estará realmente presente dentro del modelo de Simulink.

En el sistema de Fuerzas Aerodinámicas se implementará la relación de potencia aerodinámica (2.1). Cabe recordar que ésta depende del viento, que se genera y alimenta a la máquina de forma independiente, y del coeficiente de potencia $C_p(\beta, \lambda)$. En este caso, como lo que se estudia es la zona MPPT de la máquina, el ángulo de pitch se fija a 0 deg en todas las simulaciones ($\beta = 0\,\text{deg}$) como está indicado en la imagen 4.2. Por lo tanto, para tener una respuesta realista del comportamiento de la máquina en su rango de operación, habría que hallar de alguna forma la superficie que relaciona el C_p en función del ángulo de pitch y el TSR. Esto se realiza mediante simulaciones con el módulo aerodinámico de OpenFAST, y las operaciones que hay que realizar para ello se detallan en la subsección4.1.1. Suponiendo que se cuenta con esta superficie, mediante la velocidad de la máquina, que es un dato conocido, y la velocidad del viento, que también lo es, se puede calcular el TSR (2.2). Con este TSR y con el pitch que se le está indicando a la máquina, bien como en este caso que se marca un pitch fijado, pero también valdría para un pitch generado por controlador, se puede obtener

el C_p empleando la superficie con una interpolación doble, cometiendo un error muy bajo en caso de haber obtenido puntos de la curva lo suficientemente próximos. Con este dato ya podría calcularse la potencia aerodinámica, y con una operación trivial el par aerodinámico generado sobre el eje (2.4).

Este par obtenido, pasaría a través de un bloque de drive train. En el caso de la máquina IEA 15MW, ésta cuenta con un sistema direct drive, sin caja de cambios, por lo tanto su relación de transmisión de par y velocidad se supone 1. No se han considerado efectos menores de pequeñas torsiones o deformaciones de material que vendrían dados por la rigidez del mismo, ya que no aportarían información de utilidad al modelo y el impacto en los resultados sería inapreciable. Lo que se hace en este bloque, es aplicar el rendimiento del generador a la potencia aerodinámica para obtener la mecánica (4.1).

$$\tau_M = \eta_G \tau_A \tag{4.1}$$

Si bien esto no sería estrictamente cierto desde el punto de vista físico, dado que lo que se está considerando es la potencia eléctrica útil con un par equivalente. Hacer esta operación en este punto sería equivalente a nivel de cálculo a hacerla en el bloque del generador. Otra opción sería trabajar con el par eléctrico mayorado y multiplicarlo por el rendimiento, pero esa opción era menos limpia desde el punto de vista de cálculo. Se ha tomado también el supuesto de que todas las pérdidas eléctricas se disipan en forma de par y no de velocidad.

En el subsistema de velocidad del generador entonces, se recibe un par mecánico de arrastre T_M y un par eléctrico de freno que viene por comandos generados en el controlador T_E. La particularización entonces para la máquina implicaría tomar la ecuación (2.7), y eliminar el rendimiento y la relación de transmisión, con lo que para este caso esta relación se puede expresar en el dominio de Laplace como (4.2).

$$\omega = \frac{\tau_M - \tau_E}{J(s+b)} \tag{4.2}$$

Nótese que se ha añadido un factor b de amortiguamiento por si se desean aplicar pérdidas de velocidad por rodamientos u otras partes del sistema mecánico; en caso de no hacerlo puede dejarse con valor 0. Además, cabe recalcar que en este caso la velocidad del generador y la del rotor aerodinámico son exactamente las mismas por considerar el eje completamente rígido.

Para trabajar con este modelo se aprovechará la capacidad de Python de poder invocar entornos de Matlab. Para ello se emplea un paquete distribuido en la propia instalación de este último conocido como Matlab Engine for Python. Con estas capacidades, la clase que ejerce como interfaz entre el modelo de control y el entorno de simulación será capaz de crear una sesión de Matlab. Dentro de esta sesión se puede abrir el modelo de Simulink y proporcionarle los valores de configuración necesarios para la ejecución. Para ello los valores constantes de la turbina eólica se han dejado en función de variables del workspace de Matlab.

Además, los valores de los integradores se dejan también en función de un punto inicial para poder inicializar las simulaciones de forma suave en un punto operacional, ya que de otro modo habría un gran desperdicio de tiempo de simulación hasta que ésta convergiera a un punto adecuado, sumándole además el efecto que esos puntos de trabajo serían de nulo interés para el modelo DDPG.

El flujo de trabajo de este modelo será entonces:

- **1 Arranque:** la clase interfaz crea la sesión de Matlab y con ella el modelo una sola vez, ya que el proceso de arranque puede llevar varios segundos, con lo que inicializar una sesión para cada simulación sería terriblemente subóptimo.
- **2 Establecimiento de los valores constantes:** Una vez se ha establecido comunicación, se asignan los valores constantes del modelo, como puede ser el radio de rotor del aerogenerador, o los valores de la superficie Cp. Se establecen también los parámetros de simulación de tiempos inicial, final, y paso temporal. Esta operación se realiza también en una única ocasión.
- **3 Simulaciones:** el primer paso para simular es proporcionar los valores iniciales, que se calculan desde el entorno de entrenamiento y se transmiten a los integradores del modelo de entorno del aerogenerador. Una vez se ha realizado esta operación, se puede ir ejecutando el modelo paso a paso hasta que recibe la orden de reiniciarse con nuevos valores desde la interfaz. Es decir, la simulación es siempre la misma y, para cada episodio nuevo, lo que se hace es reiniciar sus valores.
- **4 Cierre:** si bien no es una cuestión que influya en el desarrollo del proyecto, sí que forma parte del ciclo de vida de la aplicación. La interfaz cuenta con la capacidad de hacer un cierre seguro desde la sesión, que sería equivalente al cierre manual de la aplicación si esta estuviera abierta en modo escritorio por parte de un usuario.

Por último, cabe mencionar que OpenFAST también tiene la capacidad de ser invocado desde Simulink, con lo que las simulaciones podrían funcionar empleando éste como interfaz entre Python y el código aeroelástico. No obstante, en este trabajo se descarta esa opción por considerar el presente modelo lo suficientemente representativo, con lo que no habría necesidad de incrementar la dificultad de operación y mantenimiento, así como el coste computacional que esta decisión implicaría.

4.1.1. Obtención del coeficiente de potencia aerodinámico

La obtención del C_p es un requisito imprescindible para que el modelo propuesto funcione. Hay autores que proponen métodos analíticos como en (Castillo et al. [2023]), en el que se emplea una fórmula para obtener dicho coeficiente en función del pitch y el TSR. No obstante, esos métodos son complejos de ajustar para un modelo de turbina en concreto, y su respuesta

no es buena en muchas ocasiones. Por esa razón este tipo de ajustes no han tenido peso en la industria. Por ello en este trabajo se opta por otro tipo de estrategia, que consiste en obtener el C_p de forma experimental mediante simulaciones, haciendo un barrido para mapear esta superficie en función del pitch y del TSR. El código necesario para realizar esta operación se expone en el anexo C.

Tomando como referencia el modelo de máquina existente en el repositorio IEA15 MW (Gaertner et al. [2020]), es posible lanzar simulaciones con el módulo aerodinámico de OpenFAST, conocido como Aerodyn, para obtener los valores de los coeficientes aerodinámicos sectorizados o totales. Estas simulaciones, se realizan en presencia de un viento constante y con un pitch y una velocidad de rotación prefijadas, obviando otros efectos mecánicos que se producen sobre la máquina tales como fuerzas o desplazamientos, centrándose así sólo en la parte aerodinámica.

Para tener un mapeo efectivo de los componentes, lo que se hizo fue recorrer una lista de valores de pitch $\beta \in [\beta_{min}, \beta_{max}]$, y hacer lo mismo anidando otra lista en la exploración de variables de TSR $\lambda \in [\lambda_{min}, \lambda_{max}]$. Dado que Aerodyn no trabaja de forma directa con el TSR, lo que se hizo fue mantener un viento constante, y modificar la velocidad de rotación de la máquina despejando esta de (2.2), con el fin de obtener este valor según lo mostrado en la ecuación (4.3), generando así un vector de velocidades de rotación que implican diferentes TSR.

$$\omega[i] = \frac{v_w \lambda[i]}{R} \tag{4.3}$$

Una vez cruzadas estas listas, se generan ficheros con formato de inputs de Aerodyn con el fin de poder generar las simulaciones. Dado que el número de éstas era elevado, durante la implementación se paralelizó esta operación para reducir el número de horas necesarias para realizar la operación de obtención de la superficie.

Una vez se han llevado a cabo las simulaciones, éstas se agrupan y se puede obtener la matriz de valores con la que se podrá hacer la interpolación del C_p dentro del modelo. Para hacer un mapeo extensivo de la superficie es necesario contar con pasos relativamente pequeños para no producir error, como ya se dijo. A continuación se presenta en la tabla 4.1 con los valores que se han tomado, y que representan un abanico más que suficiente de datos.

Tabla 4.1: Rangos de interpolación para la superficie de potencia.

Variable	**Valor Mínimo**	**Valor Máximo**	**Paso**
Pitch [deg]	-3	25	0,1
TSR [-]	2	15	0,1

El resultado de esta operación será una matriz de 280×130, teniendo que ejecutar 36400 simulaciones para cubrir el rango deseado. Estas simulaciones, son de escasa duración, ya que el modelo no necesita excesivo tiempo para converger. En este caso se ha marcado como

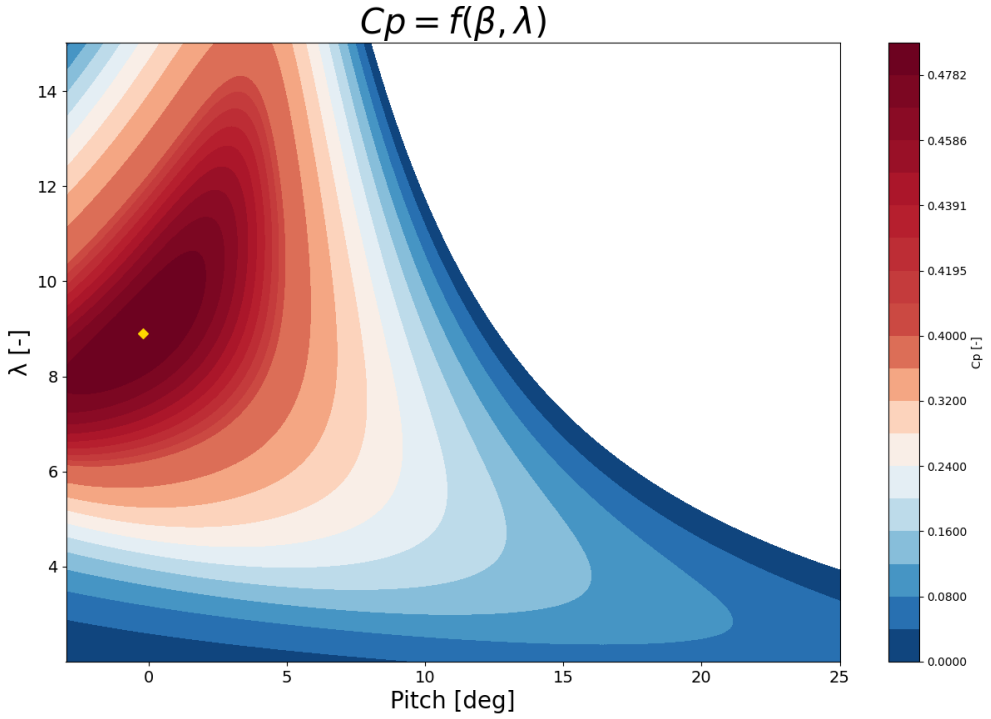

Figura 4.3: Superficie del coeficiente de potencia en función de β y λ obtenida.

12 segundos. Para no depender de la posición azimutal del rotor, que puede inducir pequeñas variaciones, el C_p se toma como el valor medio de toda la simulación.

A continuación, se puede ver un resumen en pseudocódigo de esta operativa para una mejor comprensión en el Algoritmo 2.

Con esta configuración se ha obtenido la superficie que se muestra a en la figura 4.3. En esta figura, el C_p se representa mediante curvas de nivel, y en ella se marca además el punto en el que se ha obtenido el valor máximo de 0,4879. En base a esto se obtienen el resto de valores para los cuales el C_p es máximo, lo cual será clave en el MPPT, ya que habrá que fijar ese óptimo en el pitch, e intentar aproximar el TSR lo mejor posible dentro de las variaciones de viento. Estos valores de pitch y TSR se muestran en la tabla 4.2.

Tabla 4.2: Valores de pitch y TSR para máximo C_p.

Pitch [deg]	**TSR [deg]**	**C$_\tau$[−]**
0,0	8,9107	0,0548

Algorithm 2 Pseudocódigo para el proceso de generación de la superficie C_p

→Inicializar el vector de pitch deseado $\beta = [\beta_{min}, \beta_{min} + \beta_{step}, \ldots, \beta_{max}]$.
→Inicializar el vector de TSR $\lambda = [\lambda_{min}, \lambda_{min} + \lambda_{step}, \ldots, \lambda_{max}]$.
→Fijar un valor de velocidad de viento constante v_w
→Calcular las velocidades de rotor para obtener los TSR deseados $\omega[i] = \frac{v_w \lambda[i]}{R}$
for b en β **do**
  for r_s en ω **do**
    →Generar información para una simulación con la tupla b, r_s
  end for
end for
→Exportar las simulaciones a varios ficheros para ejecutarlas de forma paralela. Por ejemplo 10000 simulaciones por fichero.
→Lanzar tantos procesos de Aerodyn/OpenFAST como sea necesario.
→Realizar el postproceso generando una matriz en un fichero ".mat" para su uso en el modelo.

4.2. Obtención del modelo de viento

Como se comentó en la sección 2.5.1, en este trabajo en vez de alimentar un rotor completo con un bloque de viento se creará un modelo de viento equivalente que tenga un comportamiento realista. Para ello se tomaba el viento como la suma de un valor estable más el producto de un ruido blanco que pasa a través de un filtro de segundo orden (2.14). Este filtro tendrá que tener un comportamiento frecuencial similar al espectro de frecuencia de la turbulencia del viento definido en (2.22). Para ello, se toma la función de transferencia del filtro de segundo orden (4.4).

$$H(s) = \frac{K_v}{s^2 + a_1 s + a_0} \tag{4.4}$$

De la que el espectro de densidad de potencia se obtiene como (4.5).

$$S_H(f) = H(j2\pi f)H^*(j2\pi f) \tag{4.5}$$

Se puede definir una función de coste para minimizar la diferencia de respuesta en frecuencia entre $S_H(f)$ y $S_e(f)$, previamente definida en (2.22). Esta función de coste tendrá como objetivo ajustar la ganancia del filtro para que sea lo más similar posible a la del espectro efectivo del viento. Para ello tendrá que acomodar los parámetros a_0, a_1, y K_v con el fin de minimizar la distancia en todo el conjunto de rango de frecuencias (4.6).

$$J = \int_{f_1}^{f_2} \left[\log\left(\frac{S_H(f)}{S_e(0)}\right) - \log\left(\frac{S_e(f)}{S_e(0)}\right) \right] \tag{4.6}$$

Donde J es el valor de la función en el rango a minimizar, y f_1 y f_2 es el rango de frecuencia que se exploran. Entonces, aplicando una función de la librería *scipy* de python con el método de Nelder-Mead simplex (Nelder and Mead [1965]) programado, se puede obtener

estos parámetros para una velocidad de viento media v_m y una intensidad de turbulencia I determinadas. En la figura 4.4 se puede validar el ajuste para una velocidad de viento media de $10m/s$.

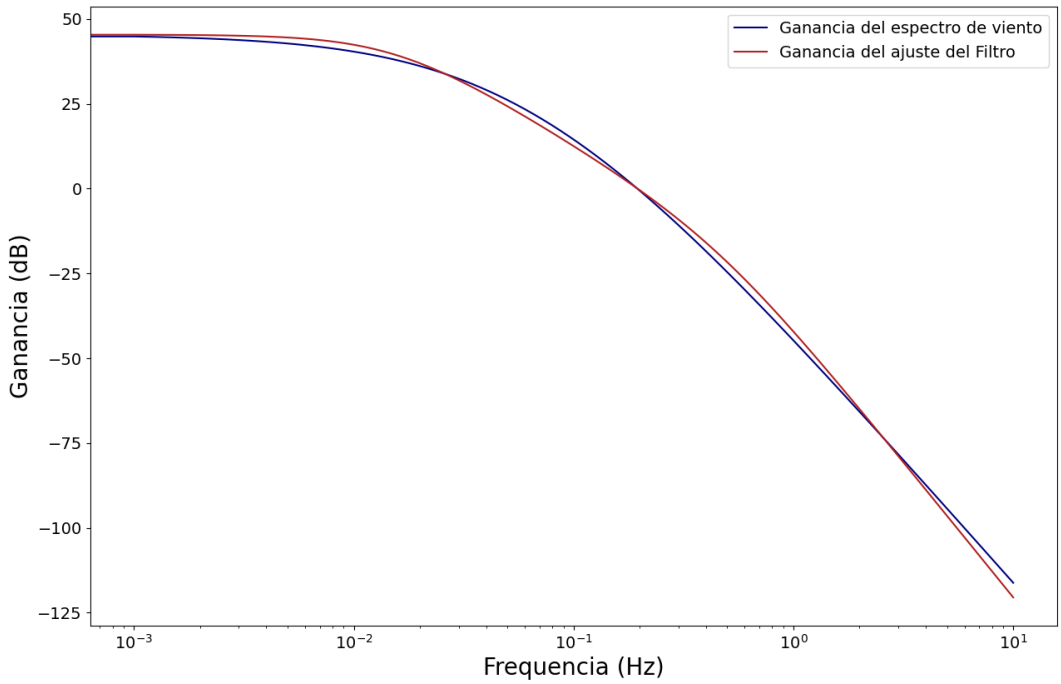

Figura 4.4: Ajuste de la ganancia del filtro (línea roja) que representa la turbulencia respecto a la modelizada del viento (línea azul).

Cabe destacar que este proceso sería válido para una velocidad de viento concreta. Esto hace que el trabajo de ajuste tenga que realizarse en diversos puntos de viento para poder cubrir todo el rango de funcionamiento de la máquina. Los valores obtenidos para las ganancias del filtro puede interpolarse entre unos y otros, o ajustarse por un polinomio. No obstante, en este trabajo se toman velocidades de viento que ya han sido previamente mapeadas, con lo que este paso no sería estrictamente necesario, ya que la interpolación se realizará en un punto existente. En el modelo que se encuentra dentro de Simulink, está programado como una interpolación para una inicialización más cómoda. Las variables que se han tenido en cuenta a la hora de realizar este proceso se listan en la tabla 4.3

Con estos valores, se han obtenido las ganancias para los parámetros del filtro que se muestran a continuación en la figura 4.5. En ella se incluyen los valores que se obtendrían interpolando entre los distintos puntos, y también los valores que se obtendrían de ajustar este según una función polinomial. El código necesario para llevar a cabo esta operación puede consultarse en el anexo D.

Tabla 4.3: Variables de entrada para cálculo de la turbulencia.

Variable	**Valor**
$v_m[m/s]$	$\in [3, 25]$ con paso 1
$I[\%]$	13,55
$f[Hz]$	$\in [10^{-3}, 10^{1}]$ con 100 puntos espaciados logarítmicamente
$h[m]$	150
$h_i[m]$	1000
$R[m]$	120,97

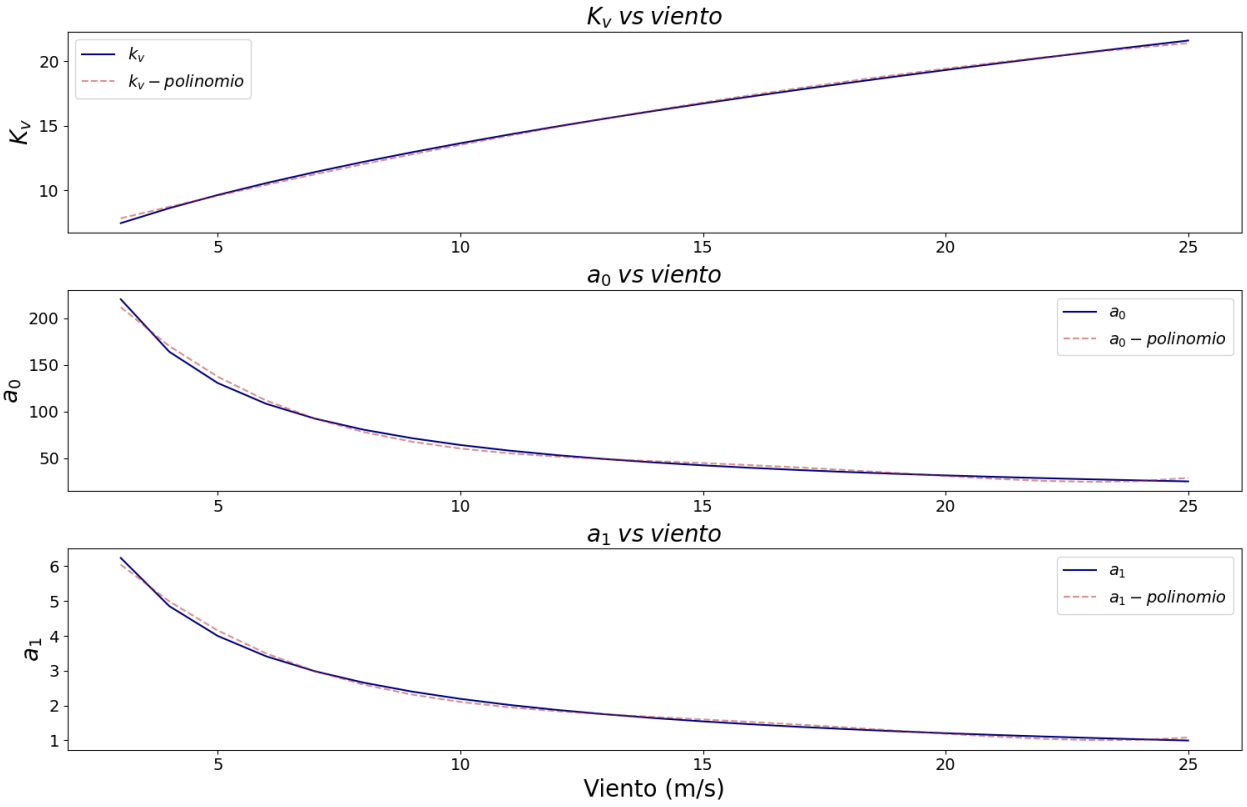

Figura 4.5: Ganancia de los filtros ajustada para todo el rango de vientos.

Capítulo 5

Diseño y aplicación del modelo de aprendizaje por refuerzo DDPG

En este capítulo se abordarán el diseño y la implementación del modelo de controlador de aprendizaje por refuerzo DDPG aplicado. El controlador, como ya se dijo, se implementa en Python sobre la librería de Tensorflow (Abadi et al. [2015]) con Keras (Chollet [2015]) como interfaz. Tensorflow es una potente librería optimizada para modelos de machine learning que cuenta con las funcionalidades de creación y gestión de redes neuronales para su funcionamiento y entrenamiento. Para trabajar con ella de una forma simplificada se cuenta con Keras como interfaz de las operaciones. Los modelos DDPG, cuentan con la desventaja de que son muy sensibles a la estructura de redes propuesta y al ajuste de hiperparámetros realizado, por lo que se hicieron numerosas pruebas hasta encontrar una configuración funcional que permitió llevar a cabo el aprendizaje de forma satisfactoria. En este capítulo se ofrece una presentación de la estructura de diseño del modelo de controlador, así como una explicación de las decisiones tomadas respecto al valor de los hiperparámetros y el diseño de recompensas, que será una de las claves para su correcto funcionamiento.

En la figura 5.1 se puede ver la particularización para el problema de la estructura de controlador DDPG mostrada en la imagen 3.3. En ella el Actor enviará una señal al modelo de entorno con el fin de realizar la regulación. Este modelo que se correspondería con el aerogenerador y el viento tratados en el capítulo anterior 4, devolverá entonces una serie de lecturas dependiendo de la perturbación de viento y de la acción que se haya realizado sobre él. De esta serie de medidas, se observan parte para realimentar de nuevo al controlador DDPG para que las redes neuronales de su Actor puedan tomar decisiones en base a su aprendizaje, creando así nuevas acciones que le permitan maximizar la recompensa obtenida.

Como acción de entrada, se ha seleccionado el ratio de cambio del par eléctrico $\dot{\tau}_E$. Entonces las acciones estarán orientadas a hacer un control del punto de MPPT del aerogenerador mediante la velocidad de par eléctrico, lo que podría verse de forma intuitiva como la regulación de velocidad que lleva a cabo el conductor de un automóvil con su pedal del acelerador.

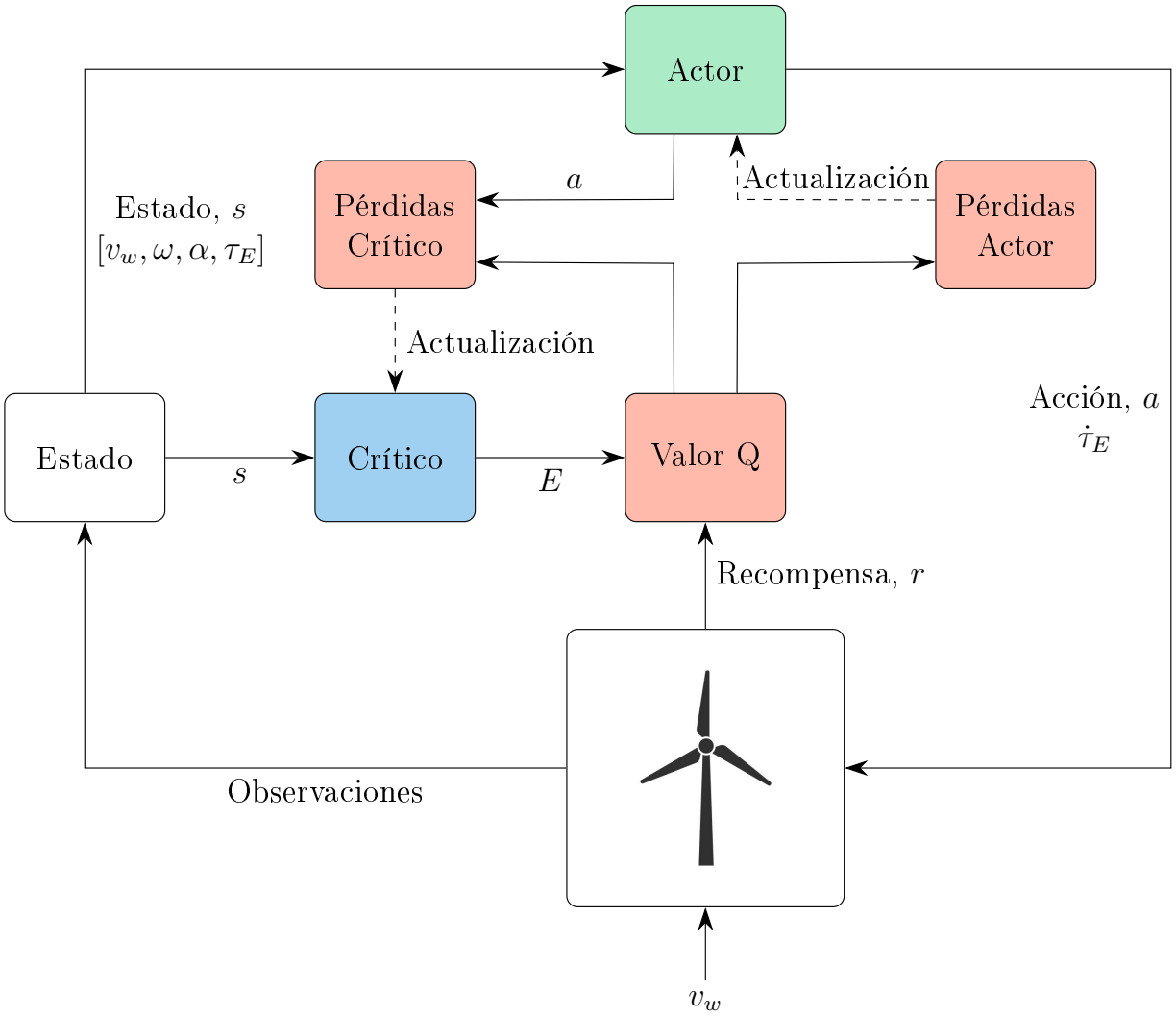

Figura 5.1: Esquema de funcionamiento del controlador DDPG.

La razón que subyace a esta elección de variable de control viene dada por los datos que proporciona el equipo creador de la turbina, en la que especifícan un valor máximo de variación de par electromagnético. En caso de intentar realizar la regulación en términos de par directamente, habría que limitar su derivada una vez ha dado la salida la red neuronal, con lo que se ha estimado que tomar directamente ésta es la forma más sencilla de cumplir la limitación. Adicionalmente en la política del Actuador, se implementarán métodos de saturación para que cuando el par sea máximo o mínimo, su derivada no pueda seguir creciendo en el sentido de la limitación.

Respecto a las variables de estado del entorno, como puede verse en la imagen 5.1 se han elegido la velocidad del viento v_m, la velocidad de rotación del rotor ω, la aceleración del rotor α, y el estado actual del par electromagnético τ_E. Parece contraintuitivo no contar con la potencia dentro de estas variables, y al principio se incluyó. No obstante esta aproximación daba peores resultados, y se concluye que probablemente se deba al efecto de que esta variable presenta una dependencia de forma directa con la velocidad y el par, por lo que al ser una variable dependiente hacía que el sistema abstrayera el problema de una forma más pobre.

5.1. Estructura de las redes neuronales

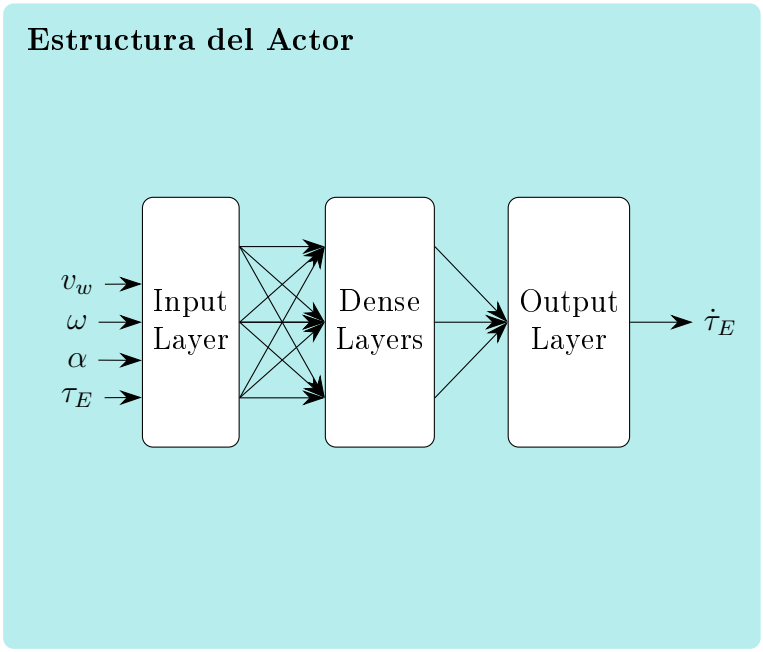

Figura 5.2: Estructura de la red neuronal del Actor.

En la figura 5.2, se puede ver la estructura que se ha tomado para obtener las redes objetivo y modelo del Actor. Estas redes se componen de una capa de entrada que recibe el valor de los estados que se consideran descriptivos del sistema, y que a continuación pasan por una serie de capas densas de neuronas para ofrecer una salida. Esta salida sería la derivada

del par eléctrico $\dot{\tau}_E$, como se mencionó al principio de este capítulo.

Un aspecto importante a la hora de mejorar los entrenamientos consiste en la normalización de las entradas ya que entradas muy dispares en magnitud, como es el caso, inducen a un gran cambio en la covarianza que existe dentro de las neuronas de la red (Ioffe and Szegedy [2015]). Para normalizar entonces estas entradas, se toma el rango de operación esperado, y se normaliza éste de forma que su salida se encuentre entre -1 y 1 de forma normal, pudiendo ser que en algún momento se exceda levemente uno de los valores pero estando igualmente todas las variables en un orden de escala similar, lo que acelera o directamente permite la convergencia del aprendizaje.

Otra de las cuestiones que pueden surgir es el número de neuronas a emplear. Este valor depende en gran medida del problema, y para llevarlo a término se tomaron las directrices expuestas en (Nguyen et al. [2021]). La profundidad de capas en una red neuronal, mejoran la capacidad de abstacción de los modelos, permitiendo captar relaciones más complejas. Mientras redes más anchas permiten una mejor paralelización, pudiendo capturar de forma simultánea los efectos de diferentes entradas. No obstante, el valor a tomar ha de estar ajustado ya que en caso de hacer las redes demasiado pequeñas, tanto en ancho como en largo, penalizará o hará imposible el aprendizaje. Por contra, hacerlas demasiado profundas puede acarrear problemas como incrementar de forma significativa los costes computacionales, o hacer que aparezcan problemas de desvanecimiento de gradientes o overfit. Hacerlas demasiado anchas puede acarrear también problemas de overfit y de rendimiento computacional. Con estos preceptos se ha optado por una estructura para el Actor que se muestra en la tabla 5.1, y que se corresponde con el detalle de la figura 5.2.

Tabla 5.1: Capas de las redes neuronales del Actor.

Capa	**Neuronas**	**Capa de entrada**	**Activación**
Input	4	-	-
Dense_1	256	Input	ReLu
Dense_2	512	Dense_1	ReLu
Dense_3	512	Dense_2	ReLu
Output	1	Dense_3	Tanh

Respecto al Crítico, el modelo guarda similitudes con el Actor. La estructura de éste puede verse en la figura 5.3, y en ella se aprecian dos entradas diferenciadas, una que se compone de las entradas provenientes del estado del entorno, y otra que se correspondería con la salida que ha ofrecido el Actor con el fin de poder evaluar el efecto de ésta. Más adelante junta estas informaciones con una capa de concatenación, y es capaz entonces de ofrecer su estimación del valor de la ecuación de Bellman (3.4) después de pasar por otra serie de capas densas.

Los criterios para establecer el número de capas, y la estructura en general de la misma son los mismos que se han expuesto hasta ahora, y pueden verse en la tabla 5.2.

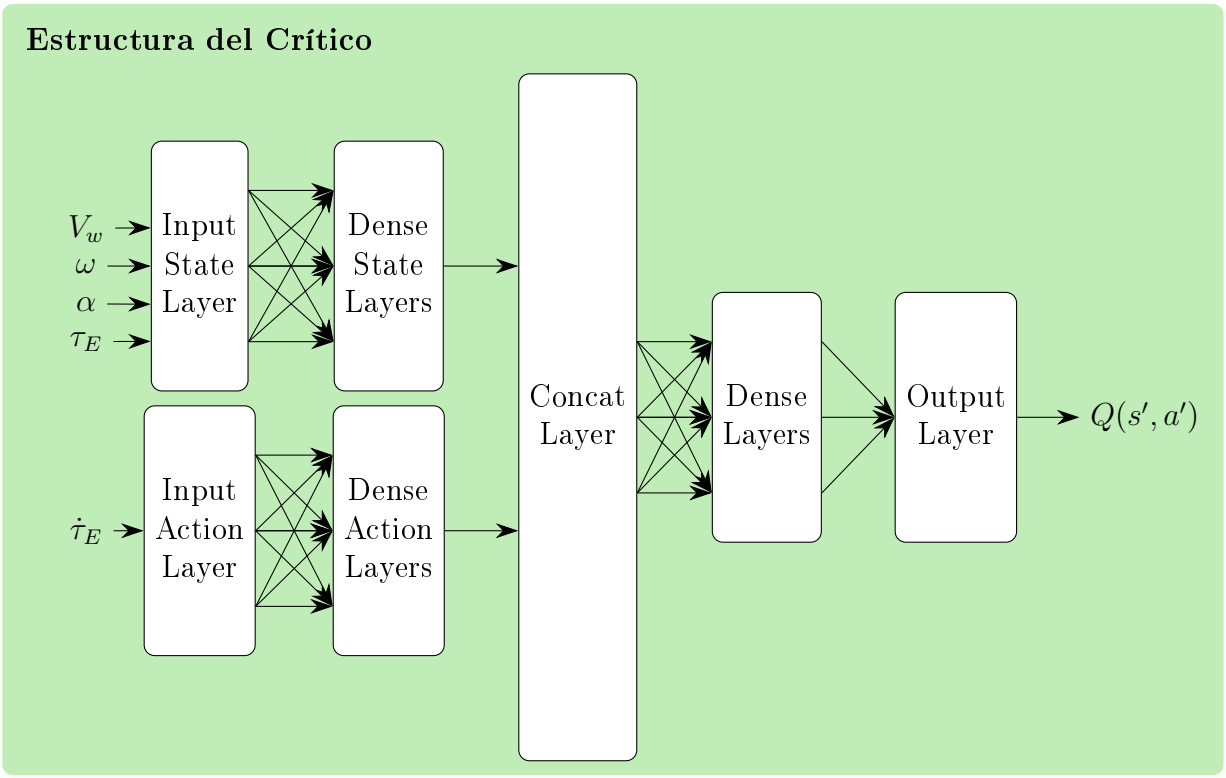

Figura 5.3: Estructura de las redes neuronales del Crítico.

Tabla 5.2: Capas de la red neuronal del Actor.

Capa	**Neuronas**	**Capa de entrada**	**Activación**
State_Input	4	-	-
Dense_State_1	256	State_Input	ReLu
Dense_State_2	256	Dense_State_1	ReLu
Action_Input	1	-	-
Dense_Action	64	Action_Input	ReLu
Concat	-	Dense_State_2 & Dense_Action	-
Dense_1	512	Concat	ReLu
Dense_2	512	Dense_1	ReLu
Dense_2	512	Dense_3	ReLu
Output	1	Dense_3	ReLu

Puede verse que la activación de la salida de la capa del Actor, a diferencia de la del Crítico, se produce en forma de Tanh. La salida en función de tangente hiperbólica es de uso común en el aprendizaje por refuerzo cuando se trabaja con espacios contínuos, y ofrece unos valores en el rango de $[-1, 1]$. Estos valores tendrán por lo tanto que ser escalados convenientemente dentro del rango de actuaciones para poder ser aplicados al modelo. En este caso, dado que la entrada al modelo es el ratio de cambio del par eléctrico, éste se obtendrá escalando la salida de la red neuronal del Actor por los límites de este valor, que como son los mismos tanto en sentido negativo como positivo, se podrá simplificar según la acción (5.1).

$$\dot{\tau}_E = \dot{\tau}_{Actor}|\dot{\tau}_{max}| \tag{5.1}$$

5.2. Diseño de las recompensas

Como ya se mencionó, un diseño óptimo de las recompensas ha de ser cuidadoso para que el sistema funcione de forma adecuada. En caso de no hacer esto de forma correcta, los efectos pueden ser que el modelo sea incapaz de aprender, o incluso que lo aprendido se corresponda con un comportamiento no deseado (Guo [2017]).

Las recompensas en este trabajo están orientadas a obtener un comportamiento determinado, que es seguir la curva de MPPT de un aerogenerador de la forma más óptima posible. No obstante, también se tienen en cuenta efectos colaterales que provocarían que la máquina pudiera funcionar peor, o que su operación fuera más costosa o insegura. Por ello, se adopta un enfoque multiobjetivo por el cual la recompensa total recibida es una suma de recompensas ponderadas por factores individuales según la importancia que revisten dentro del comportamiento general. Se adopta además un esquema por el que las recompensas individuales tendrán un valor 0 en su punto máximo, y estarán normalizadas para poder ser comparables entre sí. Con este enfoque, en el punto ideal de funcionamiento el valor de recompensa será 0, mientras que cuando se va alejando del mismo éste se irá haciendo cada vez más negativo (5.2).

$$R = -\sum_{i}^{n} f_i r_i \tag{5.2}$$

El número total de recompensas que se han añadido es de 7, teniendo cada una de ellas su función que se explicará a continuación. Las primeras estarán asociadas a la maximización de la potencia en forma de seguimiento del TSR y realizando un ajuste de potencia; la tercera estará destinada a evitar sobrevelocidades; la cuarta a proporcionar herramientas extra al controlador para evitar sobrevelocidades y sobreaceleraciones; la quinta irá destinada a penalizar las actuaciones que provoquen una sobreaceleración del rotor; la sexta estará orientada a minimizar el uso del actuador con el objetivo de que las actuaciones sean preferiblemente

suaves; y la séptima y última, estará dedicada a castigar fuertemente estados indeseables o peligrosos que pueden aparecer durante la operación del aerogenerador.

Como se ha visto, se puede asumir que en cualquier aerogenerador el punto óptimo de funcionamiento se va a encontrar en la velocidad de rotor para la cual el TSR es el que proporciona un C_p máximo. Este valor óptimo es conocido y se ha hallado anteriormente en este trabajo y puede consultarse en la tabla 4.2. El controlador deberá ser capaz de aprender el par ratio de par eléctrico que ha de aplicar en cada momento para mantener este valor lo más próximo posible al deseado (5.3). El valor de tendencia de recompensa se puede comprobar en la figura 5.4 para un rango de TSR, y en esta puede verse que el valor mínimo, que más tarde será negado y pasará a ser el máximo, se encuentra en el punto de TSR óptimo.

$$r_\lambda = \left| 1 - \frac{\frac{\omega R}{v_w}}{\lambda_{opt}} \right| \tag{5.3}$$

Figura 5.4: Recompensa asociada al TSR.

En grandes aerogeneradores, con palas largas y flexibles, puede ser que este ajuste sea ligeramente mejorable también debido a tolerancias de montaje, en las que el TSR máximo no genere toda la potencia teórica. Para ello, se introduce un factor de ajuste que serviría para afrontar estos efectos y que introduce la potencia como la que idealmente debiera generarse en ese punto de operación (5.4). Este factor debe verse como una ayuda a un ajuste fino en

combinación con el ajuste grueso que se lleva a cabo con el TSR, por lo tanto, ha de tener un factor reducido respecto a este. En la figura 5.5 se pueden ver las recompensas para este en función de la potencia para un viento de $9m/s$, al que le correspondería un valor de aproximadamente $10{,}3 \times 10^6[W]$. Como puede verse, en el punto que corresponde a esa potencia el valor de la función será el mínimo, y se irá acrecentando a medida que la potencia extraída se aleje más. Cabría destacar también que el punto óptimo de potencia irá cambiando según el viento que exista en ese momento, tal y como se extrae de la ecuación (5.4).

$$r_p = \left(1 - \frac{P_E}{\frac{1}{2}\rho\pi R^2 {v_w}^3 C_{p_{opt}}\eta}\right)^2 \tag{5.4}$$

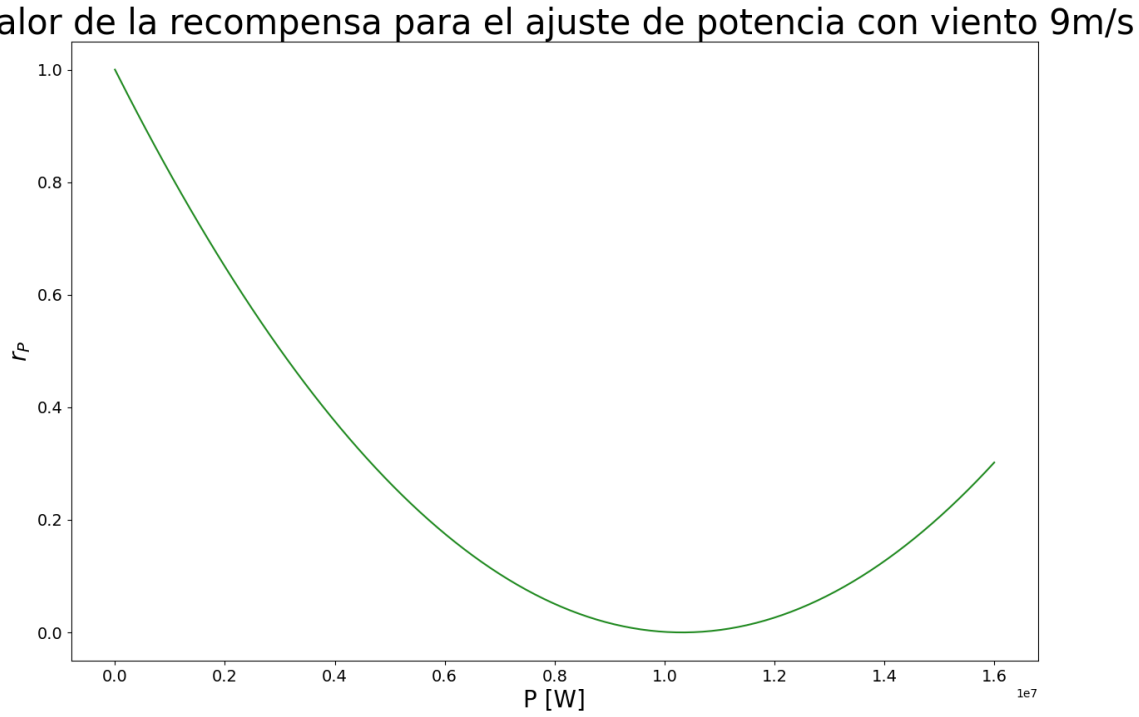

Figura 5.5: Recompensa asociada al ajuste de potencia.

Respecto a la velocidad de rotor, la recompensa se diseña con el fin de castigar las sobrevelocidades que puedan producirse por ser este un comportamiento subóptimo desde el punto de vista del trabajo del generador, ya que podría generar fuerzas electromotrices que creen un gran número de potencia reactiva, aumentando de esta forma su temperatura y mermando su eficiencia. Además, desde el punto de vista del comportamiento puede inducir también a comportamientos peligrosos o dañinos. Por ello se introduce el término (5.5). Los valores de recompensa para esta función pueden verse en la figura 5.6, en la que como se observa el trabajo en zonas por debajo de nominal se corresponderían con el máximo de la función que, a medida que crece en sobrevelocidad, irán teniendo una penalización mayor.

$$r_\omega = \begin{cases} \left(1 - \frac{\omega}{\omega_{nominal}}\right)^2 & \text{si } \omega > \omega_{nominal} \\ 0 & \text{en otro caso} \end{cases} \tag{5.5}$$

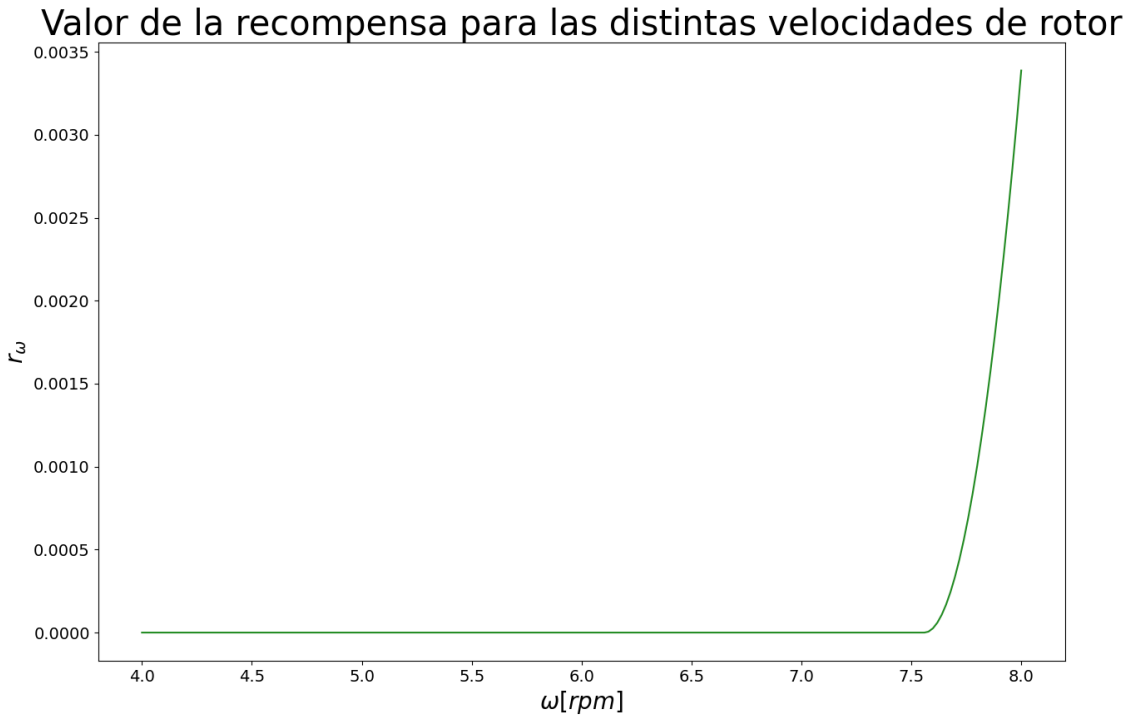

Figura 5.6: Recompensa asociada al valor de velocidad del rotor.

Se permite una pequeña cantidad de par según los rangos de actuación para ayudar a alcanzar ciertos estados de forma más rápida (5.6). La adición de este término ayuda al controlador a poder controlar de forma más efectiva sobrevelocidades repentinas causadas por ráfagas. Para el cálculo de su factor hay que tener en cuenta que éste no puede ser considerado un factor de operación común. La gráfica de la evolución de este valor se puede ver en la figura 5.7. En la imagen se ve que a medida que el par crece por encima de nominal, este crea un castigo por ser utilizado. Si bien es cierto que la forma comparte filosofía y similitud con la de sobrevelocidad 5.6, habrá que ponderarlas según su grado de deseabilidad, permitiendo que el uso excesivo de par sea una herramienta puntual con un menor castigo asociado.

$$r_{\tau_E} = \begin{cases} \left(1 - \frac{\tau_E}{\tau_{E_{rated}}}\right)^2 & \text{for } \tau_E > \tau_{E_{rated}} \\ 0 & \text{en otro caso} \end{cases} \tag{5.6}$$

La actuación de par se castiga ya que es deseable mantener esta de forma lo más constante posible con el fin de obtener una actuación suave. No obstante, ya que es un requisito

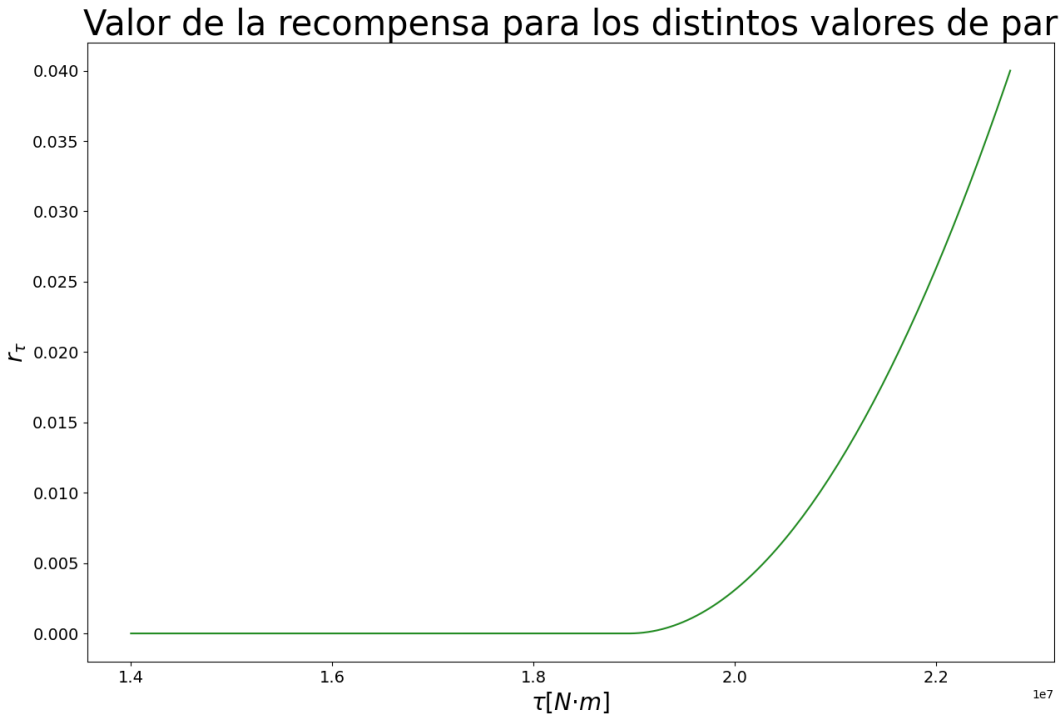

Figura 5.7: Recompensa asociada al valor de par del generador.

imprescindible para alcanzar el máximo valor posible de potencia, se prefiere una actuación lo más moderada posible (5.7). La actuación está limitada por un valor de $4{,}5 \times 10^6[N \cdot m]$, y la evolución de esta recompensa puede encontrarse en la figura 5.8, en la que como puede verse, a mayor uso del actuador mayor castigo.

$$r_{\dot{\tau}_E} = \left(\frac{\dot{\tau}_E}{\dot{\tau}_{E_{\text{máx}}}} \right)^2 \tag{5.7}$$

Altas aceleraciones en el rotor son un síntoma de mal funcionamiento ya que introducen en el sistema cargas de fatiga que acortan la vida de los materiales. Además, pueden llevar a estados peligrosos y causar inestabilidad. Para afrontar esto se proporciona un castigo al sistema en caso de alcanzar aceleraciones por encima de un umbral permitido (5.8). Dado que este valor para la turbina no está definido por parte del equipo de trabajo que ha sido responsable de su diseño, se toma un valor de $0{,}5 rpm/s^2$. La evolución de este valor puede verse en la figura 5.9, en la que como se observa este sería 0 para valores inferiores al propuesto como límite, a partir del cual la función tiene un escalón y pasa a crecer de forma cuadrática.

$$r_\alpha = \begin{cases} \left(\frac{\alpha}{\alpha_{\text{máx}}} \right)^2 & \text{si } |\alpha| > \alpha_{\text{máx}} \\ 0 & \text{en otro caso} \end{cases} \tag{5.8}$$

Figura 5.8: Recompensa asociada al valor de ratio de cambio del par electromagnético.

Por último, a pesar de las recompensas previamente expuestas, durante las simulaciones se pueden presentar situaciones no deseables en las que el aerogenerador se puede situar en estados peligrosos o subóptimos de funcionamiento tales como sobrepotencia, sobrevelocidad, infrapotencia, e infravelocidad. Para prevenir esto, se proporcionan una serie de castigos cada vez que el controlador llega a uno de estos estados. Cada uno de ellos tiene una recompensa negativa asociada, y los peores, que serían los coligados con estados peligrosos como sobrepotencia y sobrevelocidad (no confundir con un funcionamiento por encima de nominal, ésta se correspondería con una velocidad de corte de operación), tienen un efecto más punitivo.

$$r_{indeseable} = \begin{cases} 30 & \uparrow\uparrow\uparrow \text{ para estados de peligro} \\ 20 & \uparrow\uparrow \text{ para estados de parada} \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

Como los estados no deseables, al igual que la aceleración máxima permitida, no están definidos para la turbina de referencia, se elige un grupo de valores razonables como límites de $\omega \in [4, 8]rpm$, $P_E \in [0, 18]MW$.

Entonces, para obtener el valor de la recompensa total, habría que éscalar estas por su factor. Este puede extraerse de las gráficas escalándolas en base al punto de funcionamiento y a la variabilidad que presentan, y multiplicándolas luego por un factor de importancia

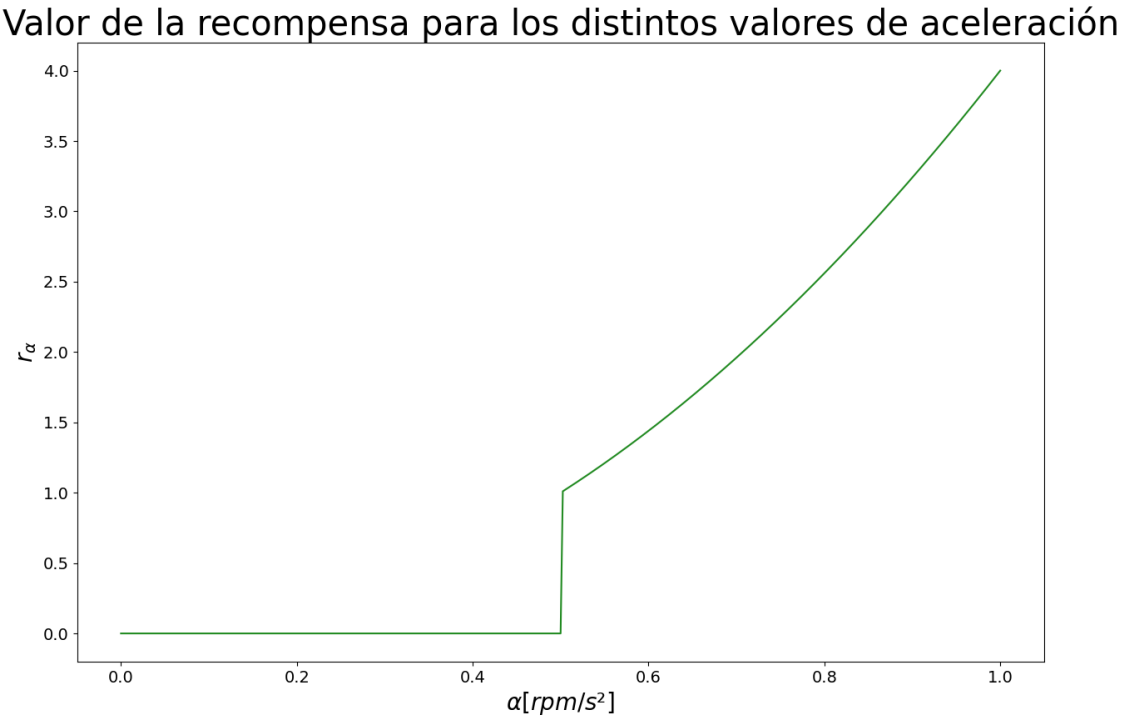

Figura 5.9: Recompensa asociada al valor de aceleración del rotor.

porcentual que presentan para la misma. Por ejemplo, el par máximo en su punto máximo de actuación tiene un valor aproximado de 0,04 ya que presenta poca variabilidad respecto a su punto nominal; si se le quiere dar a esta variable una importancia de 20 % sobre el total de la actuación, su factor sería $\frac{0,2}{0,04}$. La relación de los distintos factores asociados a las recompensas f_i que habría que aplicar en la ecuación (5.2) se muestra a continuación enla tabla 5.3.

Tabla 5.3: Factores de ponderamiento de las recompensas según la ecuación de recompensa total (5.2).

Factor	**Recompensa asociada**	**Valor**
f_λ	r_λ	1/0,3
f_p	r_p	0,5
f_ω	r_ω	1/0,003387
f_{τ_E}	r_{τ_E}	0,2/0,04
$f_{\dot{\tau}_E}$	$r_{\dot{\tau}_E}$	0,01
f_α	r_α	0,15
$f_{indeseable}$	$r_{indeseable}$	1

Como apunte final, en los modelos se emplean números de coma flotante y precisión simple de 32 bits. Con el fin de no perder resolución en los decimales a la hora de hacer el

entrenamiento, la recompensa total se escala por un factor de 1000.

5.3. Hiperparámetros de entrenamiento

Los modelos basados en DDPG son muy sensibles a la elección de los hiperparámetros de entrenamiento. Durante el transcurso de estos se han probado numerosas combinaciones. No obstante es probable que para obtener una configuración realmente óptima haya que hacer una búsqueda más extensiva y, probablemente, emplear un sintonizador automático para ir probando diferentes soluciones e ir convergiendo hacia la óptima. Los hiperparámetros que se han empleado han sido los siguientes:

- τ : parámetro de actualización suave para los pares de valores de redes neuronales de Crítico y Actor (3.8). Para este parámetro se han probado valores entre 1×10^{-3} y 5×10^{-3}, logrando convergencia con ambos.

- γ : Factor de descuento (3.2). Se han probado valores en el rango de $[0{,}900; 0{,}995]$. Los mejores comportamientos se han conseguido con 0,900, y eso tiene su lógica ya que al ser un sistema que depende de una perturbación estocástica externa, es difícil predecir a largo plazo para el Crítico las consecuencias de las acciones, ya que no estarán completamente influenciadas por él. Por ejemplo, con este factor, las recompensas en un horizonte temporal de un segundo estarían ponderadas por un valor de aproximadamente 0,35 respecto a las presentes, mientras que si el factor fuera por ejemplo 0,99, estas estarían ponderadas por un factor de 0,90.

- **Tamaño de minibatch:** durante el proceso de entrenamiento, el tamaño elegido de los minibatches que se toman del buffer se ha establecido como 64, 128, 256, y 512. Dado que con 128 se obtiene una respuesta convergente, se optó por este valor para tener una mayor agilidad computacional. Como nota adicional, cabe destacar que todos los números elegidos son potencia de 2 (así como los presentes en el número de las neuronas de la red). Esto se debe a que al tener la posibilidad de ejecutar el entrenamiento en la GPU en vez de en la CPU, estando estos dispositivos especialmente preparados para trabajar con un número de potencia que depende de 2 en operaciones matriciales. Además la asignación en memoria sería también más eficiente. Si bien cabe destacar que estos valores podrían tomar cualquier número entero razonable.

- **Tamaño de buffer:** Se probó con tamaños de buffer diferentes. Como se comentaba anteriormente, tamaños demasiado cortos pueden inducir a errores de aprendizaje por no conservar información suficiente sobre las distintas exploraciones que se han llevado a cabo en el entorno; mientras que tamaños excesivos, podrían ralentizar la convergencia de los modelos por evaluar en etapas avanzadas datos del inicio del entrenamiento. En este trabajo se ha elegido un tamaño de buffer de 1×10^6, que es un valor mayor

que otros que se han empleado pero con el que se han obtenido buenos resultados de convergencia.

- **Ratio de aprendizaje del Crítico:** es el cambio de gradiente que permite el optimizador que está minimizando la función de pérdidas del Crítico. Se han probado diversos valores siendo el más grande 2×10^{-4}. Durante el entrenamiento se puede empezar por ejemplo con este valor inicial, e ir reduciéndolo de forma manual o automática para asegurar una mejor convergencia. Como optimizador se ha utilizado Adam (Kingma and Ba [2017]), sustentado en el método de gradiente descendente estocástico basado en una estimación adaptativa de los momentos de primer y segundo orden.

- **Ratio de aprendizaje del Actor:** es el cambio de gradiente que permite el optimizador que está minimizando la función de pérdidas del Actor. De la misma forma que con el Crítico, se han probado diversos valores siendo el más grande 1×10^{-4}. Este valor además ha de cumplir la regla de ser menor o igual que el ratio del Crítico, ya que como ése es el que genera el gradiente de evaluación para las pérdidas del Actor, no tendrá sentido que el segundo intente aprender de forma más rápida que el primero. Por lo tanto, si se reduce el ratio de aprendizaje del Crítico durante la simulación, el ratio de aprendizaje del Actor ha de verse también reducido. Como optimizador se ha utilizado también Adam.

5.4. Proceso de entrenamiento y ruido de exploración

Como ya se ha mencionado en la sección 3.1, una de las claves para entrenar modelos de controladores basados en aprendizaje por refuerzo es la exploración. En este trabajo se han probado varias alternativas, desde un proceso Ornstein-Uhlenbeck sumado a la salida de las actuaciones del Actor con un factor de decaimiento ϵ (3.13) a usar la misma alternativa pero con una distribución gausiana. Se puede ver un ejemplo de varios procesos Ornstein-Uhlenbeck en la figura 5.10.

No obstante, el ya explicado ruido de proceso (3.14), ha conseguido superar a los anteriores en términos de convergencia y ajuste fino del espacio de los estados. El valor de este ruido puede ir ajustándose de forma manual o automática a lo largo de la simulación, de la misma forma que se hace con los anteriores mediante una distancia entre la acción propuesta por la red original y la alcanzada con el Actor auxiliar ruidoso.

Independientemente del ruido que se elija, las acciones han de saturarse por valor mínimo y máximo antes de transformarse a una acción que se le proporciona al modelo de la forma que se expone en (5.1), ya que si la acción se encontrara en el límite y el ruido fuera en esa dirección, el valor permitido podría verse sobrepasado. Es decir, el valor de la acción normalizada ha de estar entre $[-1, 1]$. Además, como en este caso se está trabajando con el ratio de cambio de par, y no con el par en sí mismo, habrá que saturar también este valor

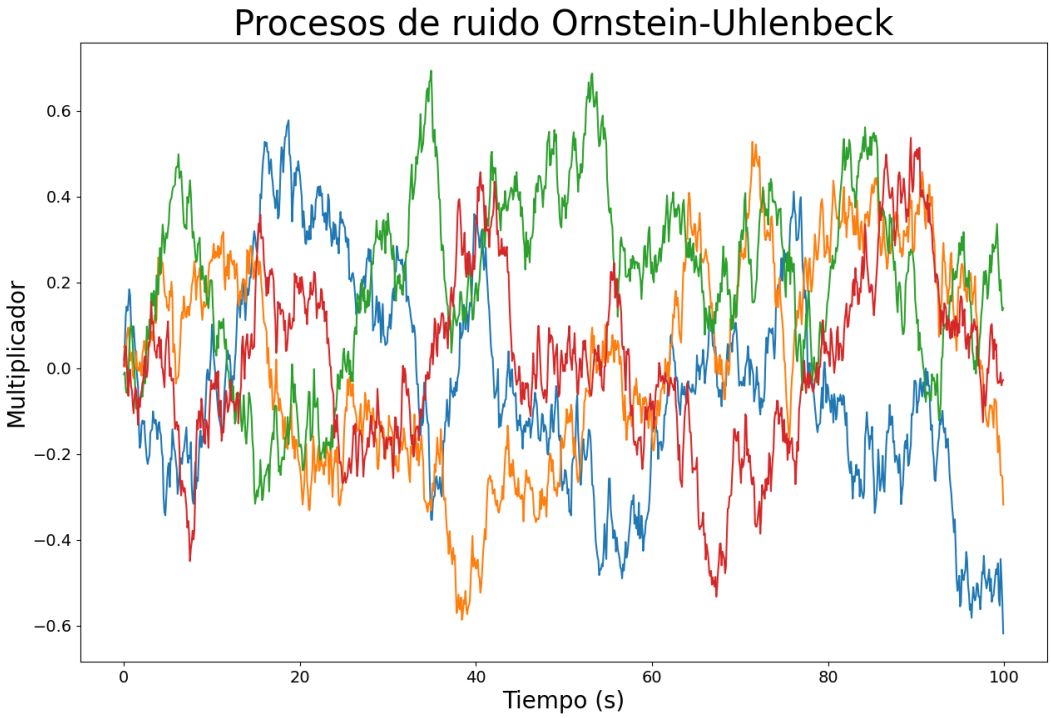

Figura 5.10: Ejemplos de ruido generado con procesos Ornstein-Uhlenbeck.

cuando se sitúe en uno de los límites de actuación. Es decir, si por ejemplo el par es igual al máximo permitido, la salida de la actuación de su ratio de cambio tendrá que estar limitada por $[-1, 0]$, no permitiendo a este valor crecer más.

Por último, cabe destacar la existencia de entornos de de código abierto disponibles como campo de pruebas para medir las capacidades de los distintos modelos. El más destacable es Gymnasium (Towers et al. [2024]), que ofrece distintos problemas de control para los distintos modelos de aprendizaje por refuerzo. Los entornos que se ofrecen en esta librería se han convertido en uno de los estándares académicos de banco de resultados para comparar las distintas capacidades de modelos e implementaciones. Si bien en este trabajo no se muestran los resultados por ser algo paralelo a los mismos, se han realizado pruebas para comprobar la validez, tanto de la implementación del modelo DDPG, como de ciertas mejoras, o los tipos de ruido, con el fin de contrastarlas contra un entorno estándar ya comprobado, pudiendo verificar así la validez o no de las propias implementaciones eliminando la incertidumbre de la creación de modelos propios.

5.5. Aplicación práctica y flujo de información

Una vez se ha explicado de forma conveniente la parte de los modelos de entorno y de controlador, en esta sección se profundiza un poco más en el flujo de trabajo y la aplicación

práctica de los distintos módulos y como interactúan entre sí para generar el objetivo final, que es un modelo DDPG entrenado. Los distintos módulos se proporcionan en anexos al final de este documento, y que pueden ser consultados si se quiere profundizar en alguno de los detalles que aquí se comentarán.

Lo primero será hallar los valores estáticos necesarios para la ejecución del modelo del entorno. Estos serán los valores de la superficie C_p, que se obtienen según según las instrucciones dadas en la sección 4.1.1; y los valores de parametrización de la turbulencia de viento, que se calcularon según lo expuesto en 4.2.

Retrotrayéndose entonces a la figura 4.1, puede verse que hay un módulo general que coordina todos los esfuerzos. Este módulo es el responsable de inicializar valores y de hacer las llamadas apropiadas a las distintas partes, y puede consultarse en el apéndice E. Sobre este módulo, además estará programada la obtención de los valores según la política actual del Actor.

Al principio se hace una llamada a las clases responsables de los servicios auxiliares, que se encargan de la creación o carga de los modelos, así como de hacer salvaguardas de los mismos; hay otra clase que se encarga de la gestión de los parámetros de aprendizaje y de la actualización de las redes neuronales; y otra que se encarga de la configuración de las simulaciones, estableciendo la duración y el paso de las mismas, así como el número de éstas. Este módulo de servicios auxiliares se puede consultar en el anexo F.

A continuación se inicializa el buffer. En éste se almacenará el número deseado de pasos de simulación para ir ejecutando el entrenamiento haciendo uso de ellos. Estos pasos de simulación, una vez se ha llegado al límite, adoptarán una estrategia FIFO para ir substituyendo los más antiguos por nuevos y permanecer entonces siempre lleno de valores. Además, cuando se inicializa el buffer, para hacer las operaciones más eficientes se inicializan los diferentes vectores en la librería *numpy* de python para tener una memoria asignada y no cambiante, con lo que se consigue que estos no tengan que crecer y ser sus valores asignados en cada iteración. El buffer, además, contará con la función de llevar a cabo el entrenamiento de las redes neuronales ya que éstas podrán de esa forma acceder a los valores en memoria que este tiene sin intermediarios, lo que hará la operación más rápida. Este método se programa con el decorador "@tf.function" que abstrae del flujo normal de lógica del programa, permitiendo que Tensorflow construya un grafo estático que mejora la velocidad de esta ejecución de forma notable. El código para el buffer puede encontrarse en el apéndice G.

El siguiente paso sería obtener los valores de configuración del aerogenerador. Para ello se invoca el módulo correspondiente, que es capaz de generar un diccionario de valores que se le pasarán a continuación al modelo de entorno creado en Simulink con el fin de que emule a la turbina con los parámetros adecuados. Este módulo se encargará también, durante el inicio de las simulaciones, de calcular el valor inicial de velocidad y par que le correspondería según el viento para poder inicializar los integradores de forma suave, pudiendo de esa manera aprovechar el total de la simulación sin tener que esperar hasta un punto de convergencia.

Este módulo puede verse en el anexo H.

Seguidamente se procede a inicializar la interfaz con Simulink. Este módulo, se encargará de inicializar una sesión de Matlab y, dentro de ella, abrir el modelo de Simulink y configurarlo con los datos del aerogenerador. Se encargará también de establecer los valores del viento de inicio, obteniendo del módulo principal un valor de viento inicial que será uno de los valores $7m/s$, $8m/s$, ó $9m/s$, obtenido de forma aleatoria; y una semilla, también aleatoria, para inicializar el proceso de ruido blanco y representar una turbulencia distinta en cada una de las simulaciones. Otra de las funciones que ejerce este módulo es la de proporcionar el valor de entrada $\dot{\tau}_E$ que proviene del cálculo del controlador, aplicar un paso de simulación, y recopilar la respuesta que ha dado el modelo. Con esta respuesta se realiza una normalización de los valores del estado $[v_m, \omega, \alpha, \tau_E]$, que se almacenan en el buffer para proveer de datos de entrenamiento a la red neuronal. Con este retorno se calcula también en esta clase el valor de las recompensas generadas, y se almacenan conjuntamente con los anteriores datos y el de la acción en el buffer. La última función que cumpliría este módulo sería la de cerrar la simulación y la sesión de forma segura. Cabe destacar que la simulación y el modelo se abren una sóla vez en cada proceso de entrenamiento, ya que hacerlo cada vez que se va a ejecutar una simulación sería altamente costoso en términos temporales. Por lo tanto, una vez el módulo recibe la orden de fin de simulación, éste la reiniciará al tiempo inicial y establecerá unos nuevos valores de viento. Se puede encontrar su código en el apéndice I.

Por último, antes de entrar en el proceso de entrenamiento, se invoca a un módulo auxiliar de logging. Este módulo va almacenando los datos de cada simulación y volcándolos a un par de ficheros, un csv con los valores de las distintas series temporales, y otro que contiene el sumario de la simulación, como los valores medios, el viento y la semilla, etc. Si bien este módulo no sería imprescindible desde el punto de vista del proceso, es muy últil desde el punto de vista del debug y la consulta de elementos previos. Además, cuenta con un visor de simulaciones con capacidad de sacar gráficas para las simulaciones, de forma que se puede ir verificando el proceso de entrenamiento de una forma interactiva que permite tomar decisiones, como cambiar manualmente el nivel de ruido o los ratios de aprendizaje. El código puede consultarse en el anexo J.

Una vez realizada esta operativa inicial, el modelo inicia su proceso de entrenamiento. Para ello cuenta con un número de episodios asignados que correrán hasta que estos se agoten o el usuario lo interrumpa. Al iniciar cada uno de los episodios, se inicializa un viento y una semilla de forma aleatoria y, con este valor de viento, se obtiene el valor de parámetros iniciales de los integradores del modelo al que se le aplica una pequeña variación para no repetir excesivamente puntos de entrenamiento. Entonces, se entra en el bucle propio de la simulación. En este bucle ejecutará con el δt elegido (en este caso $0{,}1s$) las acciones que considera la red del Actor con el ruido elegido, y guardando las respuestas a éstas dentro del buffer. En caso de incurrir en un evento que implicaría un comportamiento peligroso o indeseable, como los que se han descrito en la ecuación (5.9), la simulación se truncará y

se inicializará otra nueva; en caso contrario, ésta correrá hasta agotar el tiempo que le ha sido asignado. Para dotar de variabilidad a los vientos, se ha decidido tomar simulaciones de $100s$ durante el entrenamiento. Al final de cada simulación se sacarán los datos relevantes por pantalla, y esta se almacenará para su posterior consulta en caso de ser necesario. Además, mientras se está simulando, se van guardando backups del modelo por si fuera de interés volver atrás en algún punto, y existe la posibilidad de pausar el entrenamiento y realizar acciones como, por ejemplo, cambios de ratio de aprendizaje o de nivel de ruido. Finalmente, se cierra de forma adecuada la interfaz con Simulink, y se vuelcan los modelos finales a un directorio deseado.

El flujo de trabajo que ha sido expuesto en esta sección se muestra a continuación en forma de pseudocódigo para una mejor comprensión en el Algoritmo 3.

Algorithm 3 Flujo de trabajo del entrenamiento del modelo de controlador DDPG

→Inicialización de los servicios auxiliares
→Creación desde las redes neuronales del Crítico y el Actor, o carga desde un modelo previo
→Establecimiento de los parámetros de aprendizaje γ, τ, $ratio_{Actor}$, $ratio_{Critico}$
→Establecimiento de los parámetros de simulación $episodios$, t_{inicio}, t_{fin}, δ_t
→Inicialización el buffer de experiencias previas R
→Obtención de los valores de configuración del aerogenerador
→Inicialización de la interfaz con Matlab
→Inicialización del modelo de simulación estableciendo los parámetros fijos
→Inicialización del sistema de logging
for episodio = 1, M **do**
  →Obtención de un viento medio v_m y una semilla v_{seed} aleatorios
  →Transmisión del estado inicial al modelo en base a v_m
  →Obtener el estado inicial de las observaciones s_1
  while $t \neq T_{fin}$ & no $truncar$ **do**
    →Seleccionar una acción $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ de acuerdo con la política actual y el ruido de exploración
    →Enviar acción a_t a la simulación del entorno mediante la interfaz, y observar la recompensa r_t y el nuevo estado s_{t+1}
    →Almacenar la transición (s_t, a_t, r_t, s_{t+1}) en el buffer R
    →Tomar de forma aleatoria un minibatch de N transiciones (s_i, a_i, r_i, s_{i+1}) de R y llevar a cabo un paso de entrenamiento
    →Actualización de las redes objetivo del Actor y el Crítico
    →Comprobar el estado de simulación para terminaciones tempranas, por ejemplo: $truncar = \omega > \omega_{peligro} || P_E > P_{E_{peligro}}$
    →Almacenar el estado de simulación en el sistema de logging para el tiempo actual t

    if $t == T_{fin}$ || $truncar$ **then**
      →Presentar resultados de simulación y exportarlos al sistema
    end if
  end while
end for
→Desconexión segura de los servicios de Matlab
→Guardado de los modelos actuales

Capítulo 6

Resultados

El proceso de entrenamiento se realizó de forma manual, probando distintas configuraciones en las que se fueron puliendo tanto citados hiperparámetros como las recompensas; iterando de forma que, partiendo de un modelo que realizaba ciertas operaciones de forma correcta, se realizaban modificaciones, se cargaban de nuevo los pesos de las redes neuronales, y se realizaba un nuevo entrenamiento. En otras ocasiones, se estimaba que un modelo era inservible, entonces este se descartaba y se procedía a partir de nuevo de uno anterior, o volver a empezar con pesos aleatorios pero con las nuevas modificaciones.

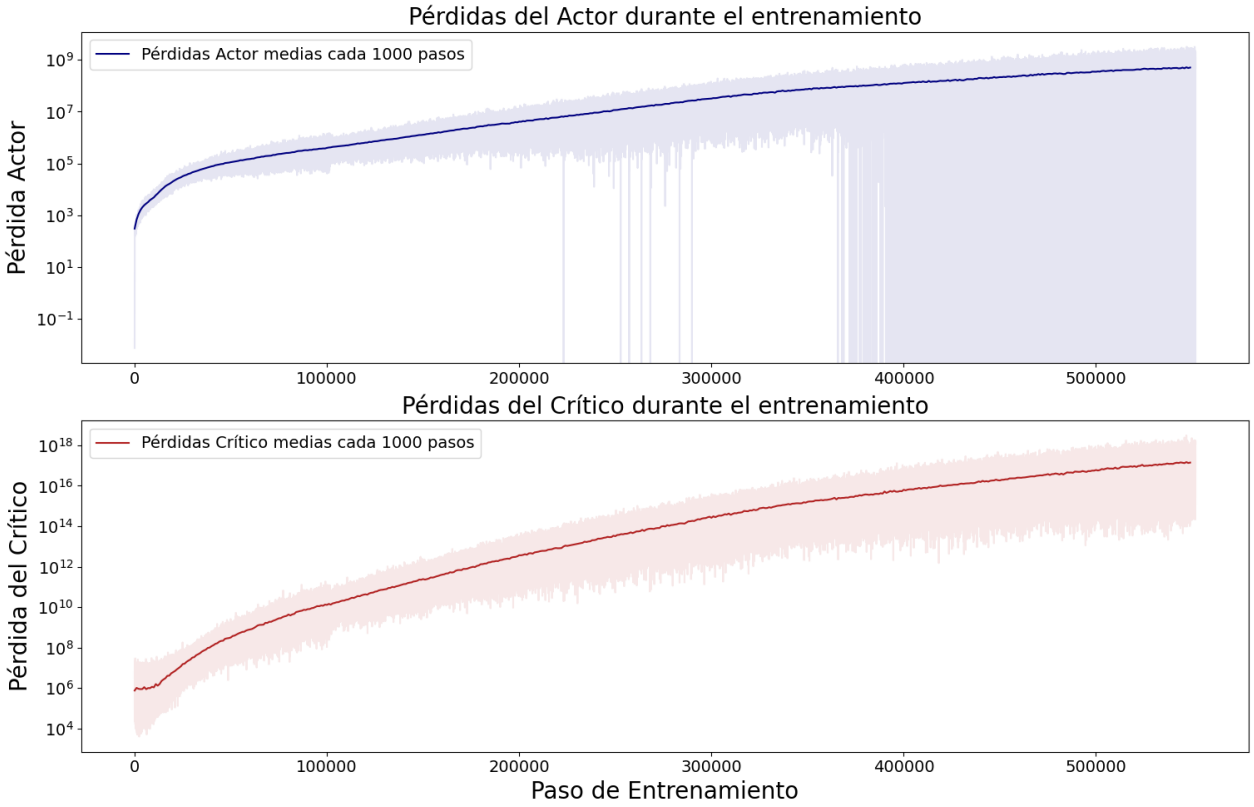

Figura 6.1: Pérdidas durante un episodio de aprendizaje defectuoso del Actor y del Crítico.

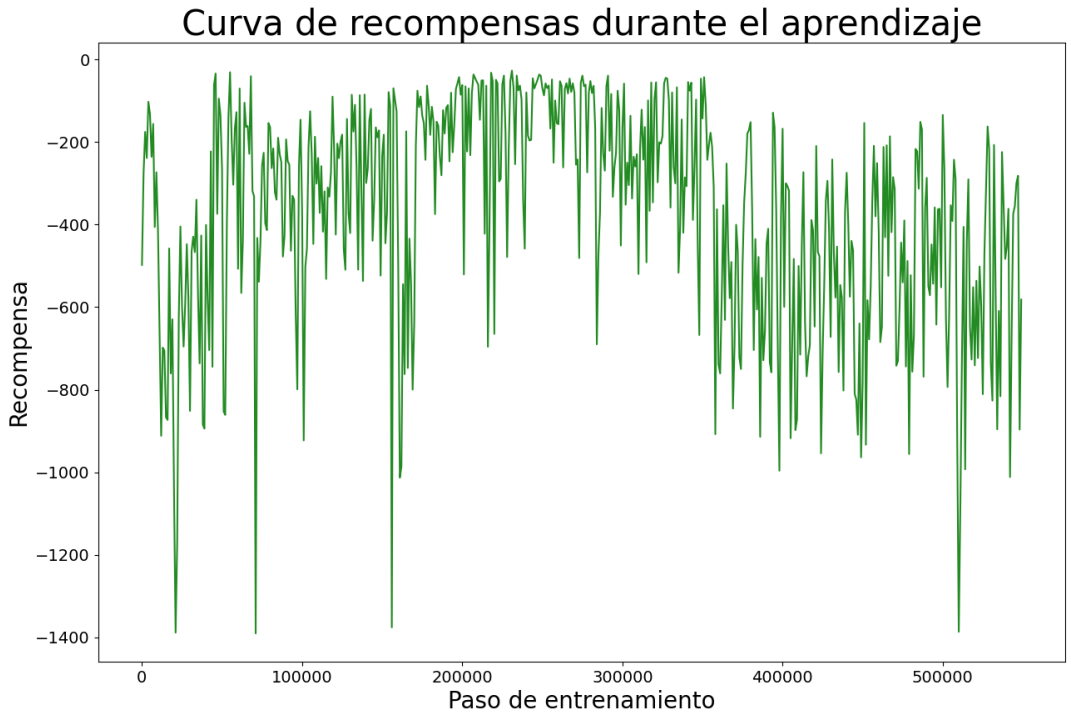

Figura 6.2: Retorno de recompensas durante un episodio de aprendizaje defectuoso.

En algunos casos se observó un problema de convergencia. Estos se pueden ver en las figuras 6.1 y 6.2, en las que se pueden ver las funciones de pérdidas y el retorno de la recompensa respectivamente. En la figura de las funciones de pérdidas de Actor y Crítico 6.1, se puede ver como estas carecen de convergencia y crecen de forma continua. Esto se atribuyó a que el factor de compensación de recompensas futuras γ era demasiado alto por situarse en un valor de 0,99. Siendo esto así, la red mostraba su incapacidad de aprendizaje al estar sujeta a una turbulencia, ya que el Crítico no es capaz de anticipar el efecto de sus acciones para un horizonte temporal demasiado largo por estar influenciado por una perturbación estocástica, lo que lleva a la optimización a fracasar. Entonces, como se puede ver en la imagen asociada al retorno de recompensa 6.2, esta no es capaz de reducirse por no estar adecuando su comportamiento al deseado.

Cabe destacar que la medida de la función de pérdidas del Actor y del Crítico no suele tomarse como un indicativo para la calidad del entrenamiento en este modelo de aprendizaje por refuerzo, sino que lo que marcará la calidad del controlador diseñado será el retorno de recompensa obtenido. No obstante, una falta de convergencia en las redes implica que este no está realizando el aprendizaje de forma adecuada, teniendo en cuenta además que en las primeras etapas de entrenamiento esta puede subir o bajar de una forma más o menos abrupta hasta que se estabiliza.

En otras ocasiones, se pudo observar el fenómeno conocido como “catastrophical for-

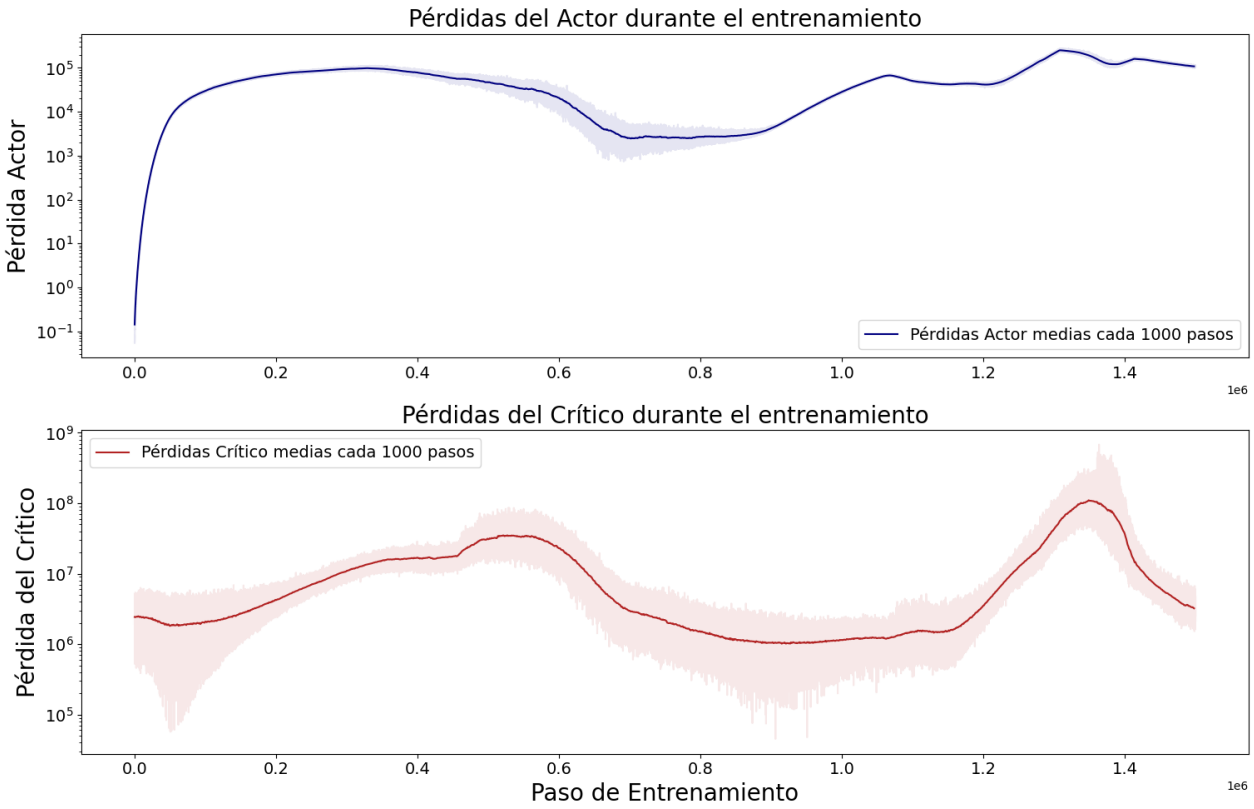

Figura 6.3: Pérdidas durante un episodio de "catastrophical forgetting" del Actor y del Crítico.

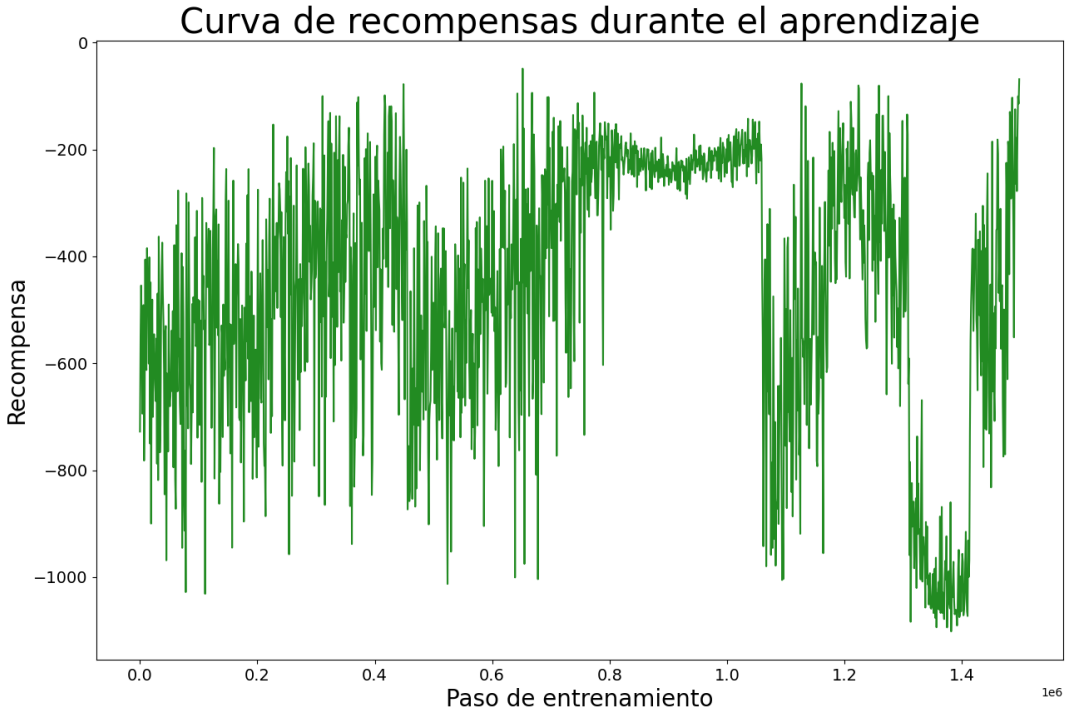

Figura 6.4: Retorno de recompensas durante un episodio de "catastrophical forgetting".

getting", fenómeno suele aparecer cuando el buffer de entrenamiento no tiene un tamaño suficiente. Entonces, cuando el modelo comienza a converger, llega a un punto con el que cuenta con demasiada información de determinado tipo y se ciñe a ella de forma exclusiva creando un overfit, haciendo entonces que el comportamiento del controlador sea excesivamente pobre al tener que afrontar cambios en el entorno como los que provoca el viento. Esto se puede observar en las imagenes 6.3 y 6.4. En la primera, se presentan las pérdidas de las redes de Actor y de Crítico y, como puede verse, en cuanto el modelo comienza a converger en ellas, el aprendizaje se vuelve inestable recreciéndose de nuevo de forma súbita. Entonces, en el reetorno de recompensa, aproximadamente en el paso de entrenamiento $1{,}2 \times 10^6$, se produce una caída repentina que devuelve este a valores iniciales, de la que parece que se recupera para caer nuevamente a un valor incluso más bajo.

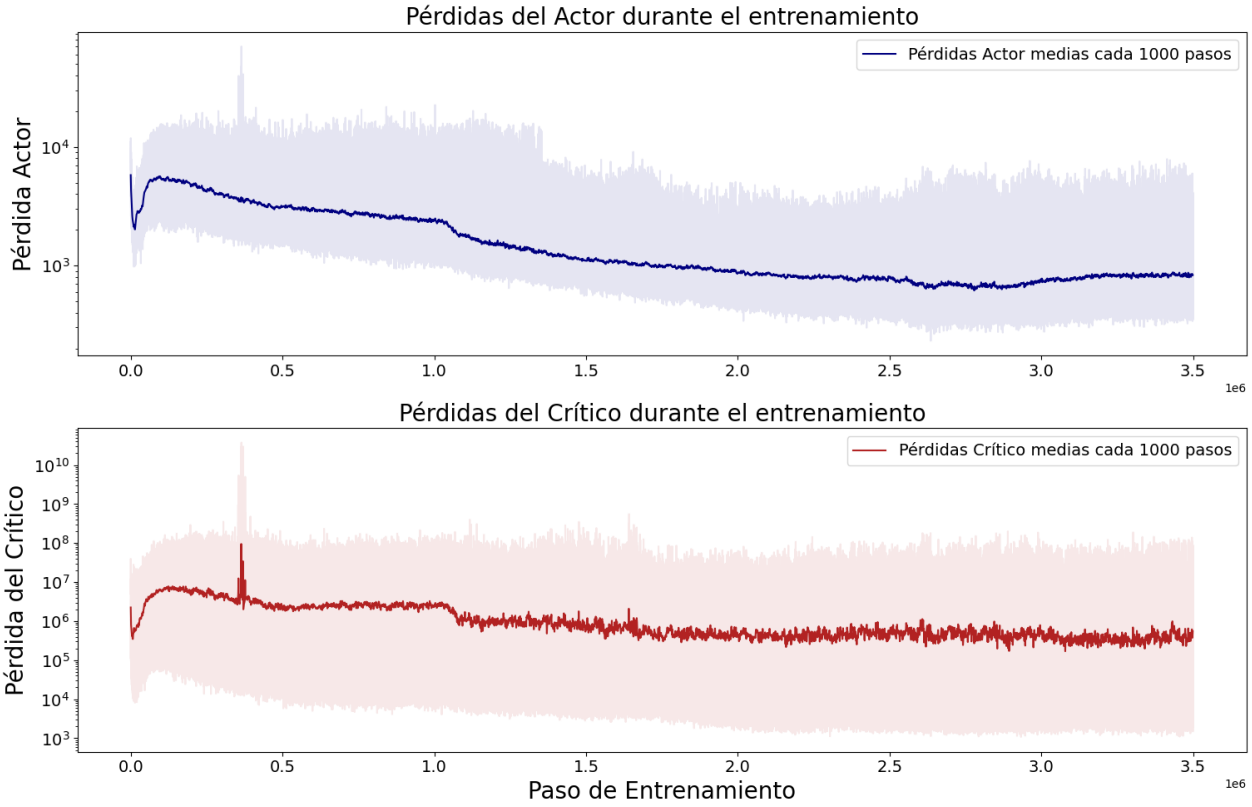

Figura 6.5: Valor de las funciones de pérdidas durante el entrenamiento final del Actor y del Crítico.

Entonces, una vez se fueron observando y solucionando estas problemáticas, se alcanzó un último modelo de controlador que es el que se presenta a continuación. El último proceso de entrenamiento del controlador ha contado con $3{,}5 \times 10^6$ pasos de entrenamiento. En la figura 6.5 se puede ver las funciones de pérdidas del Actor y del Crítico. En estas se puede apreciar una convergencia lenta, que es preferible a una excesivamente rápida que podría indicar un overfit. Si se compara esta tendencia convergente con 6.1, existe la clara diferencia de la convergencia, mientras que si se hace con 6.3, se puede verificar que el aprendizaje se

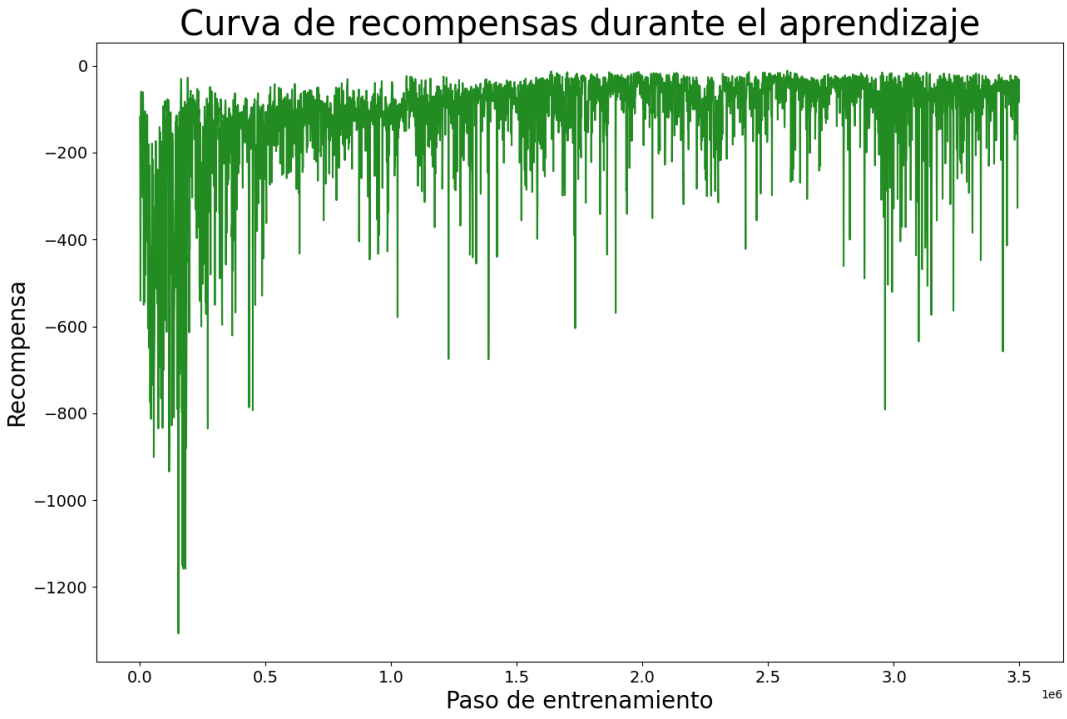

Figura 6.6: Retorno de recompensas durante el entrenamiento final del Actor y del Crítico.

realiza de forma mucho maś estable.

El retorno de recompensa del modelo final se puede ver en la figura 6.6. En este proceso, el Actor comienza ofreciendo acciones con un valor de aproximadamente -400 puntos de recompensa media, mientras que al pasar el tiempo, termina ofreciendo unos -50. Este nivel de incremento lento se debe a que, como ya se mencionó en este capítulo, el proceso de entrenamiento se realizó de forma manual e iterativa. Entonces, partiendo de un modelo entrenado previamente, se realizó un aprendizaje que buscaba mejorar lo ya obtenido con unos ratios de aprendizaje bajos. Al ser el valor 0 imposible de alcanzar, por encontrarse el modelo de aerogenerador sujeto a una perturbación constante, se consideró este valor como apto y se procedió a validar el controlador con otro tipo de pruebas.

Para comparar los resultados del controlador propuesto, se parte del último entrenamiento realizado sobre este, y se confrontan con una implementación clásica PID, similar a la que se puede ver en la figura 2.13. Las ganancias de este PID se toman del directorio en el que está disponible la turbina de referencia IEA 15MW (Gaertner et al. [2020]), y no se implementan transiciones o métodos de control adicionales para el mismo.

Entonces, una primera medida de calidad que se puede tomar, es la de comparar el modelo de control DDPG diseñado en su estado de funcionamiento final, con una simulación en la que se ha empleado el controlador PID para poder así vewrificar que los rangos y tendencias de funcionamiento son similares. Además, para poder ver la evolución del control propuesto,

se eligió un viento y una semilla de una simulación de entrenamiento con el fin de poder ver también la mejora de comportamiento que ha llevado a cabo desde sus etapas iniciales hasta su etapa final.

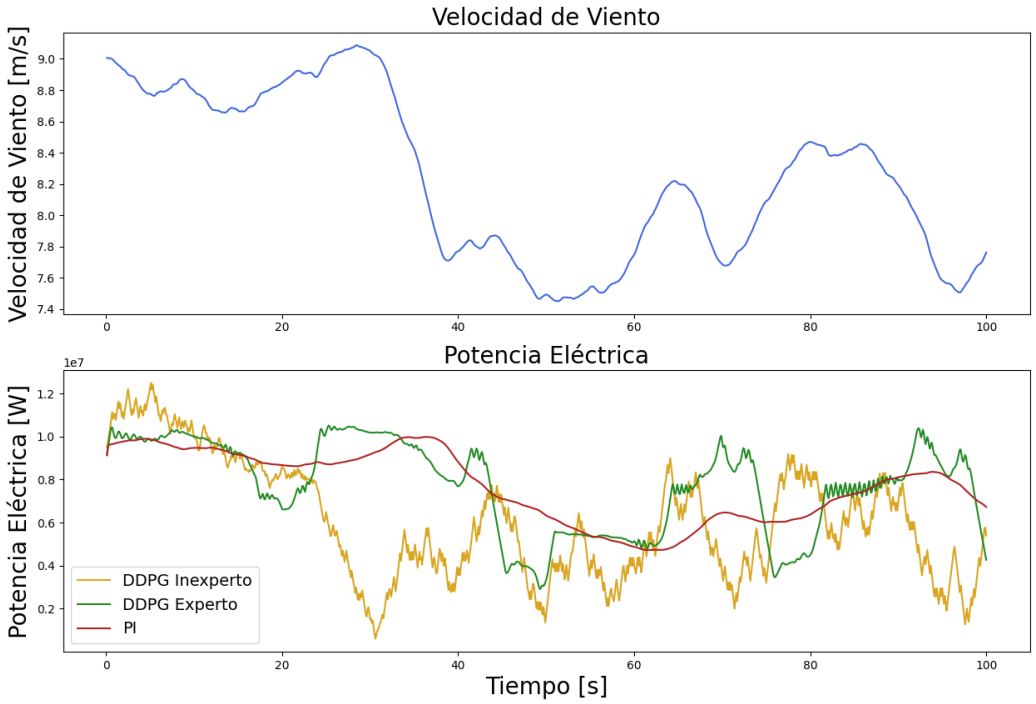

Figura 6.7: Evolución de la potencia durante el proceso de aprendizaje.

La primera comparación a realizar, sería la evolución de la potencia durante una simulación. Como puede verse en la figura 6.7, tanto los modelos de controlador DDPG, como el basado en PID, tienen un comportamiento similar, alternando los puntos en los que se sitúa uno u otro por encima en valor; con lo que se puede concluir que el control propuesto este caso ha sido capaz de controlar la potencia de forma satisfactoria. Así mismo, se puede verificar en la figura 6.8, que las líneas verde y roja pertenecientes a ambos controladores tienen tendencias parecidas a la hora de mantener la velocidad de rotor respecto al viento. Por último, el seguimiento del TSR, en ambos casos tiende a aproximarse al óptimo marcado por la línea discontínua en la figura 6.9. En cualquier caso, se puede ver que el modelo de controlador propuesto ya entrenado mejora con claridad al modelo inexperto o con menor cantidad de entrenamiento, y genera unos puntos de trabajo similares a los del control que ha propuesto el grupo encargado del diseño del aerogenerador en la simulación de pruebas.

Entonces, para llevar a cabo un proceso de validación más profundo, se realizaron 30 simulaciones seleccionando una semilla aleatoria para cada uno de los vientos de inicio $7m/s$, $8m/s$, y $9m/s$, sumando un total de 90 episodios. Se hizo entonces una comparativa entre los valores obtenidos con el controlador DDPG frente a la implementación PID clásica,

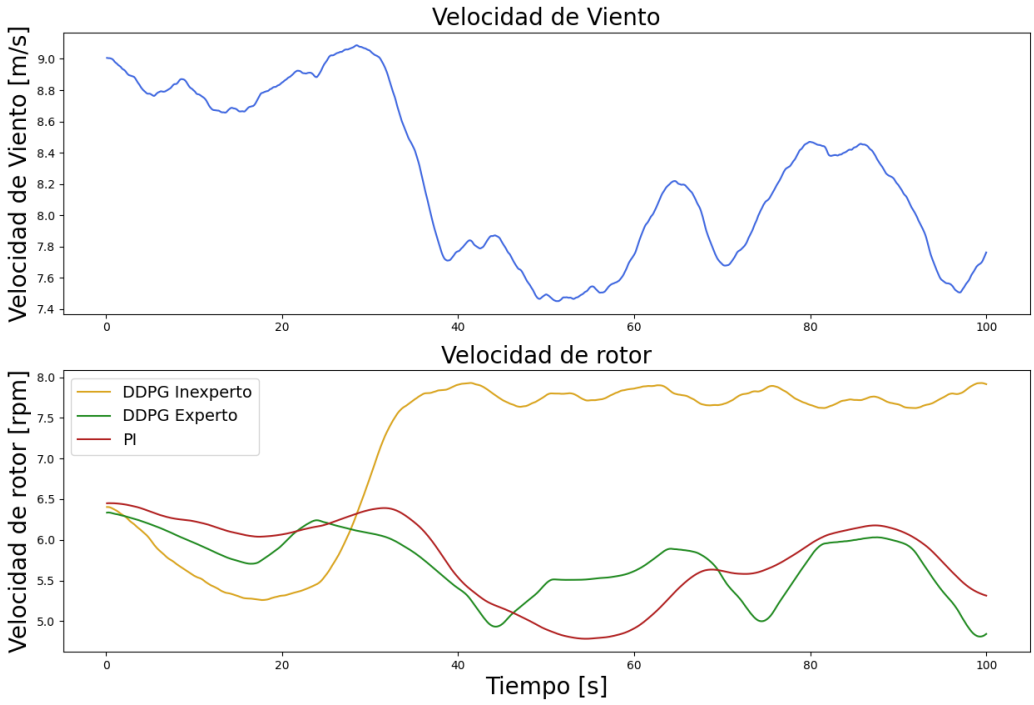

Figura 6.8: Evolución de la velocidad de rotor durante el proceso de aprendizaje.

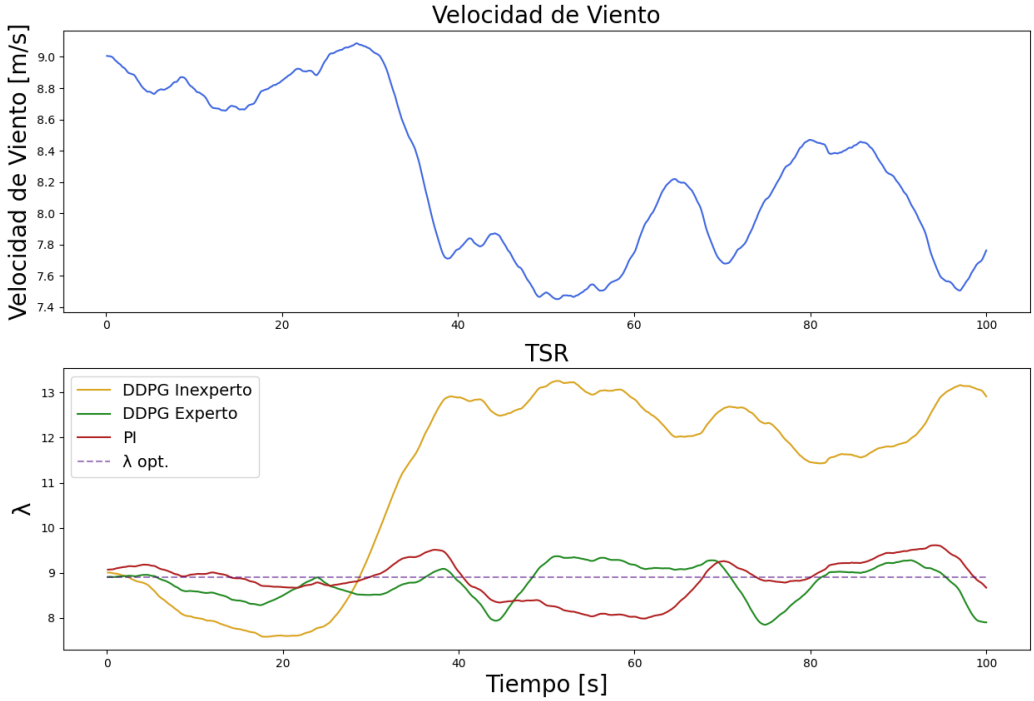

Figura 6.9: Evolución del TSR durante el proceso de aprendizaje.

enfrentando estas a los mismos perfiles de viento.

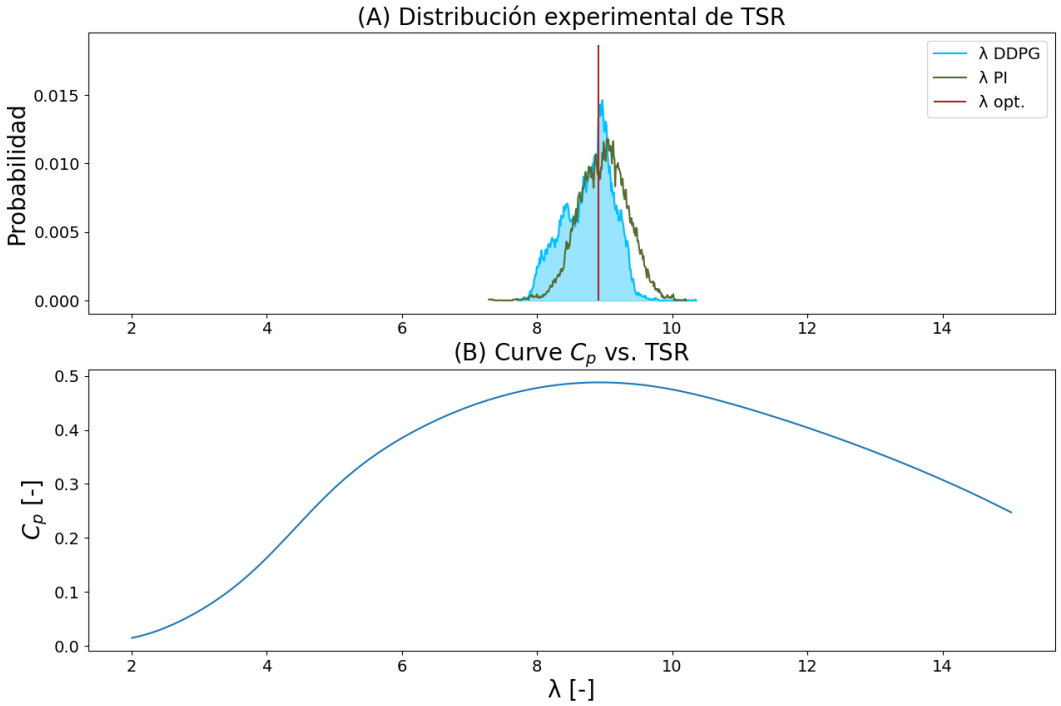

Figura 6.10: Comparativa del TSR de los controladores con el óptimo.

El primer valor que se toma para comprobar el nivel de aprendizaje, viene dado por la distribución experimental de TSR. En la figura 6.10, se puede ver la distribución que alcanza el controlador PID, y la distribución que alcanza el controlador DDPG. En ella puede verse que en ciertos puntos a la izquierda el controlador DDPG acumula más cantidad de ocurrencias que el PID, no obstante también presenta un pico más alto en el centro. El valor medio de la distribución del TSR para el DDPG es de 8,7660, compatible con el punto calculado como óptimo con Aerodyn para un C_p máximo de 8,9107. Esto demuestra que ambos controles se adaptan a la tarea de hacer un seguimiento efectivo de esta variable, siendo incluso el valor del controlador DDPG mejor en el punto central, probablemente debido a su capacidad para adaptarse mejor a las no linealidades del problema que el PID.

En la tabla 6.1 se presenta una comparativa entre los valores más relevantes del controlador DDPG y el PID para este juego de 90 simulaciones. Se puede ver que ambos están bastante parejos en el apartado de potencia, como se podría deducir de la comparativa del TSR, no obstante el nivel de producción en ese período para el controlador DDPG se incrementa en un 0,6526 %. En términos de aceleración media absoluta, el controlador propuesto incrementa un 27,4761 % el valor del control clásico, lo que quizá explicaría el incremento en las potencias. Pero no supera en ningún momento el límite marcado como dañino para esta variable que se le había impuesto de $0{,}5 rpm/s^2$. Respecto al par electromagnético medio,

el control DDPG incrementa su valor en un 5,9440 % respecto al del controlador clásico. Esto puede deberse a que especialmente en las zonas de vientos de MPPT más altos, el controlador propuesto incrementa el par para evitar entrar en sobrevelocidades que le suponen una penalización. Además, el sistema de control DDPG, habría evitado entrar en 7 situaciones provocadas por eventos indeseables que han aparecido durante las simulaciones del controlador PID, 3 infravelocidades, y 4 sobrevelocidades de parada.

Tabla 6.1: Factores de ponderamiento de las recompensas.

	DDPG	**PID**	$\mathbf{\Delta}[\%]$
$\sum \mathbf{P_E}[\mathbf{W}]$	603973974465,9016	600057762479,0137	0,6526
$\lvert\bar{\alpha}\rvert[\mathbf{rpm/s^2}]$	0,3507	0,2751	27,4761
$\bar{\tau}_{\mathbf{E}}[\mathbf{N \cdot m}]$	12346007,7112	11653325,6291	5,9440

Cabe destacar que, pese a que la comparativa puede establecer un punto de referencia para medir las capacidades del controlador DDPG, tiene que evaluarse teniendo en cuenta que el controlador PID ha sido ejecutado simplemente como un control de validación. La implementación de este último se ha hecho sólo centrándose en el seguimiento del TSR sin consideraciones adicionales como la zona inicial de arranque, que en esta máquina se sitúa por debajo de vientos de $7m/s$, o la transición entre control de par y control de pitch. Estas consideraciones y otras más sofisticadas provenientes de un control supervisorio, harían probablemente que este controlador decayera su generación de potencia y probablemente su aceleración en los límites de la zona de trabajo MPPT. No obstante, puede considerarse válido con fines comparativos como los que aquí se exponen.

Como última comprobación, se representa la concentración de puntos de potencia respecto al viento y se comparan con la potencia que se obtendría siguiendo la curva ideal obtenida mediante el código aeroelástico OpenFAST, que se ha explicado en la sección 2.3, y se ha obtenido mediante el código expuesto en el anexo B. Como puede verse en esta imagen, las coincidencias de los puntos de potencia tiende a agruparse sobre la curva, teniendo una cierta dispersión debido a la turbulencia y a que, lógicamente, los controladores no son capaces de seguir los súbitos cambios en velocidad de viento de forma instantanea en dispositivos que cuentan con tamaña inercia como los aerogeneradores.

Entonces, en base a todas las pruebas realizadas, se puede ver que el controlador DDPG propuesto ha realizado un seguimiento correcto del TSR y de la curva ideal de potencia. Es más, al compararlo con un controlador clásico creado por el grupo original de desarrollo del aerogenerador, se pudo ver que tenía tendencias similares en una simulación y a la hora de generar una distribución de TSR, y que incluso fue capaz de generar una cantidad ligeramente superior de potencia durante las ejecuciones de verificación. Con esto podría concluirse que el análisis y diseño propuestos han sido válidos, habiéndose dotado al control de la capacidad de generar las referencias adecuadas para controlar el aerogenerador IEA 15MW en su zona de trabajo de seguimiento de máxima potencia.

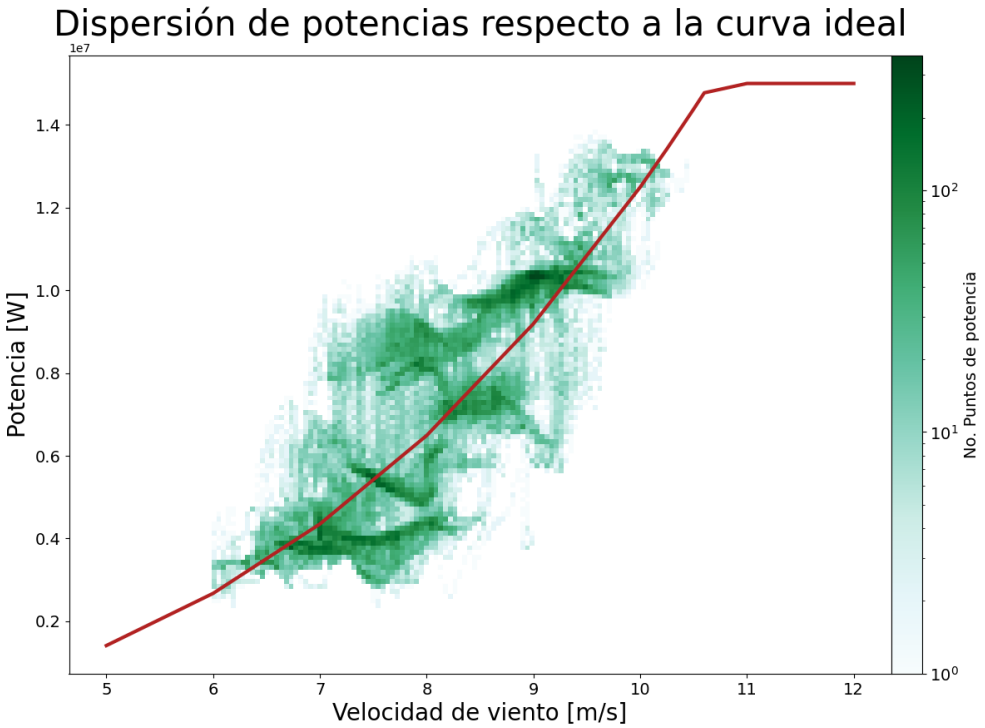

Figura 6.11: Comparación de concentración de potencias respecto a la ideal.

Capítulo 7

Conclusiones y trabajos futuros

En este trabajo se abordaron el diseño y la aplicación de un controlador de aprendizaje por refuerzo del tipo Deep Deterministic Policy Gradient para control de aerogeneradores en modo de seguimiento para su curva de máxima potencia. Para ello, se ha creado un modelo de máquina basado en el aerogenerador IEA 15MW, que permitió reducir el tiempo de entrenamiento representando fielmente variables representativas que se deseaban controlar, y que se basa en parámetros obtenidos mediante códigos más complejos. Con el fin de alimentar este modelo, se creó también un viento que representa la turbulencia de forma veraz; basado en la reproducción del espectro frecuencial de la parte inestable de un viento proveniente de un modelo meteorológico, con un ruido blanco que atraviesa un filtro de segundo orden.

Con la parte de la representación del dispositivo eólico y el viento resueltas, se procedió a crear el modelo de controlador y toda la estructura necesaria para comunicarlo con éstas. Para el diseño del controlador se plantearon y resolvieron varias cuestiones como la elección de la estructura de las neuronas, el afinamiento de hiperparámetros, o la creación de un ruido de exploración válido que hicieran posible su entrenamiento. Además, una de las principales contribuciones fue el diseño de un sistema de recompensas propio que persigue varios objetivos y no únicamente la generación de la máxima potencia bruta.

Finalmente, después de superar un proceso de entrenamiento y con el fin de validar el aprendizaje, se comparó el comportamiento del controlador propuesto con uno clásico PID, para el cual se siguió la estructura y la sintonía disponibles en el directorio de la turbina de referencia elegida. En estas pruebas se pudo verificar un funcionamiento equivalente entre ambos controladores, llegando incluso a superar el controlador propuesto al de referencia en términos como la potencia generada.

Dado que los controladores basados en aprendizaje por refuerzo están todavía introduciéndose en todos los campos, y en el de la energía eólica en particular todavía no se encuentra muy avanzado, esta propuesta presenta un punto de partida del cual pueden nacer nuevos trabajos y exploraciones. Cabría la posibilidad, por ejemplo, de afinar más cuidadosamente los hiperparámetros del modelo propuesto empleando entornos de sintonización que permitan

obtener una mejor respuesta general, ya que los modelos DDPG son sensibles a la elección de estos parámetros. Otra posibilidad, sería la revisión de las recompensas para una mejor respuesta, intentando, por ejemplo, reducir las aceleraciones del rotor del aerogenerador para que sus respuestas fueran más suaves.

Otros estudios más ambiciosos implicarían la ampliación de este trabajo para, por ejemplo, introducir también el control del ángulo de pitch de las palas con el fin de poder obtener un controlador que funcione en todo el rango operacional de la máquina. Podría también repensarse las entradas de los estados para excluir la velocidad del viento, ya que como se mencionó durante el trabajo, la lectura de los vientos es inexacta y puede estar sujeta a efectos nocivos como las sombras introducidas por el paso de pala por delante del sensor. También cabría la posibilidad de reemplazar el modelo por otros que tengan un aprendizaje más estable, como el TD3, o incluso hacer esta implementación con otros modelos más recientes que, aunque más complejos, presentan unas capacidades más robustas como el PPO.

Para finalizar, este tipo de enfoque, podrían ayudar a la creación de controles basados en el aprendizaje por refuerzo más avanzados, que emplearan un código aeroelástico y se centraran en rebajar cargas operacionales, especialmente cargas de fatiga ya que son uno de los elementos más limitantes a la hora de plantear el diseño de estas máquinas.

Bibliografía

Global Wind Energy Council. Global wind energy council. global wind 2019 report annual market update. Report, 2020.

Red Eléctrica Española. Informe del sistema eléctrico 2022. Report, 2023.

Red Eléctrica Española. Informe del sistema eléctrico 2023. Report, 2024.

R.H. Lasseter. Microgrids. In *2002 IEEE Power Engineering Society Winter Meeting. Conference Proceedings (Cat. No.02CH37309)*, volume 1, pages 305–308 vol.1, 2002. doi: 10.1109/PESW.2002.985003.

Tomislav Dragicevic, Juan C. Vasquez, Josep M. Guerrero, and Davor Skrlec. Advanced lvdc electrical power architectures and microgrids: A step toward a new generation of power distribution networks. *IEEE Electrification Magazine*, 2(1):54–65, 2014. doi: 10.1109/MELE.2013.2297033.

Khwaja Rahman and Silva Hiti. Electric vehicle proliferation and trends in developing drive system components [about this issue]. *IEEE Electrification Magazine*, 7(1):2–4, 2019. doi: 10.1109/MELE.2018.2889543.

Tomislav Dragičević, Xiaonan Lu, Juan C. Vasquez, and Josep M. Guerrero. Dc microgrids—part i: A review of control strategies and stabilization techniques. *IEEE Transactions on Power Electronics*, 31(7):4876–4891, 2016a. doi: 10.1109/TPEL.2015.2478859.

Tomislav Dragičević, Xiaonan Lu, Juan C. Vasquez, and Josep M. Guerrero. Dc microgrids—part ii: A review of power architectures, applications, and standardization issues. *IEEE Transactions on Power Electronics*, 31(5):3528–3549, 2016b. doi: 10.1109/TPEL.2015.2464277.

A. R. Nejad, J. Keller, Y. Guo, S. Sheng, H. Polinder, S. Watson, J. Dong, Z. Qin, A. Ebrahimi, R. Schelenz, F. Gutiérrez Guzmán, D. Cornel, R. Golafshan, G. Jacobs, B. Blockmans, J. Bosmans, B. Pluymers, J. Carroll, S. Koukoura, E. Hart, A. McDonald, A. Natarajan, J. Torsvik, F. K. Moghadam, P.-J. Daems, T. Verstraeten, C. Peeters, and J. Helsen. Wind turbine drivetrains: state-of-the-art technologies and future development

trends. *Wind Energy Science*, 7(1):387–411, 2022. doi: 10.5194/wes-7-387-2022. URL https://wes.copernicus.org/articles/7/387/2022/.

Wagner, Hermann-Josef. Introduction to wind energy systems(*). *EPJ Web Conf.*, 246: 00004, 2020. doi: 10.1051/epjconf/202024600004. URL https://doi.org/10.1051/epjconf/202024600004.

P J Darrow. Wind turbine control design to reduce capital costs: 7 january 2009 - 31 august 2009. Technical report, National Renewable Energy Lab. (NREL), Golden, CO (United States), 01 2010. URL https://www.osti.gov/biblio/971098.

E. A. Bossanyi. The design of closed loop controllers for wind turbines. *Wind Energy*, 3 (3):149–163, 2000. doi: https://doi.org/10.1002/we.34. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/we.34.

Evan Gaertner, Jennifer Rinker, Latha Sethuraman, Frederik Zahle, Benjamin Anderson, Garrett Barter, Nikhar Abbas, Fanzhong Meng, Pietro Bortolotti, Witold Skrzypinski, George Scott, Roland Feil, Henrik Bredmose, Katherine Dykes, Matt Sheilds, Christopher Allen, and Anthony Viselli. Definition of the IEA 15-megawatt offshore reference wind turbine. Technical report, International Energy Agency, 2020. URL https://www.nrel.gov/docs/fy20osti/75698.pdf.

The MathWorks Inc. Matlab version: 24.2.0.2863752 (r2024b), 2024. URL https://www.mathworks.com.

NREL. Openfast (formerly known as fast), 2025. URL https://github.com/OpenFAST.

Guido van Rossum and Fred L Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

François Chollet. Keras. https://github.com/fchollet/keras, 2015.

R.D. Barrio, Matilde. Santos Peñas, and Carlos Armentá Deu. Sistema de orientación de un aerogenerador empleando fuzzy logic controller. *XLV Jornadas de Automática*, 2024,

2024. doi: https://doi.org/10.17979/ja-cea.2024.45.10825. URL https://revistas.udc.es/index.php/JA_CEA/article/view/10825.

S. Heier. *Grid Integration of Wind Energy: Onshore and Offshore Conversion Systems*. Wiley, 2014. ISBN 9781118703298. URL https://books.google.es/books?id=d0FsAwAAQBAJ.

T.L. Burton, N. Jenkins, E. Bossanyi, D. Sharpe, and M. Graham. *Wind Energy Handbook*. Wiley, 2021. ISBN 9781119451167. URL https://books.google.es/books?id=XFYrEAAAQBAJ.

J.F. Manwell, J.G. McGowan, and A.L. Rogers. *Wind Energy Explained: Theory, Design and Application*. Wiley, 2010. ISBN 9780470686287. URL https://books.google.es/books?id=roaTx_Of0vAC.

T. Ackermann. *Wind Power in Power Systems*. Wiley, 2012. ISBN 9781119942085. URL https://books.google.es/books?id=y7430s86pQAC.

O. Apata and D.T.O. Oyedokun. An overview of control techniques for wind turbine systems. *Scientific African*, 10:e00566, 2020. ISSN 2468-2276. doi: https://doi.org/10.1016/j.sciaf.2020.e00566. URL https://www.sciencedirect.com/science/article/pii/S2468227620303045.

N. Abbas, D. Zalkind, L. Pao, and A. Wright. A reference open-source controller for fixed and floating offshore wind turbines. *Wind Energy Science Discussions*, 2021:1–33, 2021. doi: 10.5194/wes-2021-19. URL https://wes.copernicus.org/preprints/wes-2021-19/.

A D Wright. Modern control design for flexible wind turbines. Technical report, National Renewable Energy Lab. (NREL), Golden, CO (United States), 07 2004a. URL https://www.osti.gov/biblio/15011696.

A D Wright and L J Fingersh. Advanced control design for wind turbines; part i: Control design, implementation, and initial tests. Technical report, National Renewable Energy Lab. (NREL), Golden, CO (United States), 03 2008a. URL https://www.osti.gov/biblio/927269.

Fernando Bianchi, Hernán De Battista, and Ricardo Julian Mantz. *Wind Turbine Control Systems: Principles, Modelling and Gain Scheduling Design*. Springer, 01 2007.

Torben Knudsen, Thomas Bak, and Mohsen Soltani. Prediction models for wind speed at turbine locations in a wind farm. *Wind Energy*, 14(7):877–894, 2011. doi: https://doi.org/10.1002/we.491. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/we.491.

J Jonkman, S Butterfield, W Musial, and G Scott. Definition of a 5-mw reference wind turbine for offshore system development. Technical report, National Renewable Energy Lab.

(NREL), Golden, CO (United States), 02 2009. URL https://www.osti.gov/biblio/947422.

David Schlipf. Set-point-fading for wind turbine control, December 2019. URL https://doi.org/10.5281/zenodo.5567358.

Alan Wright. Designing and testing contols to mitigate dynamic loads in the controls advanced research turbine: Preprint. *2008 ASME Wind Energy Symposium*, 2008. 2008 ASME Wind Energy Symposium ; Conference date: 07-01-2008 Through 10-01-2008.

A D Wright and L J Fingersh. Advanced control design for wind turbines; part i: Control design, implementation, and initial tests. Technical report, National Renewable Energy Lab. (NREL), Golden, CO (United States), 03 2008b. URL https://www.osti.gov/biblio/927269.

Alan Wright. Design of controls to attenuate loads in the controls advanced research turbine: Preprint. *2004 ASME Wind Energy Symposium*, 2003. 2004 ASME Wind Energy Symposium ; Conference date: 05-01-2004 Through 08-01-2004.

A D Wright. Modern control design for flexible wind turbines. Technical report, National Renewable Energy Lab. (NREL), Golden, CO (United States), 07 2004b. URL https://www.osti.gov/biblio/15011696.

Raheem Alhamdawee and M. Manzoor Hussain. A study of conventional and modern algorithms employed for mppt in wind energy conversion systems: A review. In *2024 3rd International conference on Power Electronics and IoT Applications in Renewable Energy and its Control (PARC)*, pages 14–23, 2024. doi: 10.1109/PARC59193.2024.10486569.

Eduardo Muñoz-Palomeque, Jesús Enrique Sierra-García, and Matilde Santos. Técnicas de control inteligente para el seguimiento del punto de máxima potencia en turbinas eólicas. *Revista Iberoamericana de Automática e Informática industrial*, 21(3):193–204, mar. 2024. doi: 10.4995/riai.2024.21097. URL https://polipapers.upv.es/index.php/RIAI/article/view/21097.

Burkart Ralph, Kostas Margellos, and John Lygeros. Nonlinear control of wind turbines: An approach based on switched linear systems and feedback linearization. *Proceedings of the IEEE Conference on Decision and Control*, pages 5485–5490, 12 2011. doi: 10.1109/CDC.2011.6160709.

J. E. Sierra-García and M. Santos. Redes neuronales y aprendizaje por refuerzo en el control de turbinas eólicas. *Revista Iberoamericana de Automática e Informática industrial*, 18(4):327–335, sep. 2021. doi: 10.4995/riai.2021.16111. URL https://polipapers.upv.es/index.php/RIAI/article/view/16111.

Iec 61400-1:2019, 2019. URL https://webstore.iec.ch/en/publication/26423. Specifies essential design requirements to ensure the structural integrity of wind turbines and provide an appropriate level of protection against damage from all hazards during the planned lifetime.

Aditya K. Sabale and Nagendra K. V. Gopal. Nonlinear aeroelastic analysis of large wind turbines under turbulent wind conditions. *AIAA Journal*, 57(10):4416–4432, 2019. doi: 10.2514/1.J057404. URL https://doi.org/10.2514/1.J057404.

Sven Creutz Thomsen. Nonlinear control of a wind turbine, 2006. URL https://api.semanticscholar.org/CorpusID:107795785.

Jørgen Højstrup. Velocity spectra in the unstable planetary boundary layer. *Journal of Atmospheric Sciences*, 39(10):2239 – 2248, 1982. doi: 10.1175/1520-0469(1982)039<2239:VSITUP>2.0.CO;2. URL https://journals.ametsoc.org/view/journals/atsc/39/10/1520-0469_1982_039_2239_vsitup_2_0_co_2.xml.

Torben Knudsen. Regulering af vindmøller, 01 1983.

DTU. Hawc2, 2025. URL https://www.hawc2.dk/.

DNV. Bladed, 2025. URL https://www.dnv.com/software/services/bladed/.

Kim Branner, José Blasques, Taeseong Kim, Vladimir Fedorov, Peter Berring, Robert Bitsche, and Christian Berggreen. Anisotropic beam model for analysis and design of passive controlled wind turbine blades, 02 2012.

J M Jonkman and B J Jonkman. Fast modularization framework for wind turbine simulation: full-system linearization. *Journal of Physics: Conference Series*, 753(8):082010, sep 2016. doi: 10.1088/1742-6596/753/8/082010. URL https://dx.doi.org/10.1088/1742-6596/753/8/082010.

Omessaad Elbeji, Ben Mouna, and Sbita Lassaad. Pmsg wind energy conversion system: Modeling and control. *International Journal of Modern Nonlinear Theory and Application*, 03:88–97, 01 2014. doi: 10.4236/ijmnta.2014.33011.

Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN 9780262352703. URL https://books.google.es/books?id=uWV0DwAAQBAJ.

OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019. URL https://arxiv.org/abs/1912.06680.

Selman Djeffal, Mohamed Razi Morakchi, Abdelhamid Ghoul, and Turhan Can Kargin. Ddpg-based reinforcement learning for controlling a spatial three-section continuum robot. *Franklin Open*, 6:100077, 2024. ISSN 2773-1863. doi: https://doi.org/10.1016/j.fraope.2024.100077. URL https://www.sciencedirect.com/science/article/pii/S2773186324000082.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety, 2016. URL https://arxiv.org/abs/1606.06565.

OpenAI. Kinds of rl algorithms, 2025a. URL "https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html".

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019. URL https://arxiv.org/abs/1509.02971.

Andrea Asperti and Marco Brutto. Microracer: a didactic environment for deep reinforcement learning. *Lecture Notes in Computer Science*, To appear, 07 2022.

OpenAI. Better exploration with parameter noise, 2025b. URL "https://openai.com/index/better-exploration-with-parameter-noise/".

Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration, 2018. URL https://arxiv.org/abs/1706.01905.

Ashish Kushwaha, Madan Gopal, and Bhim Singh. Q-learning based maximum power extraction for wind energy conversion system with variable wind speed. *IEEE Transactions on Energy Conversion*, 35(3):1160–1170, 2020. doi: 10.1109/TEC.2020.2990937.

Chun Wei, Zhe Zhang, Wei Qiao, and Liyan Qu. Reinforcement-learning-based intelligent maximum power point tracking control for wind energy conversion systems. *IEEE Transactions on Industrial Electronics*, 62(10):6360–6370, 2015. doi: 10.1109/TIE.2015.2420792.

Peng Chen and Dezhi Han. Reward adaptive wind power tracking control based on deep deterministic policy gradient. *Applied Energy*, 348:121519, 2023. ISSN 0306-2619. doi:

https://doi.org/10.1016/j.apenergy.2023.121519. URL https://www.sciencedirect.com/science/article/pii/S0306261923008838.

Darkhan Zholtayev, Matteo Rubagotti, and Ton Duc Do. Deep reinforcement learning for pmsg wind turbine control via twin delayed deep deterministic policy gradient (td3). *Optimal Control Applications and Methods*, 45(4):1889–1906, 2024. doi: https://doi.org/10.1002/oca.3129. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/oca.3129.

Eduardo Muñoz-Palomeque, J Enrique Sierra-García, and Matilde Santos. Wind turbine maximum power point tracking control based on unsupervised neural networks. *Journal of Computational Design and Engineering*, 10(1):108–121, 12 2022. ISSN 2288-5048. doi: 10.1093/jcde/qwac132. URL https://doi.org/10.1093/jcde/qwac132.

J. Enrique Sierra-Garcia, Matilde Santos, and Ravi Pandit. Wind turbine pitch reinforcement learning control improved by pid regulator and learning observer. *Engineering Applications of Artificial Intelligence*, 111:104769, 2022. ISSN 0952-1976. doi: https://doi.org/10.1016/j.engappai.2022.104769. URL https://www.sciencedirect.com/science/article/pii/S0952197622000598.

Marion Coquelet, Laurent Bricteux, Maud Moens, and Philippe Chatelain. A reinforcement-learning approach for individual pitch control. *Wind Energy*, 25(8):1343–1362, 2022. doi: https://doi.org/10.1002/we.2734. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/we.2734.

Jesús Enrique Sierra-García and Matilde Santos. Exploring reward strategies for wind turbine pitch control by reinforcement learning. *Applied Sciences*, 10(21), 2020. ISSN 2076-3417. doi: 10.3390/app10217462. URL https://www.mdpi.com/2076-3417/10/21/7462.

Oscar Carranza Castillo, Viviana Reyes Andrade, Jaime José Rodríguez Rivas, and Rubén Ortega González. Comparison of power coefficients in wind turbines considering the tip speed ratio and blade pitch angle. *Energies*, 16(6), 2023. ISSN 1996-1073. doi: 10.3390/en16062774. URL https://www.mdpi.com/1996-1073/16/6/2774.

J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 01 1965. ISSN 0010-4620. doi: 10.1093/comjnl/7.4.308. URL https://doi.org/10.1093/comjnl/7.4.308.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. URL https://arxiv.org/abs/1502.03167.

Thao Nguyen, Maithra Raghu, and Simon Kornblith. Do wide and deep networks learn the same things? uncovering how neural network representations vary with width and depth, 2021. URL https://arxiv.org/abs/2010.15327.

Xiaoxiao Guo. Deep learning and reward design for reinforcement learning. 2017. URL https://api.semanticscholar.org/CorpusID:64985440.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032.

Apéndice A

Relación de valores del aerogenerador empleados

A continuación se listan en la tabla A.1 los parámetros de máquina relevantes obtenidos de la fuente principal (Gaertner et al. [2020]), con algunos estimados por el autor. Cabe destacar que existen muchos más parámetros, como la configuración aerodinámica de los perfiles, datos de rigideces de materiales, etc. con los que se han realizado simulaciones. No obstante no se incluye ya que se pretende proporcionar la información más representativa.

Tabla A.1: Parámetros del aerogenerador.

Variable	**Valor**	**Unidades**	**Origen**
Potencia nominal	15	MW	IEA15MW
Clase	IEC IB	-	IEA15MW
Ratio de potencia	326,2767	W/m^2	IEA15MW
Orientación	Upwind	-	IEA15MW
Control	VS-VP	-	IEA15MW
Viento de entrada	3	m/s	IEA15MW
Viento de salida	25	m/s	IEA15MW
Altura de hub	150	m	IEA15MW
Velocidad mínima	5	rpm	IEA15MW
Velocidad nominal	7,56	rpm	IEA15MW
Diámetro de rotor	241,94	m	IEA15MW
Densidad del aire	1,225	m	IEA15MW
Rendimiento generador	95,756	%	IEA15MW
Ratio máximo de par	4,5	$MN \cdot m/s$	IEA15MW
Ratio mínimo de par	$-4{,}5$	$MN \cdot m/s$	IEA15MW
Ratio máximo de pitch	2	$\deg/s$	IEA15MW
Ratio mínimo de pitch	-2	$\deg/s$	IEA15MW
Inercia de rotor	$3{,}5246 \times 10^8$	$kg \cdot m^2$	IEA15MW
Coeficiente de shear	0,12	-	IEA15MW
Velocidad de apagado	4	rpm	Autor
Velocidad de emergencia	8	rpm	Autor
Potencia máxima	18	MW	Autor
Intensidad de turbulencia	13,55	%	Autor

Apéndice B

Código para la obtención de curvas de potencia de un aerogenerador con OpenFAST

Este módulo implementa el código que permite generar y ejecutar múltiples simulaciones con distintos vientos estáticos con el fin de obtener las curvas de potencia de un aergoenerador con OpenFAST, apoyándose en la suite openfast_toolbox gestionada por los mismos creadores del código aeroelástico.

```
1  import numpy as np
2  import os
3  from typing import Dict,Literal
4
5  import openfast_toolbox.case_generation.case_gen as case_gen
6  import openfast_toolbox.case_generation.runner as runner
7  import openfast_toolbox.postpro as postpro
8
9
10 def power_curve_parametric(fastExecutable:str,refDir:str,mainFile:str
   ,opPoints:Dict[str,Dict[Literal['wind_speed','rotor_speed','pitch'
   ],float]],workDir:str):
11     """ Example to run a set of FAST simulations to determine a power
    curve.
12     In this example, the WS, RPM and Pitch are set within a for loop.
13     If the controller and generator are active, these are just "
   initial conditions".
14     Additional parameters may be set by adjusting the BaseDict.
15
16     This script is based on a reference directory which contains a
   reference main input file (.fst)
```

```
17      Everything is copied to a working directory.
18      The different fast inputs are generated based on a list of
    dictionaries, named ‘PARAMS‘.
19      For each dictionary:
20          - they keys are "path" to a input parameter, e.g. ‘EDFile|
    RotSpeed‘  or ‘TMax‘.
21              These should correspond to whater name of the variable is
    used in the FAST inputs files.
22          - they values are the values corresponding to this parameter
23      """
24      # --- Defining the parametric study  (list of dictionnaries with
    keys as FAST parameters)
25      BaseDict = {'TMax': 100, 'DT': 0.01, 'DT_Out': 0.1}
26      #BaseDict = case_gen.paramsNoController(BaseDict)
27      #BaseDict = case_gen.paramsStiff(BaseDict)
28      #BaseDict = case_gen.paramsNoGen(BaseDict)
29      PARAMS=[]
30      # for wsp,rpm,pitch in zip(WS,RPM,PITCH): # NOTE: same length of
    WS and RPM otherwise do multiple for loops
31      for point,vals in opPoints.items():
32          wsp = vals['wind_speed']
33          rpm = vals['rotor_speed']
34          pitch = vals['pitch']
35          # wsp,rpm,pitch
36          p=BaseDict.copy()
37          p['EDFile|RotSpeed']        = rpm
38          p['EDFile|BlPitch(1)']      = pitch
39          p['EDFile|BlPitch(2)']      = pitch
40          p['EDFile|BlPitch(3)']      = pitch
41          p['InflowFile|HWindSpeed'] = wsp
42          p['InflowFile|WindType']    = 1 # Setting steady wind
43          p['__name__']               = 'ws{:04.1f}'.format(p['
    InflowFile|HWindSpeed'])
44          PARAMS.append(p)
45
46      # --- Generating all files in a workdir
47      fastFiles=case_gen.templateReplace(PARAMS, refDir, workDir,
    removeRefSubFiles=True, main_file=mainFile)
48      print(fastFiles)
49
50      # --- Creating a batch script just in case
```

```
51      runner.writeBatch(os.path.join(workDir,'_RUN_ALL.bat'), fastFiles
    ,fastExe=fastExecutable)
52      # --- Running the simulations
53      print('>>> Running {} simulations in {} ...'.format(len(fastFiles
    ), workDir))
54      runner.run_fastfiles(fastFiles, fastExe=fastExecutable, parallel=
    True, showOutputs=False, nCores=2)
55
56      # --- Simple Postprocessing
57      outFiles = [os.path.splitext(f)[0]+'.outb' for f in fastFiles]
58
59      avg_results = postpro.averagePostPro(outFiles,avgMethod='
    constantwindow',avgParam=10, ColMap = {'WS_[m/s]':'Wind1VelX_[m/s]
    '},ColSort='WS_[m/s]')
60      print('>>> Average results:')
61      print(avg_results)
62      avg_results.to_csv(os.path.join(workDir,'_PowerCurve1.csv'),sep='
    \t',index=False)
63
64
65
66  if __name__=='__main__':
67      # asdfasdf
68      opPoints:Dict[str,Dict[Literal['wind_speed','rotor_speed','pitch'
    ],float]]={}
69      opPoints['3']={'wind_speed':3,'rotor_speed':5,'pitch':1}
70      opPoints['4']={'wind_speed':4,'rotor_speed':5.25,'pitch':1}
71      opPoints['5']={'wind_speed':5,'rotor_speed':5.5,'pitch':0}
72      opPoints['6']={'wind_speed':6,'rotor_speed':5.75,'pitch':0}
73      opPoints['7']={'wind_speed':7,'rotor_speed':6,'pitch':0}
74      opPoints['8']={'wind_speed':8,'rotor_speed':6.5,'pitch':0}
75      opPoints['9']={'wind_speed':9,'rotor_speed':6.75,'pitch':0}
76      opPoints['10']={'wind_speed':10,'rotor_speed':7,'pitch':0}
77      opPoints['10.25']={'wind_speed':10.25,'rotor_speed':7.25,'pitch'
    :0}
78      opPoints['10.5']={'wind_speed':10.5,'rotor_speed':7.5,'pitch':0}
79      opPoints['10.55']={'wind_speed':10.55,'rotor_speed':7.56,'pitch'
    :0}
80      opPoints['10.6']={'wind_speed':10.6,'rotor_speed':7.56,'pitch':0}
81      opPoints['11']={'wind_speed':11,'rotor_speed':7.56,'pitch':5}
82      opPoints['12']={'wind_speed':12,'rotor_speed':7.56,'pitch':7.5}
83      opPoints['13']={'wind_speed':13,'rotor_speed':7.56,'pitch':10}
```

```
84     opPoints['14']={'wind_speed':14,'rotor_speed':7.56,'pitch':12.5}
85     opPoints['15']={'wind_speed':15,'rotor_speed':7.56,'pitch':15}
86     # WS    = [3,5,7,9 ,11,13,15]
87     # RPM   = [5,5.5,6,7,7.56,7.56,7.56] # initial conditions
88     # PITCH = [0,0,0,0 ,5 ,10,15] # initial conditions
89     # --- Parameters for this script
90     fast_executable  = r'/usr/local/bin/openfast' # Location of a
    FAST executable (and dll)
91     ref_dir   = r'/dummy/path/to/IEA-15-240-RWT-Monopile/'     #
    Folder where the fast input files are located (will be copied)
92     main_file = 'IEA-15-240-RWT-Monopile.fst'               # Main
    file in ref_dir, used as a template
93     work_dir  = r'/dummy/path/to/out/Power_Curve_Parametric/'     #
    Output folder (will be created)
94     power_curve_parametric(fast_executable,ref_dir,main_file,opPoints
    ,work_dir)
```

Apéndice C

Código para la obtención de la superficie de Coeficiente de Potencia

En este anexo se detalla el código empleado para poder obtener las superficies de potencia. Se emplean varios módulos, uno para generar los casos de simulación de con el módulo Aerodyn de Openfast, y otro que permite la recolección de los datos de simulación y evaluación de los mismos.

C.1. Código para la generación de los csv con los casos de barrido

A continuación se presenta el código necesario para obtener los casos a simular en Aerodyn empleando los diferentes aproximaciones implementadas en distintas funciones. La que ha resultado ser más óptima desde el punto de vista computacional, al tener que llevar a cabo menos simulaciones, es la función que está denominada como "create_aerodyn_cases_for_tsr".

```
1 import numpy as np
2 import pandas as pd
3 from typing import List
4 # FORMAT TO INSERT CASES
5 #           4  NumCases      - Number of cases to run
6 # HWndSpeed  PLExp  RotSpd  Pitch   Yaw   dT    Tmax  DOF  Amplitude
     Frequency
7 # (m/s)      (-)    (rpm)   (deg)  (deg)  (s)   (s)   (-)   (-)
      (Hz)
8 #   7.15       0.12     5.08       0.       0.    0.4    100     0     0
          0
9 #   8.0        0.0      6.        0.        0.    1.0    100     0     0
       0
```

```
10 #    9.0       0.1      7.       1.       0.    0.5    51      1     5.0
       0.1
11 #    9.0       0.2      8.       2.       0.    0.51   52      1     2.0
       0.2
12
13
14
15 def create_aerodyn_cases(outPath,minimumRotorSpeed,maximumRotorSpeed,
   minimumPitch,maximumPitch,minimumWindSpeed,maximumWindSpeed,
16                          powerLawExp,speedStep=0.1,pitchStep=0.1,
   windSpeedSteps=0.1,
17                          dT=0.1,Tmax=100,maxSimultaneousSimulations
   =10000):
18     """Creates the table structure for multiple aerodyn cases.
19     Its intended for 3d interpolation using wind-rotor speed-pitch
20
21     Args:
22         outPath (str): out file path for csv generation. The file
   name should be provided without csv
23         minimumRotorSpeed (float): minimum rotor speed in rpm
24         maximumRotorSpeed (float): maximum rotor speed in rpm
25         minimumPitch (float): minimum pitch in deg
26         maximumPitch (float): maximum pitch in deg
27         minimumWindSpeed (float): minimum wind speed in m/s
28         maximumWindSpeed (float): maximum wind speed in m/s
29         powerLawExp (float): power law exponential for shear exponent
30         speedStep (float, optional): rotor speed step in rpm.
   Defaults to 0.1.
31         pitchStep (float, optional): rotor speed step in rpm.
   Defaults to 0.1.
32         windSpeedSteps (float, optional): wind speed steps in rpm.
   Defaults to 0.1.
33         dT (float, optional): time step for aerodyn cases. Defaults
   to 0.1.
34         Tmax (int, optional): maximum time for case simulation.
   Defaults to 100.
35         maxSimultaneousSimulations (int, optional): maximum
   simultaneous simulations for parallelization
36     """
37     # HWndSpeed  PLExp  RotSpd  Pitch   Yaw   dT    Tmax  DOF
   Amplitude Frequency
```

```
38      df = pd.DataFrame(columns=["HWndSpeed","PLExp","RotSpd","Pitch","
    Yaw","dT","Tmax","DOF","Amplitude","Frequency"])
39
40      for rs in np.arange(minimumRotorSpeed,maximumRotorSpeed+speedStep
    ,speedStep):
41          for ws in np.arange(minimumWindSpeed,maximumWindSpeed+
    windSpeedSteps,windSpeedSteps):
42              # if tsr is already in the list, exclude the analysis
43              for pitch in np.arange(minimumPitch,maximumPitch+
    pitchStep,pitchStep):
44                  # iterates over three values and generates a table
45                  df.loc[len(df)]=[ws,powerLawExp,rs,pitch,0,dT,Tmax
    ,0,0,0]
46                  #
47      #
48      print("Nb. of cases: "+str(len(df)))
49      # generate dataframe
50      if len(df)<=maxSimultaneousSimulations:
51          df.to_csv(outPath+".csv",sep="\t",index=False)
52      else:
53          ii = 1
54          count = 0
55          while count<len(df):
56              if count+maxSimultaneousSimulations>len(df):
57                  df.loc[count:len(df)].to_csv(outPath+"_"+str(ii)+".
    csv",sep="\t",index=False)
58              else:
59                  df.loc[count:count+maxSimultaneousSimulations-1].
    to_csv(outPath+"_"+str(ii)+".csv",sep="\t",index=False)
60              #
61              ii+=1
62              count+=maxSimultaneousSimulations
63
64
65  def create_aerodyn_cases_fixed_wind(outPath,minimumRotorSpeed,
    maximumRotorSpeed,minimumPitch,maximumPitch,
66                          powerLawExp,fixedWindSpeed=10,speedStep=0.1,
    pitchStep=0.1,
67                          dT=0.1,Tmax=100,maxSimultaneousSimulations
    =10000):
68      """Creates the table structure for multiple aerodyn cases
```

```
69      With fixed wind speed, the interpolation will go through TSR, its
        intended for 2d interpolation
70
71      Args:
72          outPath (str): out file path for csv generation. The file
    name should be provided without csv
73          minimumRotorSpeed (float): minimum rotor speed in rpm
74          maximumRotorSpeed (float): maximum rotor speed in rpm
75          minimumPitch (float): minimum pitch in deg
76          maximumPitch (float): maximum pitch in deg
77          powerLawExp (float): power law exponential for shear exponent
78          fixedWindSpeed (float, optional): wind speed in m/s. Defaults
        to 10.
79          speedStep (float, optional): rotor speed step in rpm.
    Defaults to 0.1.
80          pitchStep (float, optional): rotor speed step in rpm.
    Defaults to 0.1.
81          windSpeedSteps (float, optional): wind speed steps in rpm.
    Defaults to 0.1.
82          dT (float, optional): time step for aerodyn cases. Defaults
    to 0.1.
83          Tmax (int, optional): maximum time for case simulation.
    Defaults to 100.
84          maxSimultaneousSimulations (int, optional): maximum
    simultaneous simulations for parallelization
85      """
86      # create a TSR log for avoiding repetition
87      # HWndSpeed  PLExp  RotSpd  Pitch   Yaw   dT    Tmax  DOF
    Amplitude Frequency
88      df = pd.DataFrame(columns=["HWndSpeed","PLExp","RotSpd","Pitch","
    Yaw","dT","Tmax","DOF","Amplitude","Frequency"])
89      for pitch in np.arange(minimumPitch,maximumPitch+pitchStep,
    pitchStep):
90          for rs in np.arange(minimumRotorSpeed,maximumRotorSpeed+
    speedStep,speedStep):
91              # iterates over three values and generates a table
92              df.loc[len(df)]=[fixedWindSpeed,powerLawExp,rs,pitch,0,dT
    ,Tmax,0,0,0]
93              #
94      #
95      print("Nb. of cases: "+str(len(df)))
96      # generate dataframe
```

```
97      if len(df)<=maxSimultaneousSimulations:
98          df.to_csv(outPath+".csv",sep="\t",index=False)
99      else:
100         ii = 1
101         count = 0
102         while count<len(df):
103             if count+maxSimultaneousSimulations>len(df):
104                 df.loc[count:len(df)].to_csv(outPath+"_"+str(ii)+".
    csv",sep="\t",index=False)
105             else:
106                 df.loc[count:count+maxSimultaneousSimulations-1].
    to_csv(outPath+"_"+str(ii)+".csv",sep="\t",index=False)
107             #
108             ii+=1
109             count+=maxSimultaneousSimulations
110
111 def create_aerodyn_cases_for_tsr(outPath,minimumPitch,maximumPitch,
    minimumTSR,maximumTSR,
112                          powerLawExp,rotorRadius,fixedWindSpeed=10,
    tsrStep=0.1,pitchStep=0.1,
113                          dT=0.1,Tmax=100,maxSimultaneousSimulations
    =10000):
114     """Creates the table structure for multiple aerodyn cases
115     With fixed wind speed, and a given TSR, calculates wind speeds
116
117     Args:
118         outPath (str): out file path for csv generation. The file
    name should be provided without csv
119         minimumPitch (float): minimum pitch in deg
120         maximumPitch (float): maximum pitch in deg
121         minimumTSR (float): minimum TSR
122         maximumTSR (float): maximum TSR
123         powerLawExp (float): power law exponential for shear exponent
124         rotorRadius (float): rotor speed for calculate TSR
125         fixedWindSpeed (float, optional): wind speed in m/s. Defaults
     to 10.
126         tsrStep (float, optional): tsr step. Defaults to 0.1.
127         pitchStep (float, optional): _description_. Defaults to 0.1.
128         dT (float, optional): time step for aerodyn cases. Defaults
    to 0.1.
129         Tmax (int, optional): maximum time for case simulation.
    Defaults to 100.
```

```
130         maxSimultaneousSimulations (int, optional): maximum
    simultaneous simulations for parallelization
131     """
132     # calculates needed rotor speeds based on tsr
133     tsrSteps = np.arange(minimumTSR,maximumTSR+tsrStep,tsrStep)
134     rotorSpeeds = fixedWindSpeed*tsrSteps/rotorRadius
135     rotorSpeeds = rotorSpeeds*30/np.pi # parse from rad/s into rpm
136     # HWndSpeed  PLExp  RotSpd  Pitch   Yaw   dT    Tmax  DOF
    Amplitude Frequency
137     df = pd.DataFrame(columns=["HWndSpeed","PLExp","RotSpd","Pitch","
    Yaw","dT","Tmax","DOF","Amplitude","Frequency"])
138     for pitch in np.arange(minimumPitch,maximumPitch+pitchStep,
    pitchStep):
139         for rs in rotorSpeeds:
140             # iterates over three values and generates a table
141             df.loc[len(df)]=[fixedWindSpeed,powerLawExp,rs,pitch,0,dT
    ,Tmax,0,0,0]
142             #
143     #
144     print("Nb. of cases: "+str(len(df)))
145     # generate dataframe
146     if len(df)<=maxSimultaneousSimulations:
147         df.to_csv(outPath+".csv",sep="\t",index=False)
148     else:
149         ii = 1
150         count = 0
151         while count<len(df):
152             if count+maxSimultaneousSimulations>len(df):
153                 df.loc[count:len(df)].to_csv(outPath+"_"+str(ii)+".
    csv",sep="\t",index=False)
154             else:
155                 df.loc[count:count+maxSimultaneousSimulations-1].
    to_csv(outPath+"_"+str(ii)+".csv",sep="\t",index=False)
156             #
157             ii+=1
158             count+=maxSimultaneousSimulations
```

C.2. Código para la evaluación de los resultados

En este módulo, se detallan las clases para la evaluación de los resultados propios de la simulación de Aerodyn. En este módulo se proporcionan varias opciones para trabajar con

estos ficheros, desde juntar varias superficies provenientes de un cálculo que ha sido paralelizado, hasta hacer un plot de las figuras, pasando por la capacidad de exportar un fichero ".mat" para abrirlo desde matlab y tener entonces los valores para interpolar comodamente.

```
import numpy as np
import pandas as pd
from os import listdir,path
import scipy
import scipy.io
from typing import Dict,Literal,List
import matplotlib.pyplot as plt


AERODYN_OUT_FILE_EXTENSION=".out"
"""Aerodyn extension for out files
"""
# COLUMNS FOR AERODYN
AD_PITCH_COLUMN = "BldPitch1"
"""Blade pitch column name for aerodyn
"""
AD_WIND_COLUMN = "HWindSpeedX"
"""Wind speed column name for aerodyn
"""
AD_ROTOR_SPEED_COLUMN = "RotSpeed"
"""Rotor speed column name for aerodyn
"""
AD_TSR_COLUMN = "RtTSR"
"""TSR column name for aerodyn outputs
"""
AD_CP_COLUMN = "RtFldCp"
"""Power coefficient name for aerodyn outputs
"""
AD_CT_COLUMN = "RtFldCt"
"""Thrust coefficient name for aerodyn outputs
"""
# export column names
PITCH_COLUMN = "pitch"
"""Blade pitch column name for export
"""
TSR_COLUMN = "TSR"
"""TSR column name for export outputs
"""
```

```
39 CP_COLUMN = "Cp"
40 """Power coefficient name for export outputs
41 """
42 CT_COLUMN = "Ct"
43 """Thrust coefficient name for export outputs
44 """
45 CQ_COLUMN = "Cq"
46 """Torque coefficient name for export outputs
47 """
48 WIND_COLUMN = "wind_speed"
49 """Wind speed column name for export outputs
50 """
51 ROTOR_SPEED_COLUMN = "rotor_speed"
52 """Rotor speed column name for export outputs
53 """
54
55 CASE_COLUMN = "case"
56 """Case of study
57 """
58 #
59 # insert here my other code for creating surface images and for
   matlab
60
61
62 def get_mat_export_dict_3d_surface(data:pd.DataFrame):
63         """Returns a dictionary of the pitches vector, rotor speeds
   vector, and wind speeds vector, and correspondent cp matrix for
   create a MAT file to be imported from matlab
64         Returns:
65             Dict[Literal["TSR_mat","pitch_mat","cp_mat"],np.ndarray]:
    The mat dictionary to be exported
66         """
67         # Due to numerical derive, tsr values should be adjusted...
   otherwise matrix wont fit
68         #
69         exptDf = data.copy(deep=True)
70         #
71         #
72         # create new column with zeros
73         #
74         exptDf.sort_values(by=[PITCH_COLUMN,TSR_COLUMN],ascending=
   True)
```

```
75          pitches = np.array(exptDf[PITCH_COLUMN].unique())
76          pitches.sort()
77          rotorSpeeds = np.array(exptDf[ROTOR_SPEED_COLUMN].unique())
78          rotorSpeeds.sort()
79          windSpeeds = np.array(exptDf[WIND_COLUMN].unique())
80          windSpeeds.sort()
81          #
82          cps =[]
83          for pp in pitches:
84              cps.append([])
85              pitchDf = exptDf[exptDf[PITCH_COLUMN]==pp]
86              #
87              for rs in rotorSpeeds:
88                  cps[-1].append([])
89                  rsDf = pitchDf[pitchDf[ROTOR_SPEED_COLUMN]==rs]
90                  #
91                  for ws in windSpeeds:
92                      cps[-1][-1].append(rsDf[rsDf[WIND_COLUMN]==ws][
    CP_COLUMN])
93              #
94          cps=np.array(cps)
95          #
96          matDict:Dict[Literal["TSR_mat","pitch_mat","cp_mat"],np.
    ndarray]={
97              "rotor_speed_mat":rotorSpeeds,
98              "wind_speed_mat":windSpeeds,
99              "pitch_mat":np.deg2rad(pitches),
100             "cp_mat":cps
101         }
102         #
103         return matDict
104
105
106 def get_mat_export_dict(data:pd.DataFrame,rotorRadius:float):
107         """Returns a dictionary of the pitches vector, TSR vector and
     correspondent cp matrix for create a MAT file to be imported from
     matlab
108         Returns:
109             Dict[Literal["TSR_mat","pitch_mat","cp_mat"],np.ndarray]:
     The mat dictionary to be exported
110         """
111         #
```

```
112         exptDf = data.copy(deep=True)
113         #
114         ## This approach is caused due to numerical miss in aerodyn
    exports
115         # a more specific search can be done by using "case" column
    for traceability in aerodyn simulation cases
116         #
117         windSpeed = exptDf[WIND_COLUMN].mean()
118         tsrDict:Dict[str,Dict[str,float]] = {}
119         for rs in exptDf[ROTOR_SPEED_COLUMN].unique():
120             rsStr = "{:.6f}".format(rs)
121             #
122             tsrDict[rsStr] = rs*rotorRadius/windSpeed
123         #
124         # create new column with zeros
125         exptDf["new_TSR"] = np.zeros(len(exptDf))
126         for ii in range(len(exptDf)):
127             #
128             rsStr = "{:.6f}".format(exptDf.loc[ii,ROTOR_SPEED_COLUMN
    ])
129             exptDf.loc[ii,"new_TSR"]=tsrDict[rsStr]
130         #
131         exptDf.sort_values(by=[PITCH_COLUMN,"new_TSR"],ascending=True
    ,inplace=True)
132         pitches = np.array(exptDf[PITCH_COLUMN].unique())
133         pitches.sort()
134         lambdas = np.array(exptDf["new_TSR"].unique())
135         lambdas.sort()
136         #
137         cps =[]
138         for vv in pitches:
139             #since they are already ordered, each cp in the list
    corresponds to
140             # the sorted cp
141             cps.append(list(exptDf[exptDf[PITCH_COLUMN]==vv][
    CP_COLUMN]))
142         cps=np.array(cps)
143         #
144         matDict:Dict[Literal["TSR_mat","pitch_mat","cp_mat"],np.
    ndarray]={
145             "TSR_mat":lambdas,
146             "pitch_mat":pitches,
```

```
147             "cp_mat":cps
148         }
149         #
150         return matDict
151 
152 def evaluate_aerodyn_cases(baseDirectories:List[str],fileName:str,
    outFile:str,outFileMat:str=None):
153     """Reads a list of aerodyn cases and generates a pitch-tsr-cp
    surface with the given data
154 
155     Args:
156         baseDirectories (List[str]): list of the directories where
    simulations are made
157         fileName (str): base file name of simulations
158         outFile (str): csv out file for cp-pitch-tsr
159         outFileMat (str): out mat file
160     """
161     #
162     # create a dataframe
163     evaluatedData = pd.DataFrame(columns=[PITCH_COLUMN,TSR_COLUMN,
    CP_COLUMN,CT_COLUMN,CQ_COLUMN,ROTOR_SPEED_COLUMN,WIND_COLUMN,
    CASE_COLUMN])
164     # check all directories
165     for baseDirectory in baseDirectories:
166         #
167         print("Evaluating directory: {}".format(baseDirectory))
168         # get all files in directory
169         filesInDir = listdir(baseDirectory)
170         for ii,ff in enumerate(filesInDir):
171             #
172             if ii%100==0:
173                 print("\t{} files evaluated".format(ii))
174             # check wether a file is product of calculation
175             if ff[0:len(fileName)]==fileName and ff[-len(
    AERODYN_OUT_FILE_EXTENSION):]==AERODYN_OUT_FILE_EXTENSION:
176                 adCase = path.join(path.join(baseDirectory,ff))
177                 auxDf = pd.read_csv(adCase,sep=r"\s+",skiprows=5)
178                 # delete first row -> contains units
179                 auxDf.drop(0, axis=0, inplace=True)
180                 # obtain pitch, tsr, cp, ct, cT
181                 pitch = np.deg2rad(auxDf[AD_PITCH_COLUMN].astype(np.
    float64).mean()) # parse into rad
```

```
182                 windSpeed = auxDf[AD_WIND_COLUMN].astype(np.float64).
    mean()
183                 rotorSpeed = auxDf[AD_ROTOR_SPEED_COLUMN].astype(np.
    float64).mean()*np.pi/30 # parse into rad/s
184                 tsr = auxDf[AD_TSR_COLUMN].astype(np.float64).mean()
185                 cp = auxDf[AD_CP_COLUMN].astype(np.float64).mean()
186                 ct = auxDf[AD_CT_COLUMN].astype(np.float64).mean()
187                 cq = cp/tsr # obtain torque coefficient
188                 # save into dataframe
189                 evaluatedData.loc[len(evaluatedData)]=[pitch,tsr,cp,
    ct,cq,rotorSpeed,windSpeed,adCase]
190     # export cases to a csv
191     evaluatedData.to_csv(outFile,index=False,sep=';')
192     # export data to .mat
193     if outFileMat!=None:
194         scipy.io.savemat(outFileMat,get_mat_export_dict(evaluatedData
    ),appendmat=True)
195
196 def join_aerodyn_surfaces(files:List[str],outFile:str,outFileMat:str=
    None):
197     """join multiple surface if the analysis has been done
    individually due huge amount of data
198
199     Args:
200         files (List[str]): list of the files generated by the
    previous evaluation
201         outFile (str): name of the output file
202         outFileMat (str, optional): name of the output mat file.
    Defaults to None.
203     """
204     #
205     # create a dataframe for join the previous created files
206     evaluatedData = pd.DataFrame(columns=[PITCH_COLUMN,TSR_COLUMN,
    CP_COLUMN,CT_COLUMN,CQ_COLUMN,ROTOR_SPEED_COLUMN,WIND_COLUMN])
207     #
208     # read each file individualy and join it within the full DF
209     auxDfList = [pd.read_csv(ff,delimiter=';') for ff in files]
210     evaluatedData = pd.concat([evaluatedData,auxDfList])
211     #
212     # export cases to a csv
213     evaluatedData.to_csv(outFile,index=False,sep=';')
214     #
```

```
215     # export data to .mat
216     if outFileMat!=None:
217         scipy.io.savemat(outFileMat,get_mat_export_dict(evaluatedData
    ),appendmat=True)
218 
219 
220 def plot_cp_surface(cpFile:str):
221         cpOptMatrix=pd.read_csv(cpFile,sep=';')
222         cpOptMatrix[PITCH_COLUMN]=np.rad2deg(cpOptMatrix[PITCH_COLUMN
    ])
223         #
224         maxCpPoint = cpOptMatrix.loc[cpOptMatrix[CP_COLUMN].idxmax()]
225         # maxEnergyPitch = maxCpPoint[PITCH_COLUMN]
226         # maxEnergyTSR = maxCpPoint[TSR_COLUMN]
227         print("Maximum Cp point:")
228         print("\tMax. Cp: {}".format(maxCpPoint[CP_COLUMN]))
229         print("\tPitch: {}".format(maxCpPoint[PITCH_COLUMN]))
230         print("\tTSR: {}".format(maxCpPoint[TSR_COLUMN]))
231         print("\tCt: {}".format(maxCpPoint[CT_COLUMN]))
232         print("\tCq: {}".format(maxCpPoint[CQ_COLUMN]))
233         #
234         fig = plt.figure()
235         ax = fig.add_subplot(111)
236         if maxCpPoint[CP_COLUMN]>0.4:
237             lvl = np.linspace(0,0.4,num=10,endpoint=False)
238             lvl = np.append(lvl,np.linspace(0.4,maxCpPoint[CP_COLUMN
    ],num=10,endpoint=True))
239             contour = ax.tricontourf(cpOptMatrix[PITCH_COLUMN],
240                                      cpOptMatrix[TSR_COLUMN],
241                                      cpOptMatrix[CP_COLUMN],
242                                      cmap="RdBu_r",levels=lvl)
243         else:
244             contour = ax.tricontourf(cpOptMatrix[PITCH_COLUMN],
245                                      cpOptMatrix[TSR_COLUMN],
246                                      cpOptMatrix[CP_COLUMN],
247                                      cmap="RdBu_r",levels=np.linspace
    (0,maxCpPoint[CP_COLUMN],num=20,endpoint=True))
248         cbar = fig.colorbar(contour)
249         cbar.set_label('Cp')
250         ax.scatter(maxCpPoint[PITCH_COLUMN],maxCpPoint[TSR_COLUMN],
    marker='D',color='gold')
251         ax.set_xlabel('Pitch [deg]')
```

```
252         ax.set_ylabel('TSR')
253         ax.set_title('$Cp = f(\\beta,\\lambda)$')
254         #
255         #
256         fig.show()
```

Apéndice D

Código para el ajuste de la turbulencia de los vientos

En este anexo, se proporciona el módulo con las funciones de ajuste necesarias para llevar a cabo la extracción de las ganancias de los filtros que definen el comportamiento de la turbulencia del viento

```
1  import numpy as np
2  from numpy import pi as PI,sqrt
3  import matplotlib.pyplot as plt
4  from matplotlib.axes import Axes
5  from scipy.optimize import curve_fit,fmin
6  from typing import Dict
7
8  def nmse(x:np.ndarray,xEst:np.ndarray):
9      normFact = x.max()
10     return np.sum((x/normFact-xEst/normFact)**2)/len(x)
11
12 def parameter_approximation(x,a,b,c,d,e,f):
13     # approximation of n-th order for the wind parameter
14     #
15     return a*x**5+b*x**4+c*x**3+d*x**2+e*x+f
16
17 def fit_nOrder_parameter(xData:np.ndarray,yData:np.ndarray,nOrder:int
       ):
18     aRes=0
19     bRes=0
20     cRes=0
21     dRes=0
22     eRes=0
23     fRes=0
```

```
24      if nOrder==0:
25          pOpt,pCov = curve_fit(lambda x,f:parameter_approximation(x
    ,0,0,0,0,0,f),xData,yData)
26          fRes=pOpt[0]
27      elif nOrder==1:
28          pOpt,pCov = curve_fit(lambda x,e,f:parameter_approximation(x
    ,0,0,0,0,e,f),xData,yData)
29          eRes=pOpt[0]
30          fRes=pOpt[1]
31      elif nOrder==2:
32          pOpt,pCov = curve_fit(lambda x,d,e,f:parameter_approximation(
    x,0,0,0,d,e,f),xData,yData)
33          dRes=pOpt[0]
34          eRes=pOpt[1]
35          fRes=pOpt[2]
36      elif nOrder==3:
37          pOpt,pCov = curve_fit(lambda x,c,d,e,f:
    parameter_approximation(x,0,0,c,d,e,f),xData,yData)
38          cRes=pOpt[0]
39          dRes=pOpt[1]
40          eRes=pOpt[2]
41          fRes=pOpt[3]
42      elif nOrder==4:
43          pOpt,pCov = curve_fit(lambda x,b,c,d,e,f:
    parameter_approximation(x,0,b,c,d,e,f),xData,yData)
44          bRes=pOpt[0]
45          cRes=pOpt[1]
46          dRes=pOpt[2]
47          eRes=pOpt[3]
48          fRes=pOpt[4]
49      elif nOrder==5:
50          pOpt,pCov = curve_fit(lambda x,a,b,c,d,e,f:
    parameter_approximation(x,a,b,c,d,e,f),xData,yData)
51          aRes=pOpt[0]
52          bRes=pOpt[1]
53          cRes=pOpt[2]
54          dRes=pOpt[3]
55          eRes=pOpt[4]
56          fRes=pOpt[5]
57      #
58      return aRes,bRes,cRes,dRes,eRes,fRes
59
```

```
60
61 def fit_curve(xData,yData,fittingThreshold:float=0.0001):
62     # starting like this in order to enter into while loop
63     approximationNMSE =fittingThreshold+1
64     #
65     yEst = np.zeros(len(xData))
66     #
67     aEst=0
68     bEst=0
69     cEst=0
70     dEst=0
71     eEst=0
72     fEst=0
73     # initialize order before enter into while loop
74     nOrder =0
75     while nOrder<5 and fittingThreshold<=approximationNMSE:
76         # estimation of parameters
77         aEst,bEst,cEst,dEst,eEst,fEst = fit_nOrder_parameter(xData,
   yData,nOrder)
78         # getting back the approximated vector
79         yEst = parameter_approximation(xData,aEst,bEst,cEst,dEst,eEst
   ,fEst)
80         # get the normalized mse
81         approximationNMSE = nmse(yData,yEst)
82         # increase order
83         nOrder+=1
84     # info
85     print("")
86     print("############ Approximation NMSE")
87     print("Target: "+"{:+.6e}".format(fittingThreshold))
88     print("Obtained: "+"{:+.6e}".format(approximationNMSE))
89     print("--------------")
90     print("Polynomial of order: "+str(int(nOrder)))
91     if fittingThreshold<=approximationNMSE:
92         print("Max. order of polynomial has been reached, evaluate
   the fitting goal, the result, or consider using a linear between
   each pair of points interpolation instead.")
93     else:
94         print("Constraints were satisfied.")
95     #
96     return aEst,bEst,cEst,dEst,eEst,fEst,yEst
97
```

```
98  def second_order_gain(f,kv,p1,p2):
99      w = 2*PI*f
100     sv = kv**2
101     sv = sv/(1+(w**2)*(p1**2))
102     sv = sv/(1+(w**2)*(p2**2))
103     return sv
104 
105 def err_minimize_filter(f,statPS,p):
106     w=2*PI*f
107     sv=1/np.multiply((1+p[0]**2*w**2),(1+p[1]**2*w**2))
108     q=20*np.log10(sv)-statPS
109     err=np.sum(q**2)
110     return err
111 
112 def generate_ps_gain(ti:float,vm:float,h:float,rr:float,hi:float,f:
    float):
113     k1=h/(1+15*h/hi)
114     k2=sqrt(PI)*rr
115     sp=(22*(ti**2)*vm*k1)/((1+33*k1*f/vm)**(5/3))
116     fp=np.multiply((1+8/3*k2/vm*f),(1+4*k2/vm*f))
117     se=sp/fp
118     #
119     return se
120 
121 def plot_wind_info(wind:str,windDict:Dict[str,Dict[str,any]]):
122     print('Filter values:')
123     print('Kv: '+"{:+.6e}".format(windDict[wind]['kv']))
124     print('a1: '+"{:+.6e}".format(windDict[wind]['a1']))
125     print('a2: '+"{:+.6e}".format(windDict[wind]['a2']))
126 
127     fig = plt.figure()
128     ax = fig.add_subplot()
129     ax.semilogx(windDict[wind]['frecuencies']*2*PI,20*np.log10(
    windDict[wind]['gains']),color='blue',label='Wind spectrum gain')
130     ax.semilogx(windDict[wind]['frecuencies']*2*PI,20*np.log10(
    second_order_gain(windDict[wind]['frecuencies'],windDict[wind]["kv
    "],windDict[wind]['p1'],windDict[wind]['p2'])),color="orange",
    label='SOF')
131     # ax.set_xscale("symlog")
132     # ax.set_yscale("symlog")
133     ax.set_xlabel("Frequencies (Hz)")
134     ax.set_ylabel("Gains (dB)")
```

```
135     ax.legend()
136     fig.show()
137
138
139 np.logspace(-3,1,100)
140 # freqs=np.arange(0.0,10,0.1) # frequencies
141 freqs = np.array([0.0])
142 freqs = np.append(freqs,np.array(np.logspace(-3,1,100)))/(2*PI)
143 # freqs=freqs*(1/(2*PI))
144 ti = 0.1355 # turbulence intensity
145 #
146 h = 150 # height of the nacelle
147 rr = 241.94/2 # rotor radius
148 hlimit = 1000 # lowest inversion, atmosferic limit layer
149 #
150
151 #
152 winds = np.arange(3,25+1,1)
153 kvList = np.array([])
154 p1List = np.array([])
155 p2List = np.array([])
156 filterVals:Dict[str,Dict[str,any]]={}
157 for vm in winds:
158     # wind name
159     windName = "{:.0f}".format(vm)
160     #
161     filterVals[windName]={}
162     # static gain
163     se0 = generate_ps_gain(ti,np.float64(vm),h,rr,hlimit,0)
164     #
165     gains = [generate_ps_gain(ti,np.float64(vm),h,rr,hlimit,f) for f
    in freqs]
166     #
167     # normalized wind PS
168     windPS = 20*np.log10(gains/se0)
169     # popt,pcov = curve_fit(second_order_gain_ps,freqs,gains)
170     # ensure steady state gain by equaling kv to freq=0 power
    spectrum value
171     kv = sqrt(se0)
172
173     pOpt = fmin(lambda p:err_minimize_filter(freqs,windPS,p),x0
    =[0,0])
```

```
174     # popt,pcov = curve_fit(lambda f,p1,p2:second_order_gain_ps(f,kv,
        a1,a2),freqs,gains)
175     # kv = popt[0]
176     p1 = pOpt[0]
177     p2 = pOpt[1]
178     # obtain a from poles
179     a2=p2+p1
180     a1=p1*p2
181     #
182     filterVals[windName]['kv']=kv
183     filterVals[windName]['a1']=a1
184     filterVals[windName]['a2']=a2
185     filterVals[windName]['p1']=p1
186     filterVals[windName]['p2']=p2
187     filterVals[windName]['gains']=gains
188     filterVals[windName]['frecuencies']=freqs
189     filterVals[windName]['static_gain']=se0
190     #
191     kvList = np.append(kvList,kv)
192     p1List = np.append(p1List,p1)
193     p2List = np.append(p2List,p2)
194
195
196 # plot_wind_info("10",filterVals)
197
198 ####
199
200 fittingThreshold = 2e-4
201 #
202 kvA,kvB,kvC,kvD,kvE,kvF,kvEst = fit_curve(winds,kvList,
        fittingThreshold)
203 p1A,p1B,p1C,p1D,p1E,p1F,p1Est = fit_curve(winds,p1List,
        fittingThreshold)
204 p2A,p2B,p2C,p2D,p2E,p2F,p2Est = fit_curve(winds,p2List,
        fittingThreshold)
205 #
206
207 fig = plt.figure()
208 ax1:Axes=None
209 ax2:Axes=None
210 ax3:Axes=None
211 fig, (ax1, ax2, ax3) = plt.subplots(3, 1, layout='constrained')
```

```
212 ax1.plot(winds,kvList, label=r'$k_v$',color='b')
213 ax1.plot(winds,kvEst, label=r'$k_v-estimated$',color='r',linestyle='
    --')
214 ax1.set_xlabel('Wind (m/s)')
215 ax1.set_ylabel(r'$k_v$')
216 ax1.legend()
217 ax1.set_title(r'$k_v\;vs\;wind$')
218 ax2.plot(winds,p1List, label=r'$p_1')
219 ax2.plot(winds,p1Est, label=r'$p_1-estimated$',color='r',linestyle='
    --')
220 ax2.set_xlabel('Wind (m/s)')
221 ax2.set_ylabel(r'$p_1$')
222 ax2.legend()
223 ax2.set_title(r'$p_1\;vs\;wind$')
224 ax3.plot(winds,p2List, label=r'$p_2$')
225 ax3.plot(winds,p2Est, label=r'$p_2-estimated$',color='r',linestyle='
    --')
226 ax3.set_xlabel('Wind (m/s)')
227 ax3.set_ylabel(r'$p_2$')
228 ax3.legend()
229 ax3.set_title(r'$p_2\;vs\;wind$')
230 fig.show()
231
232
233 # TODO -> create an export and review function
234 print("")
235 print("")
236 print("% Lists")
237 print("")
238 print("")
239 print("winds= "+str(winds))
240 print("kvList= "+str(kvList))
241 print("p1List= "+str(p1List))
242 print("p2List= "+str(p2List))
243 print("% Values")
244 print("kvA= "+str(kvA)+";")
245 print("kvB= "+str(kvB)+";")
246 print("kvC= "+str(kvC)+";")
247 print("kvD= "+str(kvD)+";")
248 print("kvE= "+str(kvE)+";")
249 print("kvF= "+str(kvF)+";")
250 #
```

```
251 print("p1A= "+str(p1A)+";")
252 print("p1B= "+str(p1B)+";")
253 print("p1C= "+str(p1C)+";")
254 print("p1D= "+str(p1D)+";")
255 print("p1E= "+str(p1E)+";")
256 print("p1F= "+str(p1F)+";")
257 #
258 print("p2A= "+str(p2A)+";")
259 print("p2B= "+str(p2B)+";")
260 print("p2C= "+str(p2C)+";")
261 print("p2D= "+str(p2D)+";")
262 print("p2E= "+str(p2E)+";")
263 print("p2F= "+str(p2F)+";")
```

Apéndice E

Código del módulo general

En este anexo, se proporciona el módulo general encargado de la coordinación de las diferentes partes necesarias para llevar a cabo el entrenamiento del modelo DDPG.

```
1  import os
2
3  os.environ["KERAS_BACKEND"] = "tensorflow"
4
5  import pandas as pd
6
7  import keras
8  from keras import layers
9
10 import tensorflow as tf
11 import numpy as np
12 import matplotlib.pyplot as plt
13 from Buffer import Buffer
14 from SimpleTurbinePlant import SimpleTurbinePlant
15 from SimulationParameters import create_simulation_parameters_dict,
       get_starting_point
16 from curiosity_exploration import generate_exploration,OUActionNoise
17 from ddpg_configuration import LearningConfiguration,
       NewralNetworkManagement,SimulationConfiguration
18 from typing import Dict,Literal,List
19 import matplotlib.pyplot as plt
20 import copy
21 import datetime
22 from SimulationLog import SimulationLog
23
24 # defined policy, could be overwritten with other that applies OU or
       gaussian noise or whathever is prefered
```

```
25 def policy(actorModel,train:bool,noisyActor, state, torqueParams:Dict
   [str,any],noiseParameters:Dict[Literal['noise_scalar','noise_decay
   ','desired_distance'],float]):
26     actions = keras.ops.squeeze(actorModel(state))
27     actions = np.array([actions.numpy()])
28     if train:
29         # adds noise to weights in actor to create noisy actor
30         noisyWeights = actorModel.get_weights()
31         for i in range(len(noisyWeights)):
32             noisyWeights[i] += np.random.randn()*noiseParameters['
   noise_scalar']
33         # noisy actor
34         noisyActor.set_weights(noisyWeights)
35         #
36         actionsNoised = keras.ops.squeeze(noisyActor(state)).numpy()
37         actionsNoised = np.array([actionsNoised])
38         # meassure the distance between the action values from the
   regular and
39         # the noised actor to adjust the amount of noise that will be
    added next round
40         distance = np.sqrt(np.mean(np.square(actions-actionsNoised)))
41             # adjust the amount of noise given to the actor_noised
42         if distance > noiseParameters['desired_distance']:
43             noiseParameters['noise_scalar'] *= noiseParameters['
   noise_decay']
44         elif distance < noiseParameters['desired_distance']:
45             noiseParameters['noise_scalar'] /= noiseParameters['
   noise_decay']
46         #
47         # get the actions normalized
48         normalizedAction = copy.deepcopy(actionsNoised)
49         normalizedAction[0] = np.clip(normalizedAction[0],-1,1)
50         # Scale the actions and make sure actions are within bounds
51         actionsNoised[0] =np.clip(actionsNoised[0]*torqueParams['
   scale_factor'],-torqueParams['scale_factor'],torqueParams['
   scale_factor'])
52         #
53         return actionsNoised,normalizedAction
54     else:
55         # get the actions normalized
56         normalizedAction = copy.deepcopy(actions)
57         normalizedAction[0] = np.clip(normalizedAction[0],-1,1)
```

```
58          # Scale the actions and make sure actions are within bounds
59          actions[0] =np.clip(actions[0]*torqueParams['scale_factor'],-
        torqueParams['scale_factor'],torqueParams['scale_factor'])
60          #
61          return actions,normalizedAction
62
63 ####
64 # create NN management object
65 nnManagement = NewralNetworkManagement(SimpleTurbinePlant.STATES,
        SimpleTurbinePlant.ACTIONS)
66
67 # get the max value of action
68 torqueRateScale = SimpleTurbinePlant.MAX_TORQUE_RATE
69
70 print("Max Value of Torque Rate Action ->  {}".format(torqueRateScale
        ))
71
72 #### NOTE actor and critic could be loaded here instead of being
        created!!
73 actorModel = nnManagement.generate_actor()
74 criticModel = nnManagement.generate_critic()
75
76 targetActor = nnManagement.generate_actor()
77 targetCritic = nnManagement.generate_critic()
78
79 # noisy actor for generate actions exploration -> same structure as
        actor in order to inherit weights
80 noisyActor = nnManagement.generate_actor()
81
82 # Making the weights equal initially
83 targetActor.set_weights(actorModel.get_weights())
84 targetCritic.set_weights(criticModel.get_weights())
85
86 #### NOTE -> could be overwritten
87 # targetActor, actorModel, targetCritic, criticModel =
        NewralNetworkManagement.load_models(desiredLocation=r'/dummy/saved
        /location',modelName=r'model_name')
88 ## if it is loaded, make sure that noisy actor has the same structure
         or create a new one!
89
90 # Learning rate configuration for actor-critic models
```

```
learningConf = LearningConfiguration(criticLr=2e-4,#criticLr=1e-6, 2e-5
                                     actorLr=1e-4,# actorLr=5e-7, 1e-5
                                     tau=0.005, #tau=0.001,
                                     gamma=0.9) # probar con gamma=0.95... gamma=0.99
#############################################################################

### Simulation auxiliar configuration with episodes, backup routes, etc
simulationConfiguration = SimulationConfiguration(logName='logs_name',logOutsFolder=r'/dummy/path/for/logs',
                                                  modelOutName = 'model_name',modelOutFolder=r'/dummy/path/for/models',
                                                  backupStep=10000, # backup each n training steps
                                                  totalEpisodes=30000, # total simulations
                                                  endTime=100, # en time for simulation, start and step could be configured too
                                                  minimumWind=7, maximumWind=9) # torque only winds
#############################################################################

#
# Dictionaries for configure actuations into the policy and for configure parameter noise
#
noiseActive = True
noiseParameters:Dict[Literal['noise_scalar','noise_decay','desired_distance'],float] = {
    'noise_scalar': 0.1,
    'noise_decay':0.95,
    'desired_distance':0.4
}
torquePolicyConfig:Dict[str,any] = {
    'control_torque' : True,
    # noise params
    'use_noise' : None,
```

```
117     'oua_object_noise': None,
118     'exploration_level' : 1, # a function can handle this exploration
        level for the number of simulations
119     'distribution':'uniform',
120     'scale':0.8,
121     'stdDev':None,
122     # default fixed torque when not controlling
123     'fixed_torque_rate' : 0,
124     # scale factor for torque rate
125     'scale_factor' : torqueRateScale,
126     #
127     'max_torque' : SimpleTurbinePlant.MAX_INPUT_TORQUE,
128     'min_torque' : SimpleTurbinePlant.MIN_INPUT_TORQUE
129 }
130
131
132 #
    ##########################################################################

133 # Buffer creation
134 buffer = Buffer(nnManagement.numStates, nnManagement.numActions,
    learningConf.gamma, buffer_capacity=1000000,batch_size=128)
135
136 #
    ##########################################################################

137 # To store reward history of each episode
138 ep_reward_list = []
139 ponderated_ep_reward_list = []
140
141 ## INITIALIZE ENVIRONMENT
142 modelPath = r'/dummy/path/to/simulink/model'
143 # creation of the matlab interface
144 simpleTurbineEnvironment = SimpleTurbinePlant('Simulink_Model_Name',
    modelPath) # simulink name without extension
145 simpleTurbineEnvironment.connect_to_matlab() # stablish connection
146 # configure simulation times
147 simulationTimes=simulationConfiguration.get_time_configuration()
148 simpleTurbineEnvironment.configure_simulation(simulationTimes["
    start_time"],simulationTimes["end_time"],simulationTimes["dT"])
149 # set parameters in workspace for simulation
```

```
150 simulationParameters = create_simulation_parameters_dict(os.path.join
    (modelPath,"cp_surface.mat"))
151 simpleTurbineEnvironment.set_params(simulationParameters)
152 # Set minimum and maximum torque and pitch
153 simpleTurbineEnvironment.set_params({
154     'max_torque':np.float64(SimpleTurbinePlant.MAX_INPUT_TORQUE),
155     'min_torque':np.float64(SimpleTurbinePlant.MIN_INPUT_TORQUE),
156     'max_pitch':np.float64(SimpleTurbinePlant.MAX_INPUT_PITCH),
157     'min_pitch':np.float64(SimpleTurbinePlant.MIN_INPUT_PITCH)
158 })
159 #initialize a dummy wind and speed parameters for not breaking
    simulation when it starts
160 dummyStarts = {
161     'mean_wind':np.float64(8),'wind_seed':np.float64(1),'starting_wg'
    :np.float64(1),
162     "starting_torque":np.float64(1),"starting_pitch":np.float64(1)
163 }
164 simpleTurbineEnvironment.set_params(dummyStarts)
165 # starts simulation
166 simpleTurbineEnvironment.start_simulation()
167 ###
168 # logging
169 # logging learning and rewards
170 lossesData = pd.DataFrame(columns=['critic','actor','reward'])
171 # state names for logging
172 stateNames = ['wind_speed', 'rotor_speed','rotor_acceleration', '
    electrical_power', 'pitch', 'electrical_torque', '
    mechanical_torque','tsr','torque_rate','pitch_rate']
173 # for make the simulation go further in case not even 1 finished
174 epCounter = 0
175 #
176 while epCounter <= simulationConfiguration.totalEpisodes:
177     ##
178     # regenerate wind and seed for each simulation
179     meanWind,windSeed = simulationConfiguration.
    get_episodic_wind_config()
180     simpleTurbineEnvironment.set_params({'mean_wind':meanWind,'
    wind_seed':windSeed})
181     # set the initial state with starting speed, torque, and pitch
182     startingRotorSpeed,startingTorque,startingPitch =
    get_starting_point(meanWind)
183     startingRotorSpeed+=np.random.randn()*0.02
```

```
184     startingTorque+=np.random.randn()*0.01
185     simpleTurbineEnvironment.set_params({ # add a bit noise to speed
    in order to perturb starting point
186         "starting_wg":startingRotorSpeed,
187         "starting_torque":np.float64(startingTorque),
188         "starting_pitch":np.float64(startingPitch)
189         })
190     # simpleTurbineEnvironment.set_control_action(startingTorque,
    startingPitch)
191     # get first states of simulation
192     # _, prevState = simpleTurbineEnvironment.get_states()
193     simpleTurbineEnvironment.reset_simulation()
194     # first step with a 0 rate each
195     state,prevState,_,_,_,time = simpleTurbineEnvironment.step(0,0)
196     #
197     episodicReward = 0
198     #
199     simuLog = SimulationLog(epCounter,meanWind,windSeed,stateNames,['
    torque_actuation'],datetime.datetime.now())
200     #
201     # regenerate noises into each simulation
202     while True:
203         tfPrevState = keras.ops.expand_dims(
204             keras.ops.convert_to_tensor(prevState), 0
205         )
206         action,normalizedAction = policy(actorModel,noiseActive,
    noisyActor, tfPrevState, torquePolicyConfig, noiseParameters)
207         # Receive state and reward from environment.
208         state,normalizedState,reward,truncated,done,time =
    simpleTurbineEnvironment.step(action[0],0)
209         # Store in list in order to save to best and worse buffers
    once simulation has finished
210         #
211         buffer.record((prevState, normalizedAction, reward,
    normalizedState))
212         episodicReward += reward
213         #
214         if learningConf.train==True:
215             criticLoss,actorLoss = buffer.learn(
216                 targetActor,
217                 targetCritic,
218                 #
```

```
219                 criticModel,
220                 actorModel,
221                 #
222                 learningConf.criticOptimizer,
223                 learningConf.actorOptimizer,
224                 False
225                 )
226             # log losses
227             lossesData.loc[len(lossesData)]=[criticLoss.numpy(),
    actorLoss.numpy(),reward]
228             #
229             learningConf.update_target(targetActor,actorModel)
230             learningConf.update_target(targetCritic,criticModel)
231             # stablish backup!!!
232             trainStep +=1
233             simulationConfiguration.check_backup(trainStep,
    targetActor,actorModel,targetCritic,criticModel)
234         ## adds step info to the logger
235         simuLog.add_step_info(time,action,state,normalizedState,
    reward)
236         #
237         # End this episode when `done` or `truncated` is True
238         if done or truncated:
239             simuLog.add_closing_info(done,truncated,datetime.datetime
    .now())
240             #
241             break
242         #
243         prevState = normalizedState
244      # NOTE here the logging is dumped
245      # log only when learning -> same for increase episode, otherwise
    buffer filling
246      if learningConf.train:
247         simuLog.print_screen_info()
248         simuLog.dump_info(simulationConfiguration.logOutsFolder,
    simulationConfiguration.logName)
249         #
250         #
251         ponderated_ep_reward_list.append(simuLog.
    get_ponderated_reward())
252         ep_reward_list.append(episodicReward)
253         #
```

```
254         epCounter+=1
255 
256 # safe disconnect the environment
257 simpleTurbineEnvironment.disconnect()
258 
259 # model dump
260 lossesData.to_csv(os.path.join(simulationConfiguration.modelOutFolder
    ,simulationConfiguration.modelOutName+"_losses.csv"),sep=';')
261 NewralNetworkManagement.save_models(simulationConfiguration.
    modelOutFolder,simulationConfiguration.modelOutName,targetActor,
    actorModel,targetCritic,criticModel)
```

Apéndice F

Código del módulo de Servicios Auxiliares

En este anexo, se proporciona el módulo de Servicios Auxiliares, que está compuesto por diferentes clases que se encargan de gestionar las redes neuronales, los parámetros de aprendizaje, y los parámetros de simulación.

```
1  import keras
2  from keras import layers
3  import numpy as np
4  from typing import Dict,Literal,Tuple
5  import os
6  import random
7
8
9
10 class LearningConfiguration():
11     """Learning parameters configuration
12     """
13     def __init__(self,criticLr:float=1e-7,actorLr:float=5e-8,tau:
   float=0.005,gamma:float=0.995,learningStatus = False):
14         # Learning rate for actor-critic models
15         self.criticLr = criticLr
16         self.actorLr = criticLr
17         #
18         self.criticOptimizer = keras.optimizers.Adam(criticLr)
19         self.actorOptimizer = keras.optimizers.Adam(actorLr)
20         # soft update parameter
21         self.tau = tau
22         # future reward ponderation
23         self.gamma = gamma
```

```
24          #
25          self.train = learningStatus
26
27      def update_learning_rates(self,criticLr:float,actorLr:float):
28          """Update learning rates into optimizers
29
30          Args:
31              criticLr (float): critic learning rate
32              actorLr (float): actor learning rate
33          """
34          # Update learning
35          self.criticLr = criticLr
36          self.criticLr = actorLr
37          #
38          self.criticOptimizer.learning_rate.assign(criticLr)
39          self.actorOptimizer.learning_rate.assign(actorLr)
40
41      def update_target(self,target, original):
42          """Update NN weights with stored tau
43
44          Args:
45              target (keras.Model): target weight
46              original (keras.Model): original weight
47          """
48          target_weights = target.get_weights()
49          original_weights = original.get_weights()
50
51          for i in range(len(target_weights)):
52              target_weights[i] = original_weights[i] * self.tau +
    target_weights[i] * (1 - self.tau)
53
54          target.set_weights(target_weights)
55
56      def calculate_epsilon_greedy(actualEpsilon:float, discountFactor:
    float,minEpsilon:float=0):
57          """Calculates epsilon based on greedy policy decayment of
    epsilon=discount*Epsilon
58
59          Args:
60              actualEpsilon (float): current value of epsilon
61              discountFactor (float): discount factor applied to
    epsilon
```

```
62              minEpsilon (float): Minimum value of epsilon
63
64          Returns:
65              float: The new epsilon value
66          """
67          epsilon = actualEpsilon*discountFactor
68          epsilon = np.clip(epsilon,minEpsilon,1)
69          return epsilon
70
71      def calculate_epsilon_greedy_linear(actualEpsilon:float,
    discountFactor:float,minEpsilon:float=0):
72          """Calculates epsilon greedy based on linear greedy policy
    decayment of epsilon=Epsilon-discount
73
74          Args:
75              actualEpsilon (float): current value of epsilon
76              discountFactor (float): discount factor applied to
    epsilon
77              minEpsilon (float): Minimum value of epsilon
78
79          Returns:
80              float: The new epsilon value
81          """
82          epsilon = actualEpsilon-discountFactor
83          epsilon = np.clip(epsilon,minEpsilon,1)
84          return epsilon
85
86
87
88
89  class NewralNetworkManagement():
90
91      def __init__(self,numStates:int,numActions:int):
92          ####
93          self.numStates = numStates
94          print("Size of State Space -> {}".format(numStates))
95          self.numActions = numActions
96          print("Size of Action Space ->  {}".format(numActions))
97          ###
98
99
100     def generate_actor(self):
```

```
101         # Initialize weights between -3e-3 and 3-e3
102         last_init = keras.initializers.RandomUniform(minval=-0.003,
    maxval=0.003)
103 
104         inputs = layers.Input(shape=(self.numStates,))
105         out = layers.Dense(256, activation="relu")(inputs)
106         out = layers.Dense(512, activation="relu")(out)
107         out = layers.Dense(512, activation="relu")(out)
108         outputs = layers.Dense(self.numActions, activation="tanh",
    kernel_initializer=last_init)(out)
109         #
110         model = keras.Model(inputs, outputs)
111         return model
112 
113     def generate_critic(self):
114         # State as input
115         state_input = layers.Input(shape=(self.numStates,))
116         state_out = layers.Dense(256, activation="relu")(state_input)
117         state_out = layers.Dense(256, activation="relu")(state_out)
118 
119         # Action as input
120         action_input = layers.Input(shape=(self.numActions,))
121         action_out = layers.Dense(64, activation="relu")(action_input
    )
122 
123         # Both are passed through separate layer before concatenating
124         concat = layers.Concatenate()([state_out, action_out])
125 
126         out = layers.Dense(512, activation="relu")(concat)
127         out = layers.Dense(512, activation="relu")(out)
128         out = layers.Dense(512, activation="relu")(out)
129         outputs = layers.Dense(1)(out)
130 
131         # Outputs single value for give state-action
132         model = keras.Model([state_input, action_input], outputs)
133 
134         return model
135 
136     def save_models(desiredLocation:str,modelName:str,targetActor:
    keras.Model, actorModel:keras.Model, targetCritic:keras.Model,
    criticModel:keras.Model):
```

```
137         """Save the DDPG models in a custom location with a model
    name.
138 
139         The saving structure will be:
140             modelName_{Target|Model}_{Actor|Critic}.keras
141 
142         Args:
143             desiredLocation (str): the desired location for the
    models
144             modelName (str): model given name
145             target_actor (keras.Model): target actor model
146             actor_model (keras.Model): actor model
147             target_critic (keras.Model): target critic model
148             critic_model (keras.Model): critic model
149         """
150         #
151         basePath = os.path.join(desiredLocation,modelName)
152         #
153         targetActor.save(basePath+"_Target_Actor"+".keras")
154         actorModel.save(basePath+"_Model_Actor"+".keras")
155         targetCritic.save(basePath+"_Target_Critic"+".keras")
156         criticModel.save(basePath+"_Model_Critic"+".keras")
157         #
158 
159     def load_models(desiredLocation:str,modelName:str)-> Tuple[keras.
    Model,keras.Model,keras.Model,keras.Model]:
160         """Retrieves a set of models for a DDPG model.
161             These models have to have the following structure:
162                 /path/to/model/name_{Target|Model}_{Actor|Critic}.
    keras
163         Args:
164             desiredLocation (str): the path where the model has been
    stored
165             modelName (str): model name
166 
167         Returns:
168             Tuple[keras.Model,keras.Model,keras.Model,keras.Model]:
    The models in following order: [target actor, model actor, target
    critic, model critic]
169         """
170         #
171         basePath = os.path.join(desiredLocation,modelName)
```

```
172         #
173         targetActorName = basePath+"_Target_Actor"+".keras"
174         actorModelName = basePath+"_Model_Actor"+".keras"
175         targetCriticName = basePath+"_Target_Critic"+".keras"
176         criticModelName = basePath+"_Model_Critic"+".keras"
177         #
178         targetActor = keras.models.load_model(targetActorName)
179         modelActor = keras.models.load_model(actorModelName)
180         targetCritic = keras.models.load_model(targetCriticName)
181         modelCritic = keras.models.load_model(criticModelName)
182         #
183         return targetActor, modelActor, targetCritic, modelCritic
184
185
186 class SimulationConfiguration():
187
188     def __init__(self,logName:str,logOutsFolder:str,modelOutName:str,
    modelOutFolder:str,
189                  backupStep:int=100000,
190                  totalEpisodes:int = 30000,
191                  startTime:float=-0.1,endTime:float=100,dT:float=0.1,
192                  minimumWind:float=7,maximumWind:float=13):
193
194         # logging info
195         self.logOutsFolder = logOutsFolder
196         self.logName = logName
197         self.modelOutFolder = modelOutFolder
198         self.modelOutName = modelOutName
199         #
200         self.backupStep = backupStep
201         #
202         self.totalEpisodes = totalEpisodes
203         self.startTime = startTime
204         self.endTime = endTime
205         self.dT = dT
206         #
207         self.minimumWind=minimumWind
208         self.maximumWind=maximumWind
209         #
210         self.backupCounter = 0
211         #
212         pass
```

```
213
214     def check_backup(self,actualStep,targetActor,actorModel,
    targetCritic,criticModel):
215         if actualStep%self.backupStep==0:
216             self.backupCounter+=1
217             NewralNetworkManagement.save_models(self.modelOutFolder,
    self.modelOutName+"_bck_{:02d}".format(self.backupCounter),
    targetActor,actorModel,targetCritic,criticModel)
218             print('.......... Backup {:02d} has been saved into
    model out folder'.format(self.backupCounter))
219
220     def get_time_configuration(self) -> Dict[Literal['start_time','
    end_time','dT'],np.float64]:
221         """Generates time configuration dictionary for simulations
222
223         Returns:
224             Dict[Literal['start_time','end_time','dT'],np.float64]:
    The dictionary
225         """
226         simulationTimes:Dict[Literal['start_time','end_time','dT'],np
    .float64] = {
227             'start_time':np.float64(self.startTime),
228             'end_time':np.float64(self.endTime),
229             'dT':np.float64(self.dT)
230         }
231         return simulationTimes
232
233     def get_episodic_wind_config(self):
234         meanWind = np.float64(random.randint(self.minimumWind,self.
    maximumWind))
235         windSeed = np.float64(random.randint(1,23341))
236         #
237         return meanWind,windSeed
```

Apéndice G

Código del módulo de Buffer

En este anexo, se proporciona el módulo de Buffer, que se encarga del almacenamiento de experiencias previas para el entrenamiento, así como de entrenar las redes neuronales.

```
import numpy as np
import tensorflow as tf
import keras

class Buffer:
    """The Buffer class implements Experience Replay.

    """
    def __init__(self,num_states,num_actions,gamma, buffer_capacity=100000, batch_size=64):
        # Number of "experiences" to store at max
        self.buffer_capacity = buffer_capacity
        # Num of tuples to train on.
        self.batch_size = batch_size

        # Its tells us num of times record() was called.
        self.buffer_counter = 0

        # Instead of list of tuples as the exp.replay concept go
        # We use different np.arrays for each tuple element
        self.state_buffer = np.zeros((self.buffer_capacity, num_states))
        self.action_buffer = np.zeros((self.buffer_capacity, num_actions))
        self.reward_buffer = np.zeros((self.buffer_capacity, 1))
        self.next_state_buffer = np.zeros((self.buffer_capacity, num_states))
```

```
24
25          # Gamma
26          self.gamma=gamma
27
28      # Takes (s,a,r,s') observation tuple as input
29      def record(self, obs_tuple):
30          # Set index to zero if buffer_capacity is exceeded,
31          # replacing old records
32          index = self.buffer_counter % self.buffer_capacity
33
34          self.state_buffer[index] = obs_tuple[0]
35          self.action_buffer[index] = obs_tuple[1]
36          self.reward_buffer[index] = obs_tuple[2]
37          self.next_state_buffer[index] = obs_tuple[3]
38
39          self.buffer_counter += 1
40
41
42      # Eager execution is turned on by default in TensorFlow 2.
    Decorating with tf.function allows
43      # TensorFlow to build a static graph out of the logic and
    computations in our function.
44      # This provides a large speed up for blocks of code that contain
    many small TensorFlow operations such as this one.
45      @tf.function
46      def update(
47          self,
48          gamma,
49          #
50          state_batch,
51          action_batch,
52          reward_batch,
53          next_state_batch,
54          # NOTE added by me!!
55          target_actor,
56          target_critic,
57          #
58          critic_model,
59          actor_model,
60          #
61          critic_optimizer,
62          actor_optimizer
```

```
63
64      ):
65          # Training and updating Actor & Critic networks.
66          # See Pseudo Code.
67          with tf.GradientTape() as tape:
68              target_actions = target_actor(next_state_batch, training=
    True)
69              y = reward_batch + gamma * target_critic(
70                  [next_state_batch, target_actions], training=True
71              )
72              critic_value = critic_model([state_batch, action_batch],
    training=True)
73              critic_loss = keras.ops.mean(keras.ops.square(y -
    critic_value))
74
75          critic_grad = tape.gradient(critic_loss, critic_model.
    trainable_variables)
76          critic_optimizer.apply_gradients(
77              zip(critic_grad, critic_model.trainable_variables)
78          )
79
80          with tf.GradientTape() as tape:
81              actions = actor_model(state_batch, training=True)
82              critic_value = critic_model([state_batch, actions],
    training=True)
83              # Used `-value` as we want to maximize the value given
84              # by the critic for our actions
85              actor_loss = -keras.ops.mean(critic_value)
86
87          actor_grad = tape.gradient(actor_loss, actor_model.
    trainable_variables)
88          actor_optimizer.apply_gradients(
89              zip(actor_grad, actor_model.trainable_variables)
90          )
91
92          return critic_loss,actor_loss
93
94      # We compute the loss and update parameters
95      def learn(self,
96          # NOTE added by me!!
97          target_actor,
98          target_critic,
```

```
99          #
100         critic_model,
101         actor_model,
102         #
103         critic_optimizer,
104         actor_optimizer,
105         addLastExperience):
106         # Get sampling range
107         record_range = min(self.buffer_counter, self.buffer_capacity)
108         # Randomly sample indices
109         batch_indices = np.random.choice(record_range, self.
    batch_size)
110         ## NOTE manually add the last experience
111         if addLastExperience:
112             batch_indices = np.append(batch_indices,(self.
    buffer_counter % self.buffer_capacity)-1)
113
114         state_batch = self.state_buffer[batch_indices]
115         action_batch = self.action_buffer[batch_indices]
116         reward_batch = self.reward_buffer[batch_indices]
117         next_state_batch = self.next_state_buffer[batch_indices]
118
119         ##
120         # Convert to tensors
121         state_batch = keras.ops.convert_to_tensor(state_batch)
122         action_batch = keras.ops.convert_to_tensor(action_batch)
123         reward_batch = keras.ops.convert_to_tensor(reward_batch)
124         reward_batch = keras.ops.cast(reward_batch, dtype="float32")
125         next_state_batch = keras.ops.convert_to_tensor(
    next_state_batch)
126
127         criticLoss, actorLoss = self.update(self.gamma,state_batch,
    action_batch, reward_batch, next_state_batch,
128                     target_actor,target_critic,critic_model,
    actor_model,critic_optimizer,actor_optimizer)
129         #
130         return criticLoss,actorLoss
```

Apéndice H

Código del módulo de configuración de aerogenerador

En este anexo, se proporciona el módulo de de configuración de Aerogenerador, que se encarga de proporcionar los parámetros de máquina necesarios y de calcular los puntos de inicio de las simulaciones.

```
from typing import Dict
import scipy.io as scipio
import numpy as np
import math

FINE_PITCH = 0
RATED_SPEED = 7.56*np.pi/30
TSR_OPT = 8.9107
ROTOR_RADIUS = 241.94/2
RATED_TORQUE = 15e6/RATED_SPEED

def create_simulation_parameters_dict(cpFile:str):
    # Aerodynamic parameters
    # Maximum Cp point:
    #         Max. Cp: 0.48794251475
    #         Pitch: 0.0
    #         TSR: 8.910705836666667
    #         Ct: 0.8007977013333334
    #         Cq: 0.0547591317336686
    paramDict:Dict[str,any] = {}
    paramDict['TSR_opt'] = np.float64(TSR_OPT) # optimall TSR, [-]
    paramDict['air_dens'] = np.float64(1.225) # air density, [kg/m3]
    paramDict['rotor_radius'] = np.float64(241.94/2) # rotor radius [
   m]
```

```
24      paramDict['rated_rotor_speed'] = np.float64(7.56*np.pi/30) #
    rated rotor angular speed, [rad/s]
25      paramDict['rated_wind'] = np.float64(11.1682309213429) # wind at
    wich rotor reaches rated speed
26      paramDict['pitch_ctrl'] = np.float64(1) # usage of pitch control.
     1: yes, 0: no
27      paramDict['cp_opt'] = np.float64(0.48794251475) # optimum cp, [-]
28      paramDict['rated_aerodynamic_power'] = np.float64(1/2*paramDict['
    air_dens']*(np.pi*(paramDict['rotor_radius']**2))*(paramDict['
    rated_wind']**3)*paramDict['cp_opt']) # rated aerodynamic power [W
    ]
29      #
30      # Power could be calculated using the theoretical adjusted
    formula, or using cp surface (its the best option but requires a
    previous generated surface)
31      paramDict['cp_mode'] = 2 # rated cp calculation mode. 1: formula,
     2: lut interpolation
32      if paramDict['cp_mode']==1:
33          # Parameters for cp_formula
34          paramDict['cp_a1']=np.float64(0.5176) # [-]
35          paramDict['cp_a2']=np.float64(116) # [-]
36          paramDict['cp_a3']=np.float64(0.4) # [-]
37          paramDict['cp_a4']=np.float64(5) # [-]
38          paramDict['cp_a5']=np.float64(21) # [-]
39          paramDict['cp_a6']=np.float64(0.0068) # [-]
40          paramDict['cp_b1']=np.float64(0.08) # [-]
41          paramDict['cp_b2']=np.float64(0.035) # [-]
42          paramDict['cp_formula_correction'] = 0.92 # correction for
    cp_formula, in case it get smaller or higher values than expected
    the whole formula could be corrected using this factor, [-]
43      else:
44          # dummy values
45          # Parameters for cp_formula
46          paramDict['cp_a1']=np.float64(1) # [-]
47          paramDict['cp_a2']=np.float64(1) # [-]
48          paramDict['cp_a3']=np.float64(1) # [-]
49          paramDict['cp_a4']=np.float64(1) # [-]
50          paramDict['cp_a5']=np.float64(1) # [-]
51          paramDict['cp_a6']=np.float64(1) # [-]
52          paramDict['cp_b1']=np.float64(1) # [-]
53          paramDict['cp_b2']=np.float64(1) # [-]
```

```
54          paramDict['cp_formula_correction'] = np.float64(1) #
    correction for cp_formula, in case it get smaller or higher values
     than expected the whole formula could be corrected using this
    factor, [-]
55      #
56      # LUT values
57      # direct introduction - uncomment in order to insert hand made
    values
58      # cp_LUT_pitch = [0,1,2,3,4]; % cp LUT steps for pitch [rad/s]
59      # cp_LUT_TSR = [5,6,7,8]; % cp LUT steps for TSR [-]
60      # % pitch (rows) x TSR (columns) []
61      # cp_LUT_values = [0.45,0.44,0.43,0.42;...
62      #                  0.45,0.44,0.43,0.42;...
63      #                  0.45,0.44,0.43,0.42;...
64      #                  0.45,0.44,0.43,0.42;...
65      #                  0.45,0.44,0.43,0.42;...
66      # ]; % [-]
67      # loaded from .mat matrix
68      ## cargar aqui a curva de cp!
69      if paramDict['cp_mode']==2:
70          cpData = scipio.loadmat(cpFile)
71          # load('cp_surface.mat')
72          paramDict['cp_LUT_pitch'] = cpData['pitch_mat'][0].astype(np.
    float64) # cp LUT steps for pitch [rad/s]
73          paramDict['cp_LUT_TSR'] = cpData['TSR_mat'][0].astype(np.
    float64) # cp LUT steps for TSR [-]
74          # pitch (rows) x TSR (columns) []
75          paramDict['cp_LUT_values'] = cpData['cp_mat'].astype(np.
    float64) # [-]
76      else:
77          # dummy values
78          paramDict['cp_LUT_pitch'] = np.array([0,1]) # cp LUT steps
    for pitch [rad/s]
79          paramDict['cp_LUT_TSR'] = np.array([0,1]) # cp LUT steps for
    TSR [-]
80          # pitch (rows) x TSR (columns) []
81          paramDict['cp_LUT_values'] = np.array([[0,1],[0,1]]) # [-]
82
83      # Wind model parameters
84      paramDict['wind_mode'] = 2 # wind model used for simulation. 1:
    custom wind, 2:second order turbulent wind based on mean wind
    speed
```

```
85
86      paramDict['wind_params_mode'] = True # use wind parametrization
    or LUTs. true:parametric formula, false: LUTs
87
88      if paramDict['wind_params_mode'] == False:
89          # vm LUT in table
90          paramDict['vm_LUT']=np.array
    ([3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25]).
    astype(np.float64)
91          #kV LUT out table
92          paramDict['kv_LUT']=np.array
    ([7.4785123,8.63544217,9.65471786,10.57621351,11.42361623,12.21235944,12.9531
    .astype(np.float64)
93          # p1 LUT out table
94          paramDict['p1_LUT']=np.array
    ([220.4571354,164.09484228,130.48972708,108.20327114,92.35485239,80.51398823,
    .astype(np.float64)
95          # p2 LUT out table
96          paramDict['p2_LUT']=np.array
    ([6.24002415,4.85539537,3.99830812,3.41239813,2.9850154,2.65864853,2.40075149
    .astype(np.float64)
97      else:
98          # dummy values
99          paramDict['vm_LUT']=np.array([0,1]).astype(np.float64)
100         # kV LUT out table
101         paramDict['kv_LUT']=np.array([0,1]).astype(np.float64)
102         # p1 LUT out table
103         paramDict['p1_LUT']=np.array([0,1]).astype(np.float64)
104         # p2 LUT out table
105         paramDict['p2_LUT']=np.array([0,1]).astype(np.float64)
106     #
107     #wind parameters for formulas
108     if paramDict['wind_params_mode']:
109         # kv
110         paramDict['kvA']= np.float64(0)
111         paramDict['kvB']= np.float64(0)
112         paramDict['kvC']= np.float64(0)
113         paramDict['kvD']= np.float64(-0.013030576420311135)
114         paramDict['kvE']= np.float64(0.9800766033193007)
115         paramDict['kvF']= np.float64(5.028825808732119)
116         # p1
117         paramDict['p1A']= np.float64(0)
```

```
118         paramDict['p1B']= np.float64(0.00465315396781274)
119         paramDict['p1C']= np.float64(-0.31227247610492137)
120         paramDict['p1D']= np.float64(7.688478791765705)
121         paramDict['p1E']= np.float64(-84.81126789874325)
122         paramDict['p1F']= np.float64(405.2241342840355)
123         # p2
124         paramDict['p2A']= np.float64(0)
125         paramDict['p2B']= np.float64(0.00011039210093836054)
126         paramDict['p2C']= np.float64(-0.007453632772214246)
127         paramDict['p2D']= np.float64(0.18551548558189124)
128         paramDict['p2E']= np.float64(-2.095205143203259)
129         paramDict['p2F']= np.float64(10.854001227716992)
130     else:
131         # kv
132         paramDict['kvA']= np.float64(0)
133         paramDict['kvB']= np.float64(0)
134         paramDict['kvC']= np.float64(0)
135         paramDict['kvD']= np.float64(0)
136         paramDict['kvE']= np.float64(0)
137         paramDict['kvF']= np.float64(1)
138         # p1
139         paramDict['p1A']= np.float64(0)
140         paramDict['p1B']= np.float64(0)
141         paramDict['p1C']= np.float64(0)
142         paramDict['p1D']= np.float64(0)
143         paramDict['p1E']= np.float64(0)
144         paramDict['p1F']= np.float64(1)
145         # p2
146         paramDict['p2A']= np.float64(0)
147         paramDict['p2B']= np.float64(0)
148         paramDict['p2C']= np.float64(0)
149         paramDict['p2D']= np.float64(0)
150         paramDict['p2E']= np.float64(0)
151         paramDict['p2F']= np.float64(1)
152     #
153     # Gearbox parameters
154     paramDict['gearbox_ratio'] = np.float64(1) # gearbox torque ratio
    , [-]
155     paramDict['gearbox_rend'] = np.float64(0.95756) # mechanical
    loses due to gearbox and mechanical friction,
156                                     # electrical loses could be
    inserted here in order to simplify the model since they are
```

```
    equivalent to a mechanical power drop,
157                                     # [-]
158     paramDict['gearbox_speed_ratio'] = np.float64(1/paramDict['
    gearbox_ratio']) # gearbox speed ratio, [-]
159 
160     # PMSG parameters
161     # J = 969952+1836784; % hub inertia + generator inertia, [kg m2]
162     paramDict['J'] = np.float64(3.524605e8) # rotor inertia, source:
    iea15 definition, [kg m2]
163     paramDict['b_vel'] = np.float64(0.05) # rotor friction factor,
    [-]
164     #
165     #
166     return paramDict
167 
168 
169 
170 def get_starting_point(startingWind:float):
171     #
172     # NOTE -> levar esto o tinglado segun meta o vento -> creo que
    tenho que crear unha variable...
173     # paramDict['starting_wg'] = 9.03565994225*8/paramDict['
    rotor_radius'] # generator start speed, [rad/s]
174     if startingWind>10.65: # mirar concretamente cal e o rated...
175         startingRotorSpeed = RATED_SPEED
176         #
177         torque = RATED_TORQUE
178         #
179         if math.isclose(startingWind,7):
180             pitch = np.deg2rad(0)
181         elif math.isclose(startingWind,8):
182             pitch = np.deg2rad(0)
183         elif math.isclose(startingWind,9):
184             pitch = np.deg2rad(0)
185         elif math.isclose(startingWind,10):
186             pitch = np.deg2rad(0)
187         elif math.isclose(startingWind,11):
188             pitch = np.deg2rad(3.3201584521006464)
189         elif math.isclose(startingWind,12):
190             pitch = np.deg2rad(6.338186382504355)
191         elif math.isclose(startingWind,13):
192             pitch = np.deg2rad(8.381345946647068)
```

```
193         else:
194             pitch = FINE_PITCH
195         #
196 
197     else:
198 
199         startingRotorSpeed = TSR_OPT*startingWind/ROTOR_RADIUS
200         if startingRotorSpeed < 0.5:
201             startingRotorSpeed=0.5156228817062082
202         #
203         if math.isclose(startingWind,7):
204             torque = 8683.71748514307e3
205         elif math.isclose(startingWind,8):
206             torque = 11381.9505188598 16e3
207         elif math.isclose(startingWind,9):
208             torque = 14366.72735591954e3
209         elif math.isclose(startingWind,10):
210             torque = 17644.508132714418e3
211         elif math.isclose(startingWind,11):
212             torque = RATED_TORQUE
213         elif math.isclose(startingWind,12):
214             torque = RATED_TORQUE
215         elif math.isclose(startingWind,13):
216             torque = RATED_TORQUE
217         else:
218             torque = RATED_TORQUE
219         #
220         pitch = FINE_PITCH
221         #
222         torque = torque*0.95756 # includes generator performance
223     #
224     return startingRotorSpeed,torque,pitch
```

Apéndice I

Código del módulo de interfaz entre Python y Matlab para el modelo de aerogenerador

En este anexo, se proporciona el módulo de interfaz entre Python y Matlab. Este módulo es clave para poder inicializar y configurar tanto el modelo como el viento. Además tiene la función de ir ejecutando las simulaciones y proporcionando los valores de los estados y las recompensas al módulo principal.

```
1  import matlab.engine
2  import numpy as np
3  import matplotlib.pyplot as plt
4  from typing import List, Dict,Literal,Tuple
5
6  from SimulationParameters import create_simulation_parameters_dict,
       get_starting_point
7
8  class SimpleTurbinePlant:
9      """This class manages a simple turbine model in Simulink
10
11     Returns:
12         None
13     """
14
15     # States:
16     #
17     #   wind speed
18     #   rotor speed
19     #   rotor acceleration
20     #   torque
```

```
21
22
23      STATES = 4
24
25      # Actions:
26      #
27      #  electrical torque from 0 to 1.05*rated
28      ACTIONS = 1
29
30      # GOALS FOR REWARD
31      ELECTRICAL_POWER_GOAL = 15000000
32      ROTOR_SPEED_GOAL = 7.56*np.pi/30 #rad/s
33      TSR_GOAL = 8.9107
34
35      RATED_TORQUE = ELECTRICAL_POWER_GOAL/ROTOR_SPEED_GOAL
36
37      ROTOR_RADIUS = 241.94/2
38
39      MAX_ROTOR_ACCELERATION = 0.5*np.pi/30 # this last value means 0.5
    rpm/s^2 #0.052359877559829696 # normalized to (6.05-6)/0.1 rpm/s^2
     -> rad/s^2
40      MAX_TORQUE_RATE = 4500000 # from Rosco docu
41      MAX_PITCH_RATE = 0.0349 # from Rosco docu
42
43      RATED_WIND = 10.59 # m/s
44
45      # LIMITS FOR TRUNCATED SIMULATION
46      ROTOR_SPEED_LOWER_LIMIT = 4*np.pi/30 # minimum rotor speed
47      ROTOR_SPEED_HIGHER_LIMIT = 8*np.pi/30 #8rpm as max speed, above
    that, the fluid is compresible so BEMT breaks!!
48      ELECTRICAL_POWER_LOWER_LIMIT = 0
49      ELECTRICAL_POWER_HIGHER_LIMIT = 15e6*1.2 # overload of 18MW
50
51      # THRESHOLDS
52      MAX_INPUT_TORQUE = 15e6*1.2/(7.56*np.pi/30) # max.power/rated.
    speed
53      MIN_INPUT_TORQUE = 0
54      #
55      MAX_INPUT_PITCH = np.deg2rad(20)
56      MIN_INPUT_PITCH = 0
57
58      # REWARD FACTORS -> STORED FOR HOT CHANGES DURING CODE EXECUTION
```

```
59      REWARD_FACTORS:Dict[Literal['TSR','power_under_rated','power','
        rotor_speed','acceleration','torque','torque_actuation','
        pitch_actuation',
60                                  'underspeed_penalty','
        overspeed_penalty','underpower_penalty','overpower_penalty'],float
        ] = {
61          'TSR' : 1/0.3,
62          'power_under_rated' : 0.5,
63          'rotor_speed':1/0.003387,
64          'acceleration':0.15,
65          'torque':0.2/0.04,
66          'torque_actuation':0.01,
67          'underspeed_penalty':20,
68          'underpower_penalty':20,
69          'overspeed_penalty':30,
70          'overpower_penalty':30
71      }
72
73      def __init__(self,modelName:str = 'plant',modelPath:str=None):
74          """Initialize class with simulink model parameters
75
76          Args:
77              modelName (str, optional): The model name. Defaults to '
        plant'.
78              modelPath (str, optional): The model path. Defaults to
        None.
79          """
80          self.modelName:str = modelName #The name of the Simulink
        Model (To be placed in the same directory as the Python Code)
81          self.modelPath:str = modelPath # Path of the used model
82          #Logging the variables
83          self.yHist = 0
84          self.tHist = 0
85          #
86          self.initializedModel:bool = False
87
88      def set_control_action(self,torque:float,pitch:float):
89          """Set the control actions
90
91          Args:
92              torque (float): torque control action
93              pitch (float): pitch control action
```

```
94          """
95          #Helper Function to set value of control action
96          self.eng.set_param('{}/in_torque_rate'.format(self.modelName)
    ,'value',str(torque),nargout=0)
97          self.eng.set_param('{}/in_pitch_rate'.format(self.modelName),
    'value',str(pitch),nargout=0)
98
99      def normalize_ab(x,xMax,xMin,a,b):
100         return a+(x-xMin)*(b-a)/(xMax-xMin)
101
102     def get_normalized_states(windSpeed:float,rotorSpeed:float,
    rotorAcceleration:float,electricalPower:float,electricalTorque:
    float,pitch:float,
103                               torqueRate:float, pitchRate:float,
104                               accelerationX:float,accelerationY:float
    ,accelerationZ:float,mX:float,mY:float):
105         """Get state values and return a normalized array centered in
     0
106
107         Args:
108             windSpeed (float): speed of wind in m/s
109             rotorSpeed (float): rotor speed in rad/s
110             rotorAcceleration (float): rotor acceleration in rad/s^2
111             electricalPower (float): electrical power in W
112             pitch (float): pitch position in rad. To be included
113             torqueRate (float): torque rate in Nm/s
114             pitchRate (float): pitch rate in rad/s. To be included
115             accelerationX (float): To Be Included
116             accelerationY (float): To Be Included
117             accelerationZ (float): To Be Included
118             mX (float): To Be Included
119             mY (float): To Be Included
120         """
121         # return np.array([
122         #     windSpeed/10.6-10.6,
123         #     rotorSpeed/SimpleTurbinePlant.ROTOR_SPEED_GOAL-
    SimpleTurbinePlant.ROTOR_SPEED_GOAL,
124         #     rotorAcceleration/SimpleTurbinePlant.
    MAX_ROTOR_ACCELERATION-SimpleTurbinePlant.MAX_ROTOR_ACCELERATION,
125         #     electricalPower/SimpleTurbinePlant.
    ELECTRICAL_POWER_GOAL-SimpleTurbinePlant.ELECTRICAL_POWER_GOAL,
126         #     # pitchRate/SimpleTurbinePlant.MAX_PITCH_RATE,
```

```
127         #     # torqueRate/SimpleTurbinePlant.MAX_TORQUE_RATE # from
    Rosco docu
128         #     electricalTorque/SimpleTurbinePlant.RATED_TORQUE-
    SimpleTurbinePlant.RATED_TORQUE,
129         #     pitch/SimpleTurbinePlant.MAX_INPUT_PITCH-
    SimpleTurbinePlant.MAX_INPUT_PITCH # a ver asi o pitch...
130         #     # accelerationX*0,
131         #     # accelerationY*0,
132         #     # accelerationZ*0,
133         #     # mX*0,
134         #     # mY*0
135         #     ],dtype=np.float32)
136 
137         return np.array([
138             SimpleTurbinePlant.normalize_ab(windSpeed,25,3,-1,1),
139             SimpleTurbinePlant.normalize_ab(rotorSpeed,
    SimpleTurbinePlant.ROTOR_SPEED_HIGHER_LIMIT,SimpleTurbinePlant.
    ROTOR_SPEED_LOWER_LIMIT,-1,1),
140             SimpleTurbinePlant.normalize_ab(rotorAcceleration,
    SimpleTurbinePlant.MAX_ROTOR_ACCELERATION,-SimpleTurbinePlant.
    MAX_ROTOR_ACCELERATION,-1,1),
141             # SimpleTurbinePlant.normalize_ab(electricalPower,
    SimpleTurbinePlant.ELECTRICAL_POWER_HIGHER_LIMIT,
    SimpleTurbinePlant.ELECTRICAL_POWER_LOWER_LIMIT,-1,1),
142             # pitchRate/SimpleTurbinePlant.MAX_PITCH_RATE,
143             # torqueRate/SimpleTurbinePlant.MAX_TORQUE_RATE # from
    Rosco docu
144             SimpleTurbinePlant.normalize_ab(electricalTorque,
    SimpleTurbinePlant.MAX_INPUT_TORQUE,SimpleTurbinePlant.
    MIN_INPUT_TORQUE,-1,1),
145             # SimpleTurbinePlant.normalize_ab(pitch,
    SimpleTurbinePlant.MAX_INPUT_PITCH,SimpleTurbinePlant.
    MIN_INPUT_PITCH,-1,1),
146             # accelerationX*0,
147             # accelerationY*0,
148             # accelerationZ*0,
149             # mX*0,
150             # mY*0
151             ],dtype=np.float32)
152 
153     def get_states(self) -> Tuple[float,Dict[Literal['wind_speed','
    rotor_speed','rotor_acceleration','electrical_power','pitch','
```

```
    electrical_torque','mechanical_torque','tsr','torque_rate','
    pitch_rate'],float],np.ndarray]:
154         """Get the current states value of the simulation
155 
156         Returns:
157             tuple: time,rotorSpeed,electricalPower,pitch,
    accelerationX,accelerationY,accelerationZ,mX,mY
158         """
159         #Helper Function to get Plant Output and Time History
160         # return self.eng.workspace['out.output'],self.eng.workspace
    ['out.tout']
161 
162         time = self.eng.eval('out.tout(end)')
163         # create a dictionary for return the state
164         states:Dict[Literal['wind_speed','rotor_speed','
    rotor_acceleration','electrical_power','pitch','electrical_torque'
    ,'mechanical_torque','tsr','torque_rate','pitch_rate'],float] = {}
165 
166         #   wind speed
167         windSpeed=self.eng.eval('out.out_wind_speed(end)')
168         # rotor_acceleration
169         rotorAcceleration = self.eng.eval('out.out_rotor_acceleration
    (end)')
170         #   rotor speed
171         rotorSpeed=self.eng.eval('out.out_speed(end)')
172         #   electrical power
173         electricalPower=self.eng.eval('out.out_power(end)')
174         #   pitch
175         pitch=self.eng.eval('out.out_pitch(end)')
176         # torque_rate
177         torqueRate = self.eng.eval('out.out_torque_rate(end)')
178         # pitch_rate
179         pitchRate =  self.eng.eval('out.out_pitch_rate(end)')
180         #   NOTE: the following 3 are not considered
181         #   blade 1 x acceleration
182         accelerationX = 0
183         #   blade 1 y acceleration
184         accelerationY = 0
185         #   blade 1 z acceleration
186         accelerationZ = 0
187         #   blade 1 Mx
188         mX = 0
```

```
189         #   blade 1 My
190         mY = 0
191         # other info
192         electricalTorque = self.eng.eval('out.Te(end)')
193         mechanicalTorque = self.eng.eval('out.Tm(end)')
194         #
195         tsr = rotorSpeed*SimpleTurbinePlant.ROTOR_RADIUS/windSpeed
196         #
197         states['wind_speed'] = windSpeed
198         states['rotor_speed'] = rotorSpeed
199         states['rotor_acceleration'] = rotorAcceleration
200         states['electrical_power'] = electricalPower
201         states['pitch'] =pitch
202         states['torque_rate'] = torqueRate
203         states['pitch_rate'] = pitchRate
204         states['tsr'] = tsr
205         states['electrical_torque'] = electricalTorque
206         states['mechanical_torque'] = mechanicalTorque
207         #
208         normStates = SimpleTurbinePlant.get_normalized_states(
    windSpeed,rotorSpeed,rotorAcceleration,electricalPower,
    electricalTorque,pitch,torqueRate,pitchRate,accelerationX,
    accelerationY,accelerationZ,mX,mY)
209         return time,states,normStates
210
211     def connect_to_matlab(self):
212         """Starts matlab and loads the target model
213         """
214         print("Starting matlab")
215         self.eng = matlab.engine.start_matlab()
216
217         print("Connected to Matlab")
218
219         if self.modelPath!=None:
220             print("Model is localted at {}, Matlab path will be
    changed".format(self.modelPath))
221             # change path in order to find the model
222             self.eng.cd(self.modelPath)
223         else:
224             print("Model path is located in execution path, no
    further changes are required")
225
```

```
226
227         #Load the model
228         self.eng.eval("model = '{}'".format(self.modelName),nargout
    =0)
229         self.eng.eval("load_system(model)",nargout=0)
230         # turn off the beep!! anoying as hell...
231         self.eng.eval("beep off",nargout=0)
232
233     def configure_simulation(self,startTime:float,stopTime:float,dT:
    float):
234         """Configures the simulation duration
235
236         Args:
237             startTime (float): start time of simulation
238             stopTime (float): stop time of simulation
239             dT (float): step time of simulation
240         """
241         self.eng.set_param(self.modelName,'StartTime',str(startTime),
    'StopTime', str(stopTime), 'FixedStep', str(dT),nargout=0)
242
243     def start_simulation(self):
244         """Starts simulation on first step"""
245         #Initialize Control Action to 0
246         self.set_control_action(0,0)
247         print("Initialized Model")
248
249         #Start Simulation and then Instantly pause
250         self.eng.set_param(self.modelName,'SimulationCommand','start'
    ,'SimulationCommand','pause',nargout=0)
251         #
252         self.initializedModel = True
253
254     def set_params(self,params:Dict[str,any]):
255         """Sets a list of parameters for simulate"""
256         # modelWorkspace = self.eng.get_param(self.modelName,'
    ModelWorkspace')
257         for pkey,pval in params.items():
258             self.eng.workspace[pkey]=pval
259             # self.eng.eval("get_param('py_ctrl_test','ModelWorkspace
    ').assignin('scale',0.5)")
260             # self.eng.eval("setVariable(get_param('py_ctrl_test','
    ModelWorkspace'),'scale',1)")
```

```
261             # self.eng.eval("get_param({},'ModelWorkspace').assignin
    ({},{});".format(self.modelName,pkey,pval))
262             # modelWorkspace.assignin(pkey,pval)
263             # self.eng.assignin(self.modelName,pkey,pval,nargout=0)
264             # self.eng.setVariable(modelWorkspace,pkey,pval,nargout
    =0)
265             # self.eng.set_param(self.modelName,pkey,pval,nargout=0)
266 
267     # def reset_simulation(self):
268     #     """Resets simulation
269     #     """
270     #     self.eng.set_param(self.modelName,'SimulationCommand','stop
    ',nargout=0)
271     #     self.start_simulation()
272 
273     def reset_simulation(self):
274         """Resets the simulation to the first step
275         """
276         self.eng.set_param(self.modelName, 'SimulationCommand', 'stop
    ',nargout=0)
277         self.eng.set_param(self.modelName,'SimulationCommand','start'
    ,'SimulationCommand','pause',nargout=0)
278 
279     def calculate_reward(self,electricalPower:float,rotorSpeed:float,
    rotorAcceleration:float,torque:float,torqueRate:float,pitchRate:
    float,wind:float,tsr:float):
280         """Calculates the reward
281         """
282         # NORMAL WORKING PENALTIES
283         # tsr deviation under rated
284         tsrDeviationRMS_PU = 0
285         if wind<SimpleTurbinePlant.RATED_WIND:
286             tsrDeviationRMS_PU = SimpleTurbinePlant.REWARD_FACTORS['
    TSR']*np.abs(1-tsr/SimpleTurbinePlant.TSR_GOAL)
287         # electrical deviation penalty
288         if wind<SimpleTurbinePlant.RATED_WIND:
289             # approach to every optPower per wind!!
290             optPower = np.float64(0.5*1.225*(np.pi*(
    SimpleTurbinePlant.ROTOR_RADIUS**2))*(wind**3)*0.4879*0.95756)
291             electricalDeviationRMS_PU = SimpleTurbinePlant.
    REWARD_FACTORS['power_under_rated']*(1-electricalPower/optPower)
    **2
```

```
292         else:
293             electricalDeviationRMS_PU = SimpleTurbinePlant.
    REWARD_FACTORS['power']*(1-electricalPower/SimpleTurbinePlant.
    ELECTRICAL_POWER_GOAL)**2
294         # # rotor speed gives a punishment when its over rated ->
    lets try for the whole range instead...
295         # rotorSpeedDeviationRMS_PU = (1-rotorSpeed/
    SimpleTurbinePlant.ROTOR_SPEED_GOAL)**2
296         rotorSpeedDeviationRMS_PU = 0
297         if wind>SimpleTurbinePlant.RATED_WIND:
298             rotorSpeedDeviationRMS_PU = SimpleTurbinePlant.
    REWARD_FACTORS['rotor_speed']*(1-rotorSpeed/SimpleTurbinePlant.
    ROTOR_SPEED_GOAL)**2
299         else:
300             if rotorSpeed>SimpleTurbinePlant.ROTOR_SPEED_GOAL:
301                 rotorSpeedDeviationRMS_PU = SimpleTurbinePlant.
    REWARD_FACTORS['rotor_speed']*(rotorSpeed/SimpleTurbinePlant.
    ROTOR_SPEED_GOAL)**2
302         # acceleration penalty punishment -> if its too high
303         accelerationPenalty = 0
304         if np.abs(rotorAcceleration)>SimpleTurbinePlant.
    MAX_ROTOR_ACCELERATION:
305             accelerationPenalty = SimpleTurbinePlant.REWARD_FACTORS['
    acceleration']*(rotorAcceleration/SimpleTurbinePlant.
    MAX_ROTOR_ACCELERATION)**2
306         # # torque penalty -> avoid that maximices power using torque
     above rated
307         # speed could be considered
308         torquePenalty=0
309         if wind>SimpleTurbinePlant.RATED_WIND:
310             torquePenalty = SimpleTurbinePlant.REWARD_FACTORS['torque
    ']*(1-torque/SimpleTurbinePlant.RATED_TORQUE)**2
311         else:
312             # during MPPT only when torque exceeds
313             if torque>SimpleTurbinePlant.RATED_TORQUE:
314                 torquePenalty = SimpleTurbinePlant.REWARD_FACTORS['
    torque']*(1-torque/SimpleTurbinePlant.RATED_TORQUE)**2
315         #
316         # ACTUATOR USAGE PENALTIES -> MAKE TORQUE ACCELERATION AND
    PITCH ACCELERATION PENALTIES...
317         actUsageTorque = SimpleTurbinePlant.REWARD_FACTORS['
    torque_actuation']*(torqueRate/SimpleTurbinePlant.MAX_TORQUE_RATE)
```

```
**2
318         #
319         # STOPS AND EMERGENCIES
320         electricalEmergencyPenalty=0
321         if electricalPower>SimpleTurbinePlant.
ELECTRICAL_POWER_HIGHER_LIMIT:
322             electricalEmergencyPenalty = SimpleTurbinePlant.
REWARD_FACTORS['overpower_penalty']
323         # # add penalty for danger speed situation
324         speedEmergencyPenalty=0
325         if rotorSpeed>SimpleTurbinePlant.ROTOR_SPEED_HIGHER_LIMIT:
326             speedEmergencyPenalty= SimpleTurbinePlant.REWARD_FACTORS[
'overspeed_penalty']
327         # #
328         # # add penalty for shutdown
329         electricalShutdownPenalty=0
330         if electricalPower<SimpleTurbinePlant.
ELECTRICAL_POWER_LOWER_LIMIT:
331             electricalShutdownPenalty=SimpleTurbinePlant.
REWARD_FACTORS['underpower_penalty']
332         # #
333         speedShutdownPenalty=0
334         if rotorSpeed<SimpleTurbinePlant.ROTOR_SPEED_LOWER_LIMIT:
335             speedShutdownPenalty=SimpleTurbinePlant.REWARD_FACTORS['
underspeed_penalty']
336         # #
337         reward = -1000*(tsrDeviationRMS_PU+electricalDeviationRMS_PU+
rotorSpeedDeviationRMS_PU+
338                 accelerationPenalty+torquePenalty+
339                 actUsageTorque+
340                 electricalEmergencyPenalty+speedEmergencyPenalty+
341                 electricalShutdownPenalty+speedShutdownPenalty)
342         # #
343         return np.float64(reward)
344         # rotorSpeedDeviationRMS_PU = (SimpleTurbinePlant.
ROTOR_SPEED_GOAL-rotorSpeed)**2
345         # return -np.float64(rotorSpeedDeviationRMS_PU+
speedShutdownPenalty+speedEmergencyPenalty)
346
347     def step(self,torque_actuation:float,pitch_actuation:float) ->
Tuple[Dict[Literal['wind_speed','rotor_speed','rotor_acceleration'
,'electrical_power','pitch','electrical_torque','mechanical_torque
```

```
    ','torque_rate','pitch_rate'],float]
348
        ,np.ndarray,float,bool,bool,float]:
349         """Performs a step in the current simulation applying the
    calculated control inputs.
350             Returns the states and rewards, also provides information
     about whether the simulation
351             went to an undesired place (truncated simulation), or is
    being finished.
352
353         Args:
354             torque_actuation (float): torque actuation provided by
    controller [Nm]
355             pitch_actuation (float): pitch actuation provided by
    controller [rad]
356
357         Returns:
358             states (Dict[Literal['wind_speed','rotor_speed','
    rotor_acceleration','electrical_power','pitch','electrical_torque
    ','mechanical_torque','torque_rate','pitch_rate'],float): array
    with the states
359             reward (float): reward of the step
360             truncated (bool): indicates when the simulation goes to a
     non desired event, ie. overspeed
361             done (bool): indicates when the simulation has finished
362             time (float): simulation time
363         """
364         # if the simulation has not been started do it now
365         if not self.initializedModel:
366             self.start_simulation()
367         #
368         #Set that Control Action
369         self.set_control_action(torque_actuation,pitch_actuation)
370         #Pause the Simulation for each timestep
371         self.eng.set_param(self.modelName,'SimulationCommand','
    continue','SimulationCommand','pause',nargout=0)
372         #
373         # Get the state values
374         # time,np.array([windSpeed,rotorSpeed,electricalPower,pitch,
    accelerationX,accelerationY,accelerationZ,mX,mY],dtype=np.float32)
375         time,states,normalizedStates = self.get_states() # ollo, hai
    que sacar o tempo!!!
```

```
376         #
377         reward:float = self.calculate_reward(states['electrical_power
    '],states['rotor_speed'],states['rotor_acceleration'],states['
    electrical_torque'],torque_actuation,pitch_actuation,states['
    wind_speed'],states['tsr'])
378         # evaluate whether simulation ends by itself or if ended by
    out of range simulations
379         # ie out of range speed.
380         truncated:bool = SimpleTurbinePlant.check_truncated(states['
    electrical_power'],states['rotor_speed'])
381         # check whether simulation is finished
382         done:bool = self.eng.get_param(self.modelName,'
    SimulationStatus') in ['terminating','stopped']
383         #
384         return states,normalizedStates,reward,truncated,done,time
385
386     def check_truncated(electricalPower,rotorSpeed):
387         if (electricalPower>SimpleTurbinePlant.
    ELECTRICAL_POWER_HIGHER_LIMIT or
388             electricalPower<SimpleTurbinePlant.
    ELECTRICAL_POWER_LOWER_LIMIT or
389             rotorSpeed>SimpleTurbinePlant.ROTOR_SPEED_HIGHER_LIMIT or
390             rotorSpeed< SimpleTurbinePlant.ROTOR_SPEED_LOWER_LIMIT):
391             # when simulation is out of bounds, return true in order
    to stop simulation and restart
392             return True
393         else:
394             # otherwise keep going
395             return False
396
397     def disconnect(self):
398         # stops simulation in case it keeps running
399         self.eng.set_param(self.modelName,'SimulationCommand','stop',
    nargout=0)
400         # close sistem
401         self.eng.close_system(self.modelName,0,'closeReferencedModels
    ',True,nargout=0)
402         # close matlab
403         self.eng.quit()
```

Apéndice J

Código del módulo de logging de simulaciones

En este anexo, se proporciona el módulo de logging que permite gestionar el almacenamiento de simulaciones para comprobación y debug.

```
from typing import List,Dict,Literal
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os
import datetime



class SimulationLog():
    """Super class for store simulation logs
    """

    def __init__(self,episode:int, meanWind, windSeed,
    consideredStates:List[str],consideredActions:List[str],startTime:
    datetime.datetime):
        """Initializes the simulation log class

        Args:
            episode (int): simulation episode
            meanWind (float): mean wind in m/s
            windSeed (float): wind seed
            consideredStates (List[str]): List of considered named
    states
```

```
22            consideredActions (List[str]): List of considered actions
    , it has to have the same length and be placed in order with given
     actions
23        """
24        self.episode:int = episode # number of the simulation
25        self.meanWind:float = meanWind
26        self.windSeed:float = windSeed
27        #
28        self.time:List[float] = []
29        self.actions:List[List[float]] = []
30        self.states:List[Dict[str,float]] = []
31        self.normalizedStates:List[List[np.ndarray]] = []
32        self.rewards:List[float] = []
33        #
34        self.completeSimulation:bool = False
35        self.truncatedSimulation:bool = True
36        #
37        self.stateKeys:List[str] = consideredStates
38        self.actionKeys:List[str] = consideredActions
39        #
40        self.alreadyGeneratedDataframe:bool=False
41        self.generatedDataframe:pd.DataFrame = None
42        self.startTime = startTime
43        #
44        self.ponderatedReward:float = None
45
46    def add_step_info(self,time:float,action:List[float],states:Dict[
    str,float],normalizedState:List[np.ndarray],reward:float):
47        """Adds a step info to the logger class
48
49        Args:
50            time (float): _description_
51            action (List[float]): Actions in shape [torque,pitch]
52            states (Dict[str,float]): States of the model during
    current step
53            normalizedState (List[np.ndarray]): Normalized step for
    the model
54            reward (float): Obtained reward
55        """
56        self.time.append(time)
57        self.actions.append(action)
58        self.states.append(states)
```

```
59          self.normalizedStates.append(normalizedState)
60          self.rewards.append(reward)
61
62      def add_closing_info(self,completeSimulation:bool,
    truncatedSimulation:bool,endTime:datetime.datetime):
63          """Stores how the simulation has finished
64
65          Args:
66              completeSimulation (bool): Whether the simulation has
    been succesfully completed or not
67              truncatedSimulation (bool): Wether the simulation has
    been truncated due to missconduct or not
68          """
69          self.completeSimulation = completeSimulation
70          self.truncatedSimulation = truncatedSimulation
71          self.endTime = endTime
72
73      def plot_simulation_summary(self):
74          """Prints a summary info for the simulation. It shows various
     images aswell
75          """
76          #simulationLog:List[Dict[Literal['episode','mean_wind','
    wind_seed','actions','states','normalized_state','rewards','times
    ','complete_simulation','truncated_simulation'],any]],position):
77          #
78          data = self.log_variables_to_dataframe()
79          print("Simulation summary for simulation {}".format(self.
    episode))
80          print("")
81          print("\tMean Wind: {}".format(self.meanWind))
82          print("\tWind Seed: {}".format(self.windSeed))
83          print("\tTotal Reward: {}".format(np.sum(self.rewards)))
84          print("\t"+"Mean TSR:{:.4E} ".format(data['tsr'].mean()))
85          print("\tPonderated Reward: {}".format(np.sum(self.rewards)/
    len(self.rewards)))
86          print("\tEnd condition: {}".format("Trucated" if self.
    truncatedSimulation else "Normal"))
87          #
88          data = self.log_variables_to_dataframe()
89          #
90          fig, ax = plt.subplots(nrows=2, ncols=2)
```

```
91          fig.suptitle('States for simulation: {}'.format(self.episode)
    )
92          #
93          ax[0,0].plot(data['time'],data['wind_speed'])
94          ax[0,0].set_title('Wind speed')
95          ax[0,0].set_xlabel('Time')
96          ax[0,0].set_ylabel('Wind Speed [m/s]')
97          #
98          ax[0,1].plot(data['time'],data['electrical_power'])
99          ax[0,1].set_title('Electrical Power')
100         ax[0,1].set_xlabel('Time')
101         ax[0,1].set_ylabel('Electrical Power [W]')
102         #
103         ax[1,0].plot(data['time'],data['rotor_speed']*30/np.pi)
104         ax[1,0].set_title('Rotor Speed')
105         ax[1,0].set_xlabel('Time')
106         ax[1,0].set_ylabel('Rotor Speed [rpm]')
107         #
108         ax[1,1].plot(data['time'],np.rad2deg(data['pitch']))
109         ax[1,1].set_title('Pitch')
110         ax[1,1].set_xlabel('Time')
111         ax[1,1].set_ylabel('Pitch [deg]')
112         fig.show()
113
114         fig, ax = plt.subplots(nrows=3, ncols=1)
115         fig.suptitle('Actions for simulation: {}'.format(self.episode
    ))
116         ax[0].plot(data['time'],self.rewards)
117         ax[0].set_title('Rewards')
118         ax[0].set_xlabel('Time')
119         ax[0].set_ylabel('Reward [-]')
120         #
121         ax[1].plot(data['time'],data['torque_actuation'])
122         ax[1].set_title('Torque Actuation')
123         ax[1].set_xlabel('Time')
124         ax[1].set_ylabel('Torque Rate [Nm/s]')
125         #
126         ax[2].plot(data['time'],np.zeros(len(data['time'])))
127         ax[2].set_title('Pitches')
128         ax[2].set_xlabel('Time')
129         ax[2].set_ylabel('Pitch Rate [deg/s]')
130         fig.show()
```

```
131
132         fig, ax = plt.subplots(nrows=1, ncols=1)
133         fig.suptitle('Electrical vs Mechanical Torque in simulation:
    {}'.format(self.episode))
134         ax.plot(data['time'],data['electrical_torque'],label='
    Electrical Torque',color='firebrick')
135         ax.plot(data['time'],data['mechanical_torque'],label='
    Mechanical Torque',linestyle='--',color='forestgreen',alpha=0.5)
136         ax.set_xlabel('Time')
137         ax.set_ylabel('Torque [Nm]')
138         #
139         fig.show()
140
141         fig, ax = plt.subplots(nrows=1, ncols=1)
142         fig.suptitle('TSR tracking in simulation: {}'.format(self.
    episode))
143         ax.plot(data['time'],data['tsr'],label='TSR',color='firebrick
    ')
144         ax.hlines([8.9107],[data['time'].min()],[data['time'].max()],
    label='Objective TSR',linestyle='--',color='forestgreen',alpha
    =0.5)
145         ax.set_xlabel('Time')
146         ax.set_ylabel('TSR [-]')
147         #
148         fig.show()
149
150     def log_variables_to_dataframe(self):
151         """Returns a dataframe with the state results of simulation
152
153         Args:
154             simulationLog (List[Dict[Literal['episode','mean_wind','
    wind_seed','actions','states','normalized_state','rewards','times
    ','complete_simulation','truncated_simulation'],any]]): data
    logger container
155             position (int): position into the data logger
156
157         Returns:
158             DataFrame: a dataframe with the compressed data
159         """
160         #
161         if self.alreadyGeneratedDataframe==False:
162             cols:List[str] = ['time']
```

```
163             cols.extend(self.stateKeys)
164             cols.extend(self.actionKeys)
165             cols.append('reward')
166             data = pd.DataFrame(columns=cols)
167             #
168             for t,state,action,reward in zip(self.time,self.states,
    self.actions,self.rewards):
169                 row = [t]
170                 row.extend([state[kk] for kk in self.stateKeys])
171                 #
172                 row.extend([act for act in action])
173                 #
174                 row.append(reward)
175                 #
176                 data.loc[len(data)] = row
177             #
178             self.generatedDataframe = data
179             self.alreadyGeneratedDataframe = True
180         #
181         return self.generatedDataframe
182 
183     def dump_info(self,destinationDirectory:str, simulationName:str):
184         """Dumps the log information into a directory. The name
    structure will be
185         /dest/directory/fileName_ep{episode}_summary.log
186         /dest/directory/fileName_ep{episode}_data.csv
187 
188         Where the summary information will include a header with the
    basis and simulation status,
189         and data will have the simulation points
190 
191         Args:
192             destinationDirectory (str): directory where create the
    simulation summary
193             simulationName (str): simulation name to assign
194         """
195         #
196         basePath = os.path.join(destinationDirectory,simulationName)
197         data = self.log_variables_to_dataframe()
198         #
199         data.to_csv(basePath+"_ep{:06d}_data.csv".format(self.episode
    ),sep=';',index=False)
```

```
200         #
201         with open(basePath+"_ep{:06d}_summary.log".format(self.
    episode),mode='w') as summaryFile:
202             summaryFile.write("Summary file for simulation {}".format
    (simulationName)+"\n")
203             summaryFile.write("\t"+"Generated at {}".format(datetime.
    datetime.now())+"\n")
204             summaryFile.write("\t"+"Starting time {}".format(self.
    startTime)+"\n")
205             summaryFile.write("\t"+"End time {}".format(self.endTime)
    +"\n")
206             summaryFile.write("\t"+"Elapsed time {}".format(self.
    endTime-self.startTime)+"\n")
207             summaryFile.write(''+"\n")
208             summaryFile.write('
    ================================================================================
    '+"\n")
209             summaryFile.write('Simulated time (s): {:.2f}'.format(np.
    sum(self.time[-1]))+"\n")
210             summaryFile.write('Total Reward: {:.6f}'.format(np.sum(
    self.rewards))+"\n")
211             summaryFile.write('Ponderated Reward: {:.6f}'.format(self
    .get_ponderated_reward())+"\n")
212             summaryFile.write("Mean Wind: {}".format(self.meanWind)+"
    \n")
213             summaryFile.write("Wind Seed: {}".format(self.windSeed)+"
    \n")
214             summaryFile.write("End condition: {}".format("Trucated"
    if self.truncatedSimulation else "Normal")+"\n")
215             #
216             summaryFile.write(''+"\n")
217             summaryFile.write('
    ========================================ACTIONS
    ========================================'+"\n")
218             for actName,action in zip(self.actionKeys,self.actions):
219                 summaryFile.write('Action {} => Mean Value: {:.4f}  *
      Std.Dev: {:.4f}'.format(actName,np.mean(action),np.std(action))+
    "\n")
220             #
221             summaryFile.write(''+"\n")
222             summaryFile.write('
    ========================================STATES
```

```
    ==========================================='+"\n")
223             for stateName in self.stateKeys:
224                 stateVec = [ss[stateName] for ss in self.states]
225                 summaryFile.write('State {} => Mean Value: {:.4f}  *
     Std.Dev: {:.4f}'.format(stateName,np.mean(stateVec),np.std(
    stateVec))+"\n")
226         #
227
228     def print_screen_info(self):
229         data = self.log_variables_to_dataframe()
230         print("
    ================================================================================
    ")
231         print("Episode * {} * Mean Wind ==> {}".format(self.episode,
    self.meanWind))
232         print("\t"+"Ponderated Reward: {} - Simulated time: {:.2f}".
    format(self.get_ponderated_reward(), self.time[-1]))
233         print("\t"+"Mean torque rate:{:.4E} - Mean pitch rate: 0.0000
    ".format(np.array([ac[0] for ac in self.actions]).mean()))
234         print("\t"+"Mean TSR:{:.4E} ".format(data['tsr'].mean()))
235
236     def get_ponderated_reward(self)->float:
237         """Returns the ponderated by time episode reward
238
239         Returns:
240             float: ponderated episode reward
241         """
242         if self.ponderatedReward==None:
243             self.ponderatedReward = np.sum(self.rewards)/len(self.
    rewards)
244         return self.ponderatedReward
```